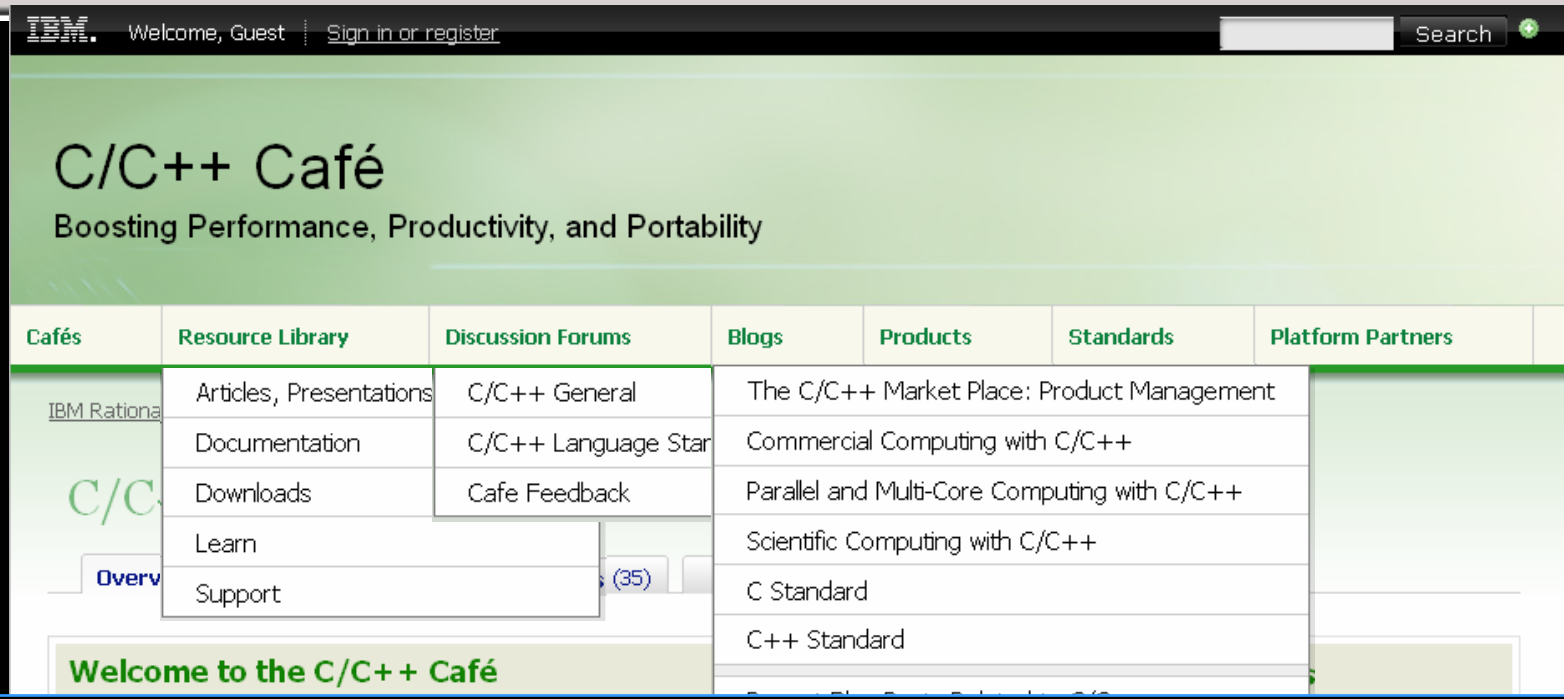
BoostCon 09

Multithreaded C++0x: the Dawn of a new Standard

Michael Wong
michaelw@ca.ibm.com
IBM Toronto Lab
Canadian C++ Standard Committee

The IBM Rational C/C++ Café

ibm.com/rational/cafe/community/ccpp



Agenda

- C++0x, goals, examples
- C++ Standard timeline, state, documents, features
- Compiler staus
- Features and summary
- Q/A

Agenda

- **Concurrent C++0x examples**
- C++ Memory Model
- Concurrent C++ Core Language
- Concurrent C++ Library
- Q/A

Hello Concurrent World

```
#include <iostream>
#include <thread> //#1
void hello() //#2
{
    std::cout<<"Hello Concurrent World"<<std::endl;
}
int main()
{
    std::thread t(hello); //#3
    t.join(); //#4
}
```

Is this valid C++ today? Are these equivalent?

```
int x = 0;

atomic<int> y = 0;

Thread 1:
  x = 17;
  y.store(1,
  memory_order_release);
  // or:       y.store(1);


Thread 2:
  while
  (y.load(memory_order_acquir
  e) != 1)
  // or:       while
  (y.load() != 1)

  assert(x == 17);
```

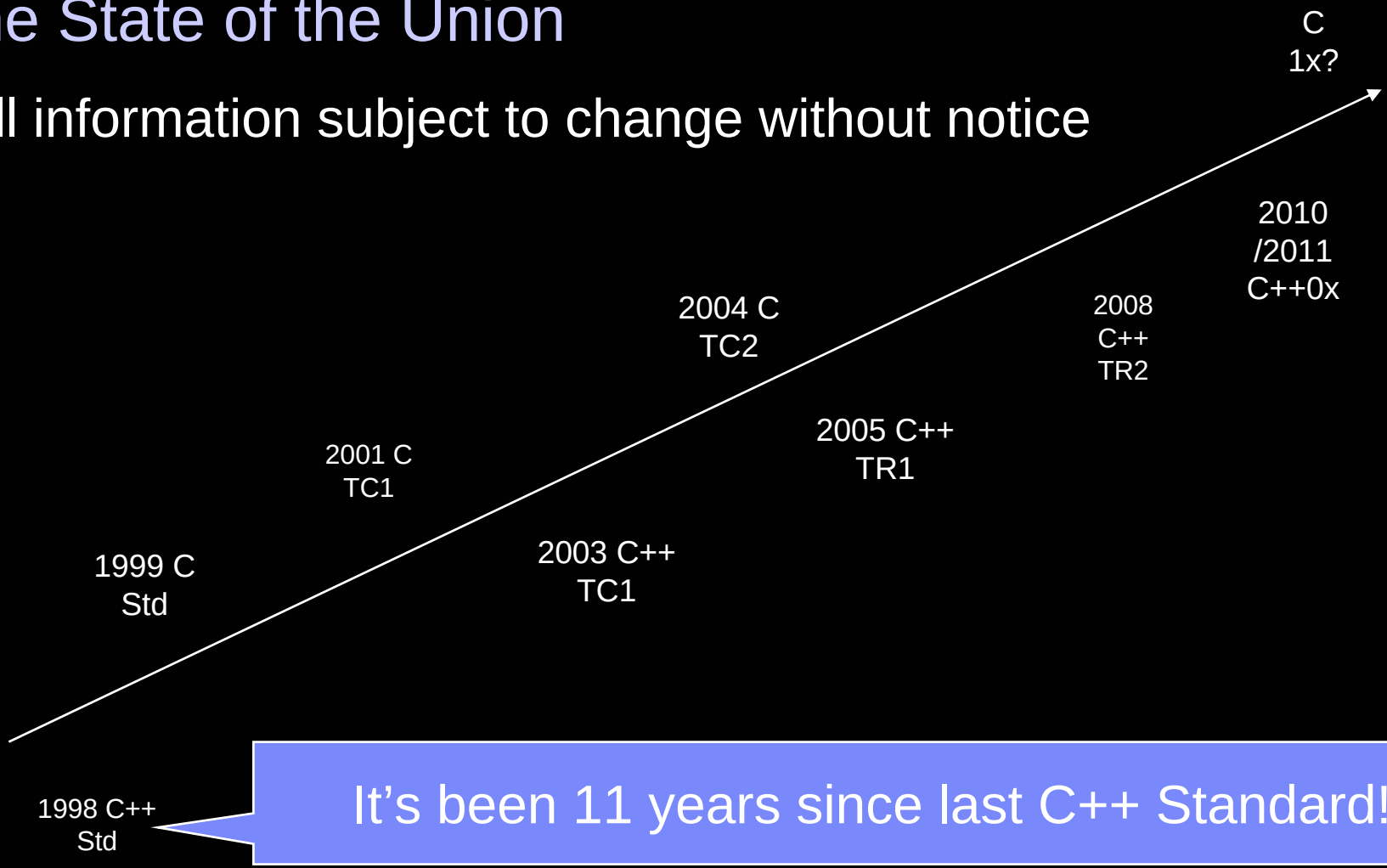
```
int x = 0;

atomic<int> y = 0;

Thread 1:
  x = 17;
  y = 1;

Thread 2:
  while (y != 1)
        continue;
  assert(x == 17);
```

Agenda

- Concurrent C++0x examples
- **C++ Memory Model**
- Concurrent C++ Core Language
- Concurrent C++ Library
- Q/A

The State of the Union

All information subject to change without notice



Concurrency in C++

- **C++98/03: does not have concurrency**
- **C++0x is in public Beta1 with all features in a Committee Draft, aiming for 2009/2010**
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf
- **C++0x will have multithreading support**
  - Memory model, atomics API
  - Language support: TLS, static init, termination, lambda function
  - Library support: thread start, join, terminate, mutex, conditon variable
  - Advanced abstractions: basic futures, thread pools

Concurrency in C++

- **Stroustrup opinion in D&E is that concurrency offered through libraries can support a variety of memory models, and approach builtin concurrency in efficiency and convenience**
- **Recent paper by Hans Boehm believe it is not possible to implement threads as a library**
  - C++ compiler can introduce data races through optimization

The C++ concurrency crew

- **Hans Boehm, Lawrence Crowl, Herb Sutter,** Doug Lea, Peter Dimov, Clark Nelson, Alexander Terekhov, Andrei Alexandrescu, David Callahan, Jeremy Manson, Sarita Adve
- **IBM crew representing PowerPC and weakly-ordered architectures**
  - Raul Silvera, Bob Blainey, Paul Mckenney, Michael Wong
  - Maged Michael, Brad Frey, Cathy May, Christopher Von Praun, Vijay Saraswat, Bill Starke, Derek Williams, Bill Hay, Raymond Mak, Ian McIntosh, Roch Archambault, Alan Kielstra
- **Some slides gathered from presentation in committee**
  - Hans Boehm, Raul Silvera, Paul Mckenney, Herb Sutter, Lawrence Crowl,

Introduction to concurrency

- **Why do we need to standardize concurrency**
  - Multi-core processors
  - Solutions for very large problems
  - Internet programming
- **Standardize existing practice**
  - C++ threads=OS threads
  - shared memory
  - Loosely based on POSIX, Boost thread
  - Does not replace other specifications
    - MPI, OpenMP, UPC, autoparallelization

Concurrency core/library

- **Core: what does it mean to share memory and how it affects variables**
  - **TLS**
  - **Static duration variable initialization/destruction**
  - ***Memory model***
  - ***Atomic operations***
  - ***Fences***
  - *Dependence based Ordering*
- **Library**
  - **How to create/synchronize/terminate threads**,
  - propagate exceptions
  - Advanced abstractions

Memory Model and instruction reordering

- Definitions:
- **Instruction reordering:** When a program executes instructions, especially memory reads and writes, in an order that is different than the order specified in the program's source code.
- **Memory model:** Describes how memory reads and writes may appear to be executed relative to their program order.
- "Compilers, chips, and caches, oh my!"
- Affects the valid optimizations that can be performed by compilers, physical processors, and caches.

Memory Model and Consistency model

- Sequential Consistency (SC)

  Sequential consistency was originally defined in 1979 by Leslie Lamport as follows:

- "… the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program"
- But chip/compiler designers can be annoyingly helpful:
- It can be more expensive to do exactly what you wrote.
- Often they'd rather do something else, that could run faster.
- Most programmers' reaction: "What do you mean, you'll *consider* executing my code the way I wrote it…?!"

Sequential Consistency: a tutorial

- The semantics of the single threaded program is defined by the program order of the statements. This is the strict sequential order. For example:

```
x = 1;

r1 = z;

y = 1;

r2 = w;
```

Sequential Consistency for program understanding

- Suppose we have two threads. Thread 1 is the sequence of statement above. Thread 2 is:

Thread 1:	Thread2:
x = 1;	w=1;
r1 = z;	r3=y;
y = 1;	z=1;
r2 = w;	r4=x;

(All variables are initialized to zero.)

- **2 of 4! Possible interleavings:**

x = 1;	x=1;
w = 1;	**w=1;**
r1 = z;	**r3=y;**
r3 = y;	**z=1;**
y = 1;	**r4=x;**
z = 1;	r1=z;
r2 = w;	y=1;
r4 = x;	r2=w;

Now add fences to control reordering

Thread 1:

x = 1;

r1 = z;

fence();

y = 1;

r2 = w;

Thread2:

w=1;

r3=y;

fence();

z=1;

r4=x;

Is r3==1 and r4==1 possible?

Is r1==1 and r2==1 possible?

Problem today

- **How much and what kind of reordering is allowed?**
  - For programmer understanding?
  - For better performance
- **What motivates each paragraph of clause 1.10, and other new chapters to the C++0x Standard**

1. **The entire software/hardware stack**
2. **Languages and compilers**
3. **Volatile**
4. **Compiler-generated data races**

Problem1: Hardware at the bottom of the stack

- **Everything from threads implementations to user code depends on memory consistency/ordering:**
- **Canonical Example (assume all init with 0, all shared variables):**

Thread 1	**Thread 2**
x=1;	**y=1;**
r1=y; //reads 0	**r2=x; //reads 0**

Can both r1 and r2 be 0?

- Intuitively (or under sequential consistency) no; some thread executes first.
- In practice, yes; compilers, thread library and hardware can reorder.
- **Most hardware will allow this outcome because they have write buffers!**

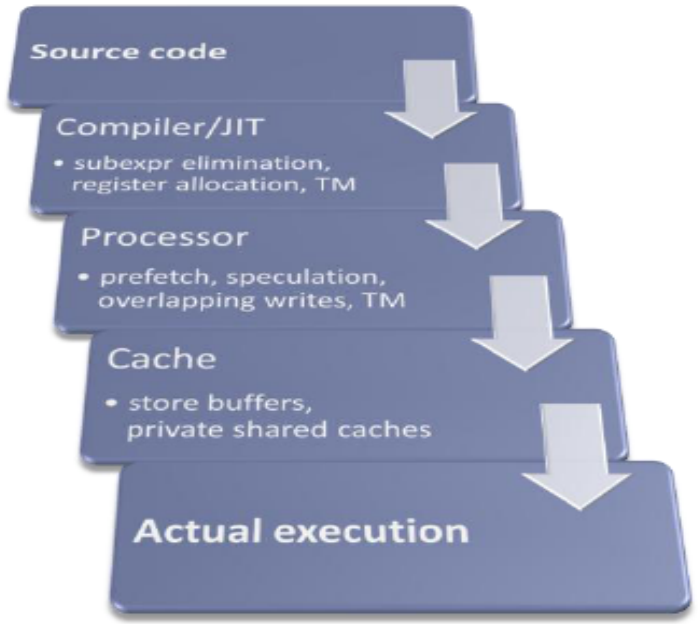
Hardware at the bottom of the stack

- **The hardware doesn't implement what we tell it is fundamentally a problem**
  - If we understand HW rules and can use them to implement a usable programming model
- **Widely held belief**
  - Weaker memory models (e.g. allowing this) is fine, since
    - We only pay for ordering (special fence instructions) when needed
      - Should be cheaper
    - Fence instructions get us sequential consistency exactly when we need it

Transformations: in the name of speed

Transformations:
Reordering + invention + removal

- The level at which the transformation happens is (usually) invisible to the programmer.
- The only thing that matters to the programmer is that a correctly synchronized program behaves as though:
  - The order in which memory operations are actually executed is equivalent to some sequential execution according to program source order.
  - Each write is visible to all processors at the same time.
- Tools and hardware (should) try to maintain that illusion. Sometimes they don't. We'll see why, and what you can do.



Dekker's and Peterson's Algorithms

- Consider (flags are shared and atomic, initially zero):

  Thread 1:

  flag1 = 1; // a: declare intent to enter

  if(flag2 != 0) { ... } // b: detect and resolve contention

  // enter critical section

  Thread 2:

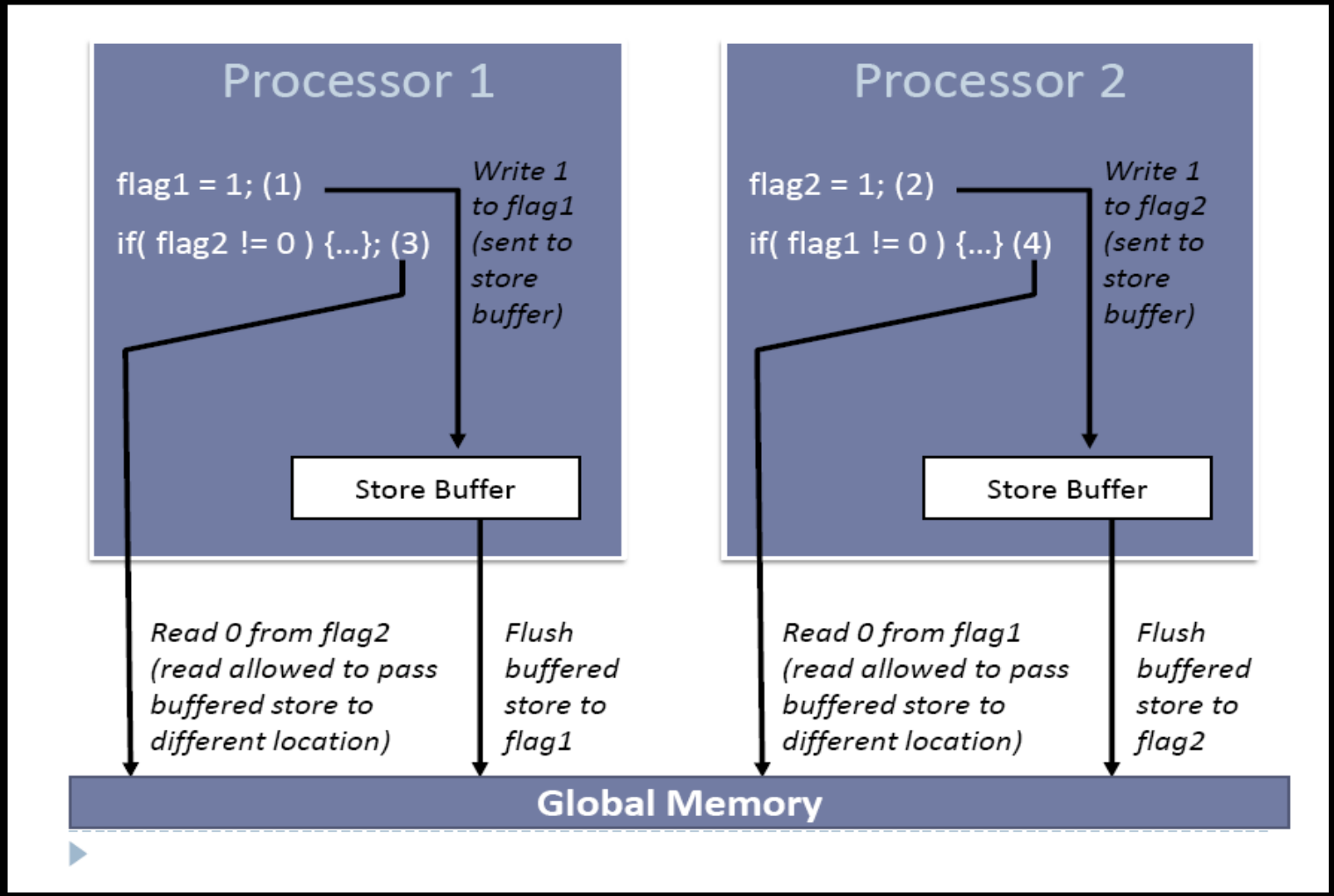
  flag2 = 1; // c: declare intent to enter

  if(flag1 != 0) { ... } // d: detect and resolve contention

  // enter critical section

- Could both threads enter the critical region?
- **Maybe:**If a can pass b, and c can pass d, we could get b->d->a->c.
- Solution 1 (good): Use a suitable atomic type (e.g., Java/.NET "volatile", C++0x **std::atomic<>**) for the flag variables.
- Solution 2 (good?): Use system locks instead of rolling your own.
- Solution 3 (harder but fast): Write a memory barrier after a and c.

What really happens when you are not looking



Memory Ordering and fences in 2006

- **Some architectures have underspecified memory ordering**
  - Confusion
  - Interesting consequences, X86
    - Gcc __sync_synchronize() full memory barrier erroneously generates no-op
    - P4 lfence, sfence instructions appeared to be no-ops in most user code, but the loads and stores are already ordered

Performance

- **Performance of fences and syncs was often neglected**
  - More then 100 cycle, best case on P4
  - Encourages
    - Clever sync avoidance techniques=bugs
  - Can easiy be much more expensive then sequential consistency everywhere(PA-RISC)
- **Some Memory models in which it appeared that fences could not enforce sequential consistency**
  - This means Java memory model is not really implementable

Independent Reads with Independent Writes

- **x,y init to 0, add fences between every instruction**

Thread 1	Thread2	Thread3	Thread4
x=1;	y=1;	r1=x;(1)	r3=y;(1)
		fence;	fence;
		r2=y;(0)	r4=x;(0)
		x set first!	y set first!

Can this be both true?

Architecture in late 2007

- **Intel and AMD published Memory models**
- **Can get sequential consistency examples like the preceding one on X86**
  - But stores to x and y in T1 and T2 need to be implemented with xchg
    - Many JVMs will need to be fixed because they didn't know this rule when they were written
- **Most other vendors are paying attention**

Problem 2: Languages and Compilers

- **Programming rules are unclear, some languages more then others**
  - Java is in best shape since fix in 2005
  - .Net and OpenMP need to be clearer
  - C99 and C++89 has no rules! Other then volatile, sequence points
  - C++0x will hopefully be the best

C++0x programming rules:

- **No simultaneous access from two threads to ordinary shared variables if one access is a write, ie data races are outlawed**
  - Posix C has this rule too, explored in Sarita Adve's Ph.D thesis
  - This rule dates back to at least Ada83
- **No Data races solves many problems:**
  - can't tell whether compiler reorders ordinary memory operations
    - If you could tell, observing thread would race with updating thread
    - Can't tell whether hardware reorders memory operations (as long as locks are handled)
  - C/C++ compilers may rely on the absence of asynchronous changes. This may have weird side effects

Consequence of “no async changes” compiler assumption:

```
unsigned x;
If (x<3) {
  // async x change
  switch (x) {
    case 0: …
     case 1: …
     case 2: …
  }
}
```

- **Assume switch statement compiled as branch table**
- **May assume x is in range**
- **Async change to x causes wild branch**
  - Not just wrong value

C and C++ thread realities before standardization

- **Common attitude that data races aren't so bad**
  - Frequently used idioms rely on benign data races:
    - Approximate counters sometimes without locking an update, and read asynchronously
    - Double-checked locking: lazy init that reads flag outside of critical section
  - Or nonportanle atomic (interlocked, __sync) operations
    - Eg. Reference counting
  - Not well-defined, and read accesses generally appear as data races to compiler
    - This can result in crashed, reads of half-updated values, uninitialized data
    - But locks are expensive enough that this is often impractical to avoid

Problem 3: A Volatile Market

- **2005 java:**

  volatile x_init:

  x=1;

  //possible fence here

  x_init=true;

  This gurantees that x becomes visible to other threads before x_init.

Volatile

- **OpenMP 2.5 and 3.0 Revision 11 Clause 1.4, Pg 15 Line 26-31**

  The **volatile** keyword in the C and C++ languages specifies a consistency mechanism that is related to the OpenMP memory consistency mechanism in the following way: **a reference that reads the value of an object with a volatile-qualified type behaves as if there were a flush operation on that object at the previous sequence point, while a reference that modifies the value of an object with a volatile-qualified type behaves as if there were a flush operation on that object at the next sequence point.**

POSIX Pthread C binding

- **David Butenof:**

“volatile … provide[s] no help whatsoever in making code ‘thread safe’”

C++ 0x WP solution

- **Concurrent access to special atomic objects is allowed**
  - atomic <int>
- **Really just a communication issue**
- **Eliminates all benign data races in C++**
- **C++ volatile continues to have nothing to do with threads**
- **Java volatile= C++ atomic**
- **C++ volatile != Jave volatile**

Problem 4: Compiler generated data races

- **Compiler may generate code that adds data races!**
  - When small struct fields are updated
  - Optimizations leading to spurious writes of old values
  - C++03 allows this!

Adjacent bitfield memory

- Given a global **s** of type struct { int **a:9**; int **b:7**; }:

  Thread 1:

  {lock<mutex> hold(aMutex);s.a = 1;}

  Thread 2:

  {lock<mutex> hold(bMutex);s.b = 1;}

- Is there a race? **Yes in C++0x, in pthreads**
  - **It may be impossible to generate code that will update the bits of** a **without updating the bits of** b**, and vice versa.**
  - **C++0x will say that this is a race. Adjacent bitfields are one "object."**

Adjacent scalar types

- What about two global variables char c; and char d;?

  Thread 1:

  {lock<mutex> hold(cMutex);c = 1;}

  Thread 2:

  {lock<mutex> hold(dMutex);d = 1;}
- Is there a race? **No ideally and in C++0x, but maybe today in pthreads**
- Say the system lays out c then d contiguously, and transforms “d = 1” to:

  char tmp[4]; // 32-bit scratchpad

  memcpy(&tmp[0], &c, 4);// read 32 bits starting at c

  tmp[2] = 1;// set only the bits of d

  memcpy(&c, &tmp[0], 4);// write 32 bits back
- Oops: Thread 2 now silently also writes to c without holding cMutex.

Other things that go Bump in the night

- There are many transformations. Here are two common ones.
- Speculation:
  - **Say the system (compiler, CPU, cache, …) speculates that a condition may be true (e.g., branch prediction), or has reason to believe that a condition is often true (e.g., it was true the last 100 times we executed this code).**
  - **To save time, we can optimistically start further execution based on that guess. If it's right, we saved time. If it's wrong, we have to undo any speculative work.**
- Register allocation:
  - **Say the program updates a variable** x **in a tight loop. To save time: Load** x **into a register, update the register, and then write the final value to** x**.**
- **Key issue: The system must not invent a write to a variable that wouldn't be written to (in an SC execution).**
- If the programmer can't see all the variables that get written to, they can't possibly know what locks to take.

Eliding locks

- Consider (where x is a shared variable, cond does not change):

  if(cond)lock x

  ...

  if(cond)use x

  ...

  if(cond)unlock x
- Q: Is this pattern safe?
- A: In theory, yes. In reality, maybe not…

Write speculation

- Consider (where x is a shared variable):

  if(cond)x = 42;

- Say the system (compiler, CPU, cache, …) speculates (predicts, guesses, measures) that **cond** (may be, will be, often is) true. Can this be transformed to:

```
r1 = x;          // read what's there
x = 42;          // perform an optimistic write
if( !cond)       // check if we guessed wrong
  x = r1;        // oops: back-out write is not SC
```

- In theory, No… **but on some implementations, Maybe.**
  - **Same key issue: Inventing a write to a location that would never be written to in an SC execution.**
  - **If this happens, it can break patterns that conditionally take a lock.**

Register Allocation

- Here's a much more common problem case:

```
void f( /*...params...*/, bool doOptionalWork ) {
   if( doOptionalWork ) xMutex.lock();
   for( ... )
      if( doOptionalWork ) ++x; // write is conditional
   if( doOptionalWork ) xMutex.unlock(); }
```

- A very likely (if deeply flawed) transformation:

```
r1 = x;
for( ... )
   if( doOptionalWork ) ++r1;
x = r1;// oops: write is not conditional
```

- If so, again, it's not safe to have a conditional lock.

Conditional entry

- Here's another variant.
- A write in a loop body is conditional on the loop's being entered!

```
void f( vector<Blah>& v ) {
  if( v.length() > 0 ) xMutex.lock();
  for( int i = 0; i < v.length(); ++i )
    ++x; // write is conditional
  if( v.length() > 0 ) xMutex.unlock(); }
```

- A very likely (if deeply flawed) transformation:

```
r1 = x;
for( int i = 0; i < v.length(); ++i )
  ++r1;
x = r1;// oops: write is not conditional
```

- If so, again, it's not safe to have a conditional lock.

Not all register allocation is Bad

- "What? Register allocation is now a Bad Thing™?!"
  - **No. Only naïve unchecked register allocation is a broken optimization.**
- This transformation is perfectly safe:

```
r1 = x;
for( ... )
   if( doOptionalWork ) ++r1;
if( doOptionalWork ) x = r1;// write is conditional
```

- So is this one ("dirty bit," much as some caches do):

```
r1 = x; bDirty = false;
for( ... )
   if( doOptionalWork ) ++r1, bDirty = true;
if( bDirty ) x = r1;// write is conditional
```

- And so is this one:

```
r1 = 0;
for( ... )
   if( doOptionalWork ) ++r1;
if( r1 != 0 ) x += r1;// write is conditional
```

(note: test is !=, not <)

Lessons learned

- All bets are off in a race:
  - Prefer to use locks **to avoid races and nearly all memory model weirdness, despite the flaws of locks. (In the future: TM?)**
  - **Avoid lock-free code. It's for wizards only, even using SC atomics.**
  - **Avoid fences even more. They're even harder, even full fences.**
- Conditional locks:
  - **Problem: Your code conditionally takes a lock, but your system changes a conditional write to be unconditional.**
  - **Option 1: In code like we've seen, replace one function having a doOptionalWorkflag with two functions (possibly overloaded):**
    - **One function always takes the lock and does the x-related work.**
    - **One function never takes the lock or touches x.**
- Option 2: Pessimistically take a lock for any variables you *mention anywhere* in a region of code.
  - **Even if updates are conditional, and by SC reasoning you could believe you won't reach that code on some paths and so won't need the lock.**

C++0x WP solution:

- **Subject to “no data races” rule:**
  - Each update affects a “memory location”
    - Scalar value, or contiguous sequence of bitfields
  - Define exactly which assignments can be “seen” by each reference to a memory location
  - For ordinary (non-atomic) references, there must be exactly one, for atomics there can be several
  - A reference to x.d after completion of both threads must see a value of 1
  - The preceding implementation of bit-field assignments is incorrect
  - Assignments x.b and x.c bitfields may still interfere

Non-terminating loops

- **Some kinds of code hoisting are problematic.**
- **Stores may not be advanced across potentially nonterminating loops.**
- **Example:**

```
for (T*p = q; p != 0; p = p -> next) ++count;

x = 42;
```

- **Uncommon? But analysis is commonly wrong.**

Subtle implication of the C++0x rule

- **No speculative writes**

```
int count; //global , may be shared between threads
for (p=q; p!=0; p=p->next)
   if (p->data>0) ++ count;
```

- **Cannot transform into**

```
int count; //global , may be shared
reg=count;
for (p=q; p!=0; p=p->next)
   if (p->data>0) ++ reg;
count=reg; //may spuriously assign to count
```

Consequence

- **Outlaws some useful optimizations,**
- **Gives programmer a simple and consistent story**
- **Prevents really mysterious compiler-introduced program bugs**
- **Outlawed optimizations can often be replaced by others**

Memory Model

- **Locks and atomic operations communicate non-atomic writes between two threads**
- **Volatile is not atomics**
- **Memory races cause undefined behavior**
- **Some optimizations are no longer legal**
- **Compiler may assume some loops terminate**

Message shared memory

- **Writes are explicitly communicated**
  - Between pairs of threads
  - Through a lock or an atomic variable
- **The mechanism is acquire and release**
  - One thread releases its memory writes
    - V=32; atomic_store_explicit(&a,3, memory_order_release);
  - Another thread acquires those writes
    - i=atomic_load_explicit(&a, memory_order_acquire); i+v;

Sequencing redefined for serial program

- **Sequence points are … gone!**
- **Sequence are now defined by ordering relations**
  - Sequence-before
  - Indeterminately-sequenced
- **A write/write or read/write pair relations**
  - That are not sequenced before
  - That are not indeterminately-sequenced
  - Results in undefined behaviour

Sequencing extended for parallel programs

- **Sequenced-before**
  - Provides intra-thread ordering
- **Acquire and release**
  - Provide inter-thread ordering
- **Happens-before relation**
  - Between memory operations in different threads

Sequenced before

- **If a memory update or side-effect *a* *is-sequenced-before* another memory operation or side-effect *b*,**
  - then informally *a* must appear to be completely evaluated before *b* in the sequential execution of a single thread, e.g. all accesses and side effects of *a* must occur before those of *b*.
  - We will say that a subexpression *A* of the source program *is-sequenced-before* another subexpression *B* of the same source program to indicate that all side-effects and memory operations performed by an execution of *A* occur-before those performed by the corresponding execution of *B*, i.e. as part of the same execution of the smallest expression that includes them both.
- **We propose roughly that wherever the current standard states that there is a sequence point between *A* and *B*, we instead state that *A* is-sequenced-before *B*. This will constitute the precise definition of *is-sequenced-before* on subexpressions, and hence on memory actions and side effects.**

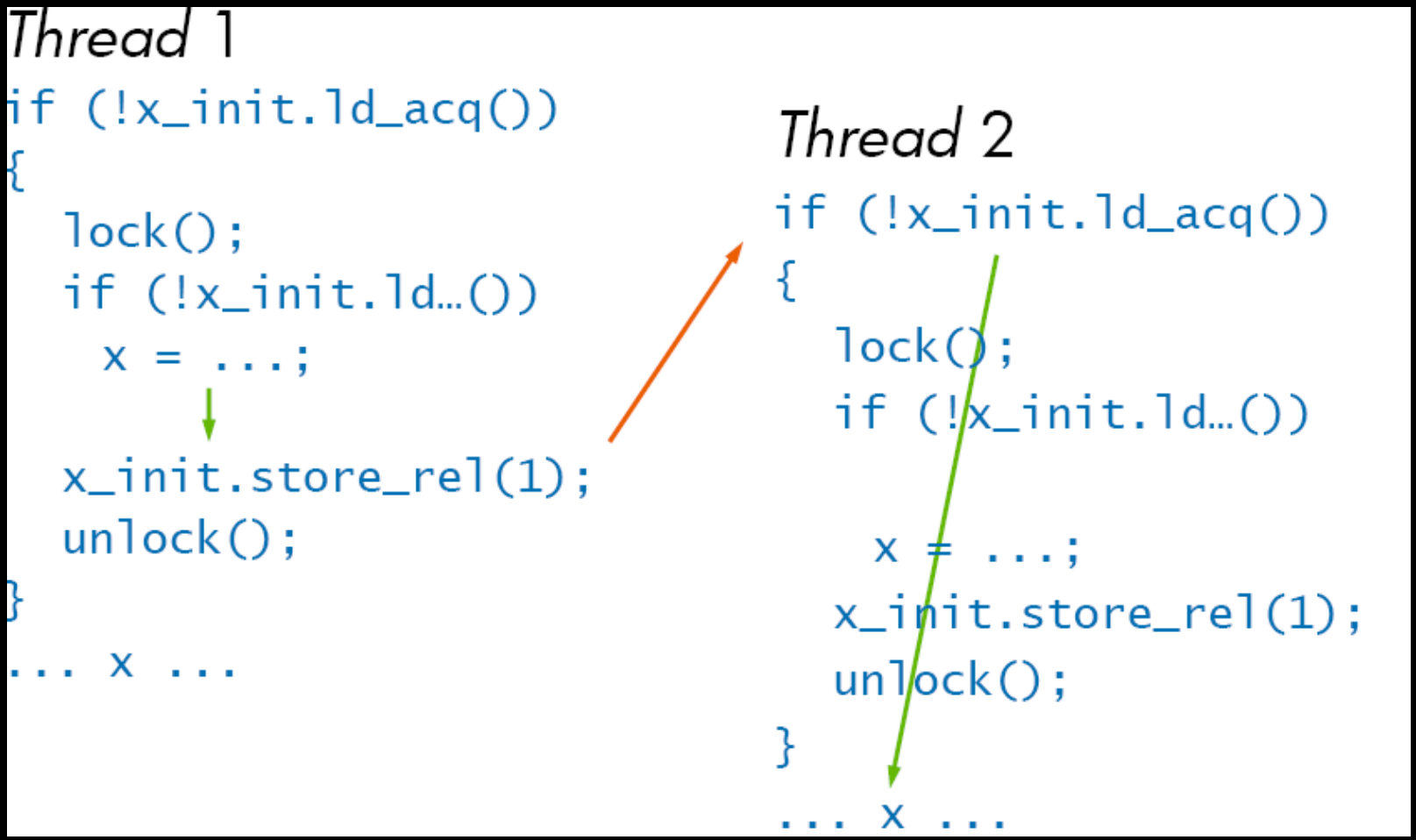
Cases

- **Function calls:**
  - The evaluations of the postfix expression and of the argument expressions are all unsequenced relative to one another. All side effects of argument expression evaluations are sequenced before the function is entered
- **Increment & Decrement:**
  - The value computation of the ++ expression is sequenced before the modification of the operand object.
- **Logical AND operator**
  - If the second expression is evaluated, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second expression.
- **Conditional Operator**
  - Every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second or third expression.
- **Comma operator**
  - Every value computation and side effect associated with the left expression is sequenced before every value computation and side effect associated with the right expression.

Happens before

- An evaluation A happens before an evaluation B if:
  - **A is sequenced before B, or**
  - **A synchronizes with B, or**
  - **for some evaluation X, A happens before X and X happens before B.**

Happens-before



Data race condition

- A non-atomic write to a memory location in one thread
- A non-atomic read from or write to that same location in another thread
- With no happens-before relations between them
- Is undefined behaviour

What is a memory location

- **A non-bitfield primitive data object**
- **A sequence of adjacent bitfields**
  - Not separated by a structure boundary
  - Not interrupted by the null bitfield
  - Avoid expensive atomic read-modify-write operations on bitfields

Effect on compiler optimization

- **Some rare optimizations are restricted**
  - Fewer speculative writes
  - Fewer speculative reads
- **Some common optimizations can be augmented**
  - They may assume that loops terminate
  - Nearly always true

Atomics: To Volatile or Not Volatile

- **Too much history in volatile to change its meaning**
- **It is not used to indicate atomicity like Java**
- **Volatile atomic means something from the environment may also change this in addition to another thread**

Requirements on atomics

- **Static initialization**
- **Reasonable implementation on current hardware**
- **Relative novices can write working code**
- **Experts can performance efficient code**

Consistency problem

- **X and y are atomic and initially 0**
  - Thread 1: x=1;
  - Thread 2: y=1;
  - Thread 3: if (x==1 && y==0)
  - Thread 4: if (x==0 && y==1)
- **Are both conditions exclusive?**
  - Is there a total store order?
- **The hardware/software system may not provide it**
- **Programming is harder without it**

Consistency models

- **Sequentially consistent**
  - What is observed is consistent with a sequential ordering of all events in the system
    - But comes with a very heavy cost
- **Weaker models**
  - More complex to code for some
    - But very efficient
- **What we decided**
  - Default is sequential consistency
  - But allow weaker semantics explicitly

Atomic Library (N2427)

- The problem:
  - **Would like to implement, for example, counters, without locks using atomics**

```
atomic<int> x;
void increment() {
    ++x;

}
```

```
int x;
mutex m;
void increment(){
    lock_guard_(m);
    x=x+1;
}
```

- **Advantages**
  - Sometimes enables much better performance
  - No space for locks, sometimes simpler.
  - Potentially safe for use with signal handlers, across processes.

Atomic DCL

```
T x;
atomic_bool x_init(false);
mutex m;
if (!x_init) {
    lock_guard _(m);
    if (!x_init) {
        x = ….
        x_init = true;
    }
}
use x;
```

- Note: Atomics are still tricky. Only a single memory operation at a time is atomic!

Atomic Design

- Want shared variables
  - **that can be concurrently updated without introducing data race,**
  - **that are atomically updated and read**
    - **half updated states are not visible,**
- that are implemented without lock overhead whenever the hardware allows,
- that provide access to hardware atomic read-modify write (fetch-and-add, xchg, cmpxchg) instructions whenever possible.

Minimal atomics

- **Need 1 primitive data types that is a must, m**ost modern hardware has instructions to implement the atomic operations
  - **for small types**
  - **and bit-wise comparison, assignment (which we require)**
  - Atomic flag type

    ```
    static atomic_flag v1= ATOMIC_FLAG_INIT
    If (atomic_flag_test_set(&v1))
        atomic_flag_clear(&v1);
    ```
- **For other types, hardware, atomic operations must be emulated with locks.**
  - Sometimes this isn.t good enough:
    - across processes, in signal/interrupt handlers.
  - is_lock_free() returns false if locks are used, and operations may block.
- **Operations on variable have attributes, which can be explicit**
  - Acquire=get other memory writes
  - Release=give my memory writes
  - Acq_and_rel=Acquire and release at the same time
  - Relaxed=no acquire or released, non-deterministic, not synchronizing with the rest of memory, but still sequential view of that variable
  - Seq-cst=Fully ordered,extra ordering semantics beyond acquire and releases, this is sequentially consistent

Other Atomic facilities

- Specializations for integral types, pointers
  - **Provide atomic increment, decrement (++, --, +=, -=)**
  - **Note: x++ is very different from x = x + 1 !**
    - **Unlike Java volatiles, where both are probably wrong!**
- Non-template (C-like) atomic types
  - **Template specializations inherit from these**
- C-like stand-alone (atomic_) function interfaces.

Race Free semantics and Atomic Memory operations

- **If a program has a race, it has undefined behavior**
  - This is sometimes known as “catch fire” semantics
  - No compiler transformation is allowed to introduce a race
    - no invented writes
    - Possibly fewer speculative stores **and (potentially) loads**
- **There are atomic memory operations that don’t cause races**
  - Can be used to implement locks/mutexes
  - Also useful for lock-free algorithms
- **Atomic memory operations are expressed as library function calls**
  - Reduces need for new language syntax

Memory Ordering Operations

```
enum memory_order {
  memory_order_relaxed,    // just atomic, no
    constraint
  memory_order_release,
  memory_order_acquire,
  memory_order_acq_rel,    // both acquire and release
  memory_order_seq_cst };  // sequentially consistent
                           // (even stronger than
    acq_rel)
```

- **Every atomic operation has a default form, implicitly using `seq_cst`, and a form with an explicit order argument**
- **When specified, argument is expected to be just an enum constant**

Basic atomics

- **Atomic<bool>**
  - Load, store, swap, cas
- **Atomic<int>**
  - Load, store, swap, cas
  - Fetch-and-(add, sub, and, or, xor)
- **Atomic<void *>**
  - Load, store, swap, cas
  - Fetch-and-(add, sub)

C-level operations

- There are new atomic data types that correspond to existing PODs, those have to be used for cross-thread communication
- Types are opaque, look like function calls
- Operations are type-generic function macros in C
- They look like overloaded functions in C++
- Operations include ordering constraints

```
static atomic_long a={1};
long t=atomic_load_explicit(&a, memory_order_acquire);
while (!atomic_compare_explicit(&a,t,t2,memory_order_relaxed))
        atomic_fetch_ior_explicit(&a,1,memory_order_acq_rel);
```

C synchronization example

```
int x = 0;

atomic_int y = 0;

Thread 1:
  x = 17;
  atomic_store_explicit(&y, 1,
  memory_order_release);
  // or: atomic_store(&y, 1);

Thread 2:
  while (atomic_load_explicit(&y,
  memory_order_acquire) != 1)
  // or: while (atomic_load(&y) != 1)
        continue;
  assert(x == 17);
```

C++ levels

- Atomic types are classes
- Small number of member functions and operators are fully ordered by default
- A defaulted parameter allows for different orderings

```
atomic<long> a={1};

long l=a; a=3;

while(!a.compare_swap(&a, &l, l+1));

++a; a+=4; a&=3;
```

Atomic assignment subtleties

- **Default assignment operator won't work properly**
  - Non-atomic loads and stores
- **Atomic assignment op is still bad**
  - Users would expect the whole assignment to be atomic
- **How to fix the problem?**
  - Since you can't fix in C90 or C++98, change it in C++0x
  - Modify POD definition
  - Offer a way to indicate defaulted and deleted functions

Atomic templates

- **Makes an atomic type from a non-atomic type**
  - Must be bitwise copyable and comparable
- **Defined specializations for basic types and pointers**
- **Suggested specializations for alignment and size**

```
atomic <int *> aip = {0};
aip=ip; aip+=4;
atomic <small_type> ag={ …. };
while (!ag.compare_swap(&ag, &g.g+4));
atomic<circus > ac; //works, but not recommended
```

Atomic freedom

- **Lock-free**
  - Robust to crashes
  - Someone will make progress
- **Wait-free**
  - Operations completed in a bounded time
  - Cas vs ll/sc
- **Address-free**
  - Atomicity does not depend on using the same address

Lock-free atomics

- **Large atomics have no hardware support**
  - Implemented with locks
- **Locks and signals don't mix**
  - Test for lock-free
- **Compile-time macros for basic types**
  - Always lock-free
  - Never lock-free
- **RTTI for each type**

Wait-free atomics

- **Hard to implement without direct HW support**
  - Resulting programs is usually HW-specific
  - Hard to be portable anyway
- **IBM argued against it since ll/sc is not wait-free**
- **Few who write this seemed to cared anyway**
  - No requirement for it
  - No query about it

Address-free atomics

- **Memory may not have a consistent address**
  - Processes sharing memory may not have the same address for that memory
  - Memory may be mapped into the address space twice
- **Atomic operations must be address-free to work**
  - One small tool for inter-process communication
- **A lock-free operation is address-free**
  - Not clear we can say this in a std way
  - But we will make our intent clear

C++0x Atomics

```
template< T > struct atomic {
    // Simplified, some details in flux .
    explicit atomic( T );
    atomic( const atomic& ) = delete;
    atomic& operator =( const atomic& ) = delete;
    void store( T ) volatile;
    T load( ) volatile;
    T operator =( T ) volatile; // similar to store()
    T operator T () volatile;
    // equivalent to load(), currently not in WP
    T swap( T ) volatile;
    bool compare_swap( T&, T) volatile;
    bool is_lock_free() const volatile;
};
```

Compare and swap

- compare_swap(expected, desired) atomically performs

```
if (load() == expected) {
    store(desired);
    return true;
} else {
    expected = load();
    return false;
}
```

- Allows atomic updates x = f(x) as

```
tmp = x;
do {
    new_val = f(tmp);
} while (!x.compare_swap(tmp, new_val));
```

Compiler Impact

- **Memory model does not say how to make an application thread safe**
  - Assumption is that source presented to compiler is thread safe
  - Undefined semantics for code with any data races
- **Memory model describes legal transformations on already safe code**
  - Compiler may not introduce any data races
- **Memory model concerned with performance**
  - Limited set of optimizations disallowed – may introduce data race
  - Allows some memory optimizations across locks
- **Quality implementation**
  - Most implementations already support a low quality implementation
  - Acquire/release operations seen as calls to opaque global functions – All shared variables may be referenced and modified

Agenda

- Concurrent C++0x examples
- C++ Memory Model
- **Concurrent C++ Core Language**
- Concurrent C++ Library
- Q/A

Non-atomic Variables

- **Thread Local Storage (TLS)**
- **Static duration variables**
  - Dynamic initialization
  - Destruction

Thread local storage variable

- **Adopt existing practice**

  __thread int a;

- **Introduce new storage duration**

  – Thread duration

  thread_local int var =3 ;

- Unique to each thread
- Accessible from every thread
- Address is not constant

Extend TLS

- **Existing practice only supports static initialization and trivial destructors**

```
std::string foo(std::string const& s2) {
  thread_local std::string s="hello";
  s+=s2;
  return s; }
```

- **Want to extend it to dynamic initializers and destructors**

  thread_local vector<int> var=f();

- Dynamic initialization allows lazy init
- OS support may be needed

Initialization of static-duration variables

- **Dynamic initialization is tricky**
  - No syntax to order most initializations
- **Without synchronization, potential data races**
- **With synchronization, potential deadlock**
- **Examine 2 kinds:**
  - Function local statics
  - globals

Function-local static storage

- **Initialization implicitly synchronized**
  - While not holding any locks
- **New algorithm contributed by Mike Burrows from Google**

```
void bar() {
  static my_class z(42+foo()); // initialization is thread-safe
  z.do_stuff();
}
```

Do not use a mutex during Initialization?

- Constructor declared as constexpr and satisfies the requirements for constant initialization
- Such objects are guaranteed to be init before any code is run as part of static init phase

```
class my_class {
    int i;
    public:
            constexpr my_class():i(0){}
            my_class(int i_):i(i_){}
            void do_stuff(); };
my_class x; // static initialization with constexpr constructor
int foo();
my_class y(42+foo()); // dynamic initialization
void f()
{ y.do_stuff(); // is y initialized? }
```

Global variable

- **Initialization implicitly synchronized**
- **Concurrent initialization enabled**
- **Initialization may not use a dynamically initialized object defined outside the translation unit**

```
extern vector<int> e;

vector <int> u; //OK, default init

vector <int> v(u); //OK within this TU

vector <int> w(e); //undefined, outside of this TU
```

If you have to dynamically initialize…

- **When std::call_once is used with an instance of std::once_flag, function is called exactly once**

```
my_class* p=0;
std::once_flag p_flag;
void create_instance() {
  p=new my_class(42+foo()); }
void baz() {
  std::call_once(p_flag,create_instance); p->do_stuff(); }
```

destruction

- First terminate all threads
- Execute destructors in a concurrent reverse of initialization
- Interleave namespace-scope vars with function-scope static vars
- Same restriction on use of vars outside current TU

Agenda

- Concurrent C++0x examples
- C++ Memory Model
- Concurrent C++ Core Language
- **Concurrent C++ Library**
- Q/A

Library Threads API (N2497)

- Based on Boost, with some improvements:
  - **Thread creation**
- **Basic thread class**
  - Fork and join model
  - Standardize access to non-standard OS implementation
- **Mutexes and condition variables**
  - Easier to use than window events
  - Enables the monitor paradigm

Termination

- **Thread termination is voluntary**
  - Return from outermost function
  - Cooperative termination, synchronous, exception-based termination
- **General opposition to asynchronous termination or interrupts**
- **What to do with exception that leaves thread that is uncaught**

Fork join

- **Basic thread class**
  - Fork a function execution
  - Void join operation

```
void f();
void bar()
{
        std::thread t1(f); //f() executes in separate thread
        …
        t1.join(); //wait for thread t1 to end
}
```

Exception capture and copy

```
Std::exception_ptr why;
void capture() {
  try{
     throwing_func();
  } catch (…) {
      why=std::copy_exception(std::current_exception() );
  }
}
void liberate() {
  rethrow_exception(why);
}
```

scheduling

- **Limited thread scheduling**
  - Yield
  - Sleep
- **Standard access to non-standard underlying OS thread handles**
  - For detailed control
- **Query for hardware concurrency**

mutexes

- non-recursive (std::mutex)
- recursive (std::recursive_mutex)
- non-recursive that allows timeouts on the lock functions (std::timed_mutex)
- recursive mutex that allows timeouts on the lock functions (std::recursive_timed_mutex)

Mutual Exclusion

- Real multi-threaded programs usually need to access shared data from multiple threads.
- For example, incrementing a counter in multiple threads:

  x = x + 1;
- Unsafe if run from multiple threads:

  tmp = x; // 17

  tmp = x; // 17

  x = tmp + 1; // 18

  x = tmp + 1; // 18
- Standard solution:
  - **Limit shared variable access to one thread at a time,using locks.**
  - **Only one thread can be holding lock at a time.**

Mutexes

```
class mutex {
    public: mutex();
    ~mutex();
    mutex(const mutex&) = delete;
    mutex& operator=(const mutex&) = delete;
    void lock();
    bool try_lock();
    void unlock();
    ...
};
```

- Class recursive_mutex is similar:
  - **allows *same* thread to acquire mutex mutiple times.**

Counter with a mutex

```
mutex m;
void increment() {
  m.lock();
  x = x + 1;
  m.unlock();
}
```

- Lock not released if critical section throws.

locks

- Hold a mutex within a given scope
- Represents the mutex acquire/release pair
- Release occurs in the destructor for the lock

```
std::upgradable_mutex mut;
void foo() {
        std::upgradable_lock<std::upgradable_mutex> read_lock(mut);
        //do read operation
        if (/*sometimes need to write */) {
                std::exclusive_lock<std::upgradable_mutex>
write_lock(std::move(read_lock));
                //do write operation, what was read hasn't changed
        }
}
```

Condition variables

- **Threads may wait on a condition variable**
  - Giving up their hold on the mutex
- **Threads may notify a condition variable**
  - To reacquire the mutex and reevaluate the condition
- **Benefits**
  - Easier then events
  - Enables monitor pattern

Waiting for events

```
std::mutex m;
std::condition_variable cond;
bool data_ready;
void process_data();
void foo() {
std::unique_lock<std::mutex> lk(m);
  while(!data_ready) { cond.wait(lk); }
  process_data(); } .
```

Setting the Data Ready

```
void set_data_ready() {
  std::lock_guard<std::mutex> lk(m);
  data_ready=true;

  cond.notify_one(); }

void foo() {

  std::unique_lock<std::mutex> lk(m);
  cond.wait(lk,[]{return data_ready;});
  process_data(); }
```

Agenda

- Concurrent C++0x examples
- C++ Memory Model
- Concurrent C++ Core Language
- Concurrent C++ Library
- **Q/A**

Food for thought and Q/A

- **This is the chance to make comments on the C++0x draft through us or the National Body rep:**
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2857.pdf
- Memory Model:
  - **http://www.hpl.hp.com/personal/Hans_Boehm/c++mm**
- **Participate and feedback to Compiler**
- **Talk to me at my blog:**
  - http://www.ibm.com/software/rational/cafe/blogs/cpp-standard

My blogs and email address

- michaelw@ca.ibm.com
- Rational C/C++ cafe: http://www.ibm.com/software/rational/cafe/community/ccpp
- My Blogs:
- Parallel & Multi-Core Computing http://www.ibm.com/software/rational/cafe/blogs/ccpp-parallel-multicore
- C++ Language & Standard http://www.ibm.com/software/rational/cafe/blogs/cpp-standard
- Commercial Computing http://www.ibm.com/software/rational/cafe/blogs/ccpp-commercial
- Boost test results http://www.ibm.com/support/docview.wss?rs=2239&context=SSJT9L&uid=swg27006911
- C/C++ Compilers Support Page http://www.ibm.com/software/awdtools/ccompilers/support/
- C/C++ Feature Request Interface http://www.ibm.com/support/docview.wss?uid=swg27005811
- XL Fortran Compiler Support Page http://www.ibm.com/software/awdtools/fortran/xlfortran/support/
- XL Fortran Feature Request Interface http://www.ibm.com/support/docview.wss?uid=swg27005812

Acknowledgement

- **Some slides are borrowed from committee presentations by various committee members, their proposals, and private communication**