# CSE 160
# Lecture 7

# C++11 Memory Model

Scott B. Baden

# Today's lecture

- C++11 Memory model
- Implementing synchronzation

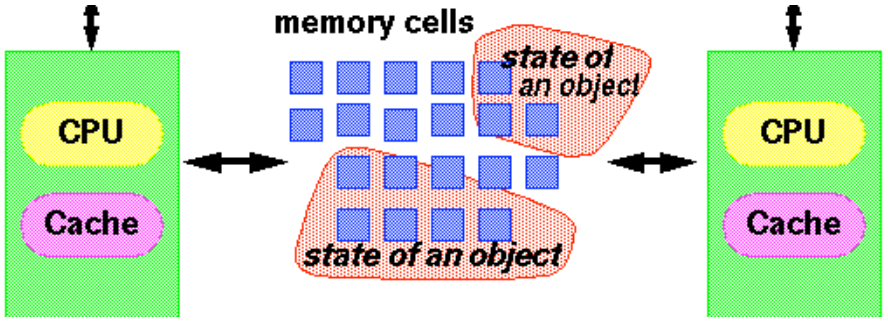
# Announcements

# The C++11 Memory model

- The C++11 memory model makes minimal guarantees about semantics of memory access
- Bounds the potential effects of optimizations on execution semantics and discusses techniques for programmers to control some aspects of semantics enabling use to ensure code correctness
  - Compiler optimizations that move code
  - Hardware scheduler that executes instructions out of order
- Guarantees made by the memory model are weaker than most programmers intuitively expect, and are also weaker than those typically provided on any given C++ implementation

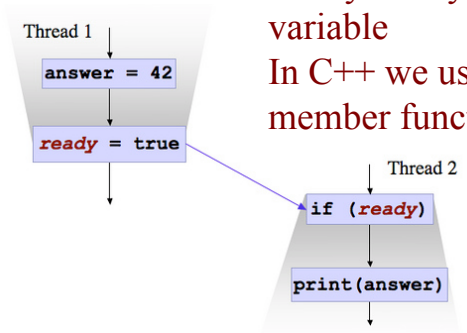
# Preliminaries

- Each thread runs on its own CPU
- This may or may not be the case in a specific run of a program
- The C++ memory model describes an abstract relation between threads and memory
- The model makes guarantees about properties concerning interaction between instruction sequences and variables in memory



# Communication

- Special mechanisms are needed to guarantee that communication happens between threads
- Memory writes made by one thread can become visible, but no guarantee.
- *Without explicit communication, you can't guarantee which writes get seen by other threads, or even the order in which they get seen*
- The C++ atomic variable (and the Java volatile modifier) constitutes a special mechanism to guarantee that communication happens between threads
- When one thread writes to a *synchronization variable*, and another thread sees that write, the first thread is telling the second about all of the contents of memory up until it performed the write to that variable




Ready is a synchronization variable
In C++ we use load and store member functions

*All the memory contents seen by T1, before it wrote to ready, must be visible to T2,*

*after it reads the value true for ready*.

http://jeremymanson.blogspot.com/2008/11/what-volatile-means-in-java.html

# Rules

- **3 rules** that mostly concern when values must be transferred between main memory and per-thread memory
- **Atomicity.** Which instructions must have indivisible effects? Only concerned with instance and static variables, including array elements, but **not** local variables inside methods
- **Visibility.** Under what conditions the effects of one thread are visible to another? The effects of interest are: writes to variables, as seen via reads of those variables
- **Ordering.** Under what conditions the effects of operations can appear out of order to any given thread? In particular, reads and writes associated with sequences of assignment statements.

# Sequentially consistent execution

- In this program, if x=y=0 initially, it is not possible for r1=r2=0 at the end

| Thread 1 | Thread 2 |
|---|---|
| x = 1;<br>r1 = y; | y = 1;<br>r2 = x; |

- Multithreaded execution could result in many possible *sequentially consistent* interleavings

| Execution 1 | Execution 2 | Execution 3 |
|---|---|---|
| x = 1;<br>r1 = y;<br>y = 1;<br>r2 = x;<br>**// r1 == 0 ∧ r2 == 1** | y = 1;<br>r2 = x;<br>x = 1;<br>r1 = y;<br>**// r1 = 1 ∧ r2 = 0** | x = 1;<br>y = 1;<br>r1 = y;<br>r2 = x;<br>**// r1 = 1 ∧r2 ==1** |

# Sequentially consistency in practice

- Too expensive to guarantee sequentially consistency all the time
  - Code transformations made by the compiler
  - Instruction reordering in modern processors
  - Write buffers in processors
- In short, different threads perceive that memory references are reordered

# Data races

- We say that a program allows a *data race* on a particular set of inputs if there is a *sequentially consistent execution*, i.e. an interleaving of operations of the individual threads, in which two conflicting operations can be executed "simultaneously" (Boehm)
- We'll say that operations can be executed "simultaneously", if they occur next to each other in the interleaving, and correspond to different threads
- We can guarantee sequential consistency only when the program avoids data races
- This program has a data race

| Execution 3 |
| --- |
| **x = 1;**<br>**y = 1;**<br>r1 = y;<br>r2 = x;<br>**// r1 = 1 ∧ r2 ==1** |

| Thread 1 | Thread 2 |
| --- | --- |
| x = 1;<br>r1 = y; | y = 1;<br>r2 = x; |

# "Happens-before"

- An important fundamental concept in understanding the memory model
- Run on two separate threads, with counter = 0
  A: counter++;
  B: prints out counter
- Even if B occurs after A, no guarantee that B will see 0
  …
- … unless we establish *happens-before relationship* between these two statements
- What guarantee is made by a *happens-before* relationship ?
  *A guarantee that memory writes by one specific statement are visible to another specific statement*

- Different ways of accomplishing this: synchronization, atomics, variables, thread creation and completion

# Avoiding Data races

- C++ and Java provide *synchronization* variables to communicate between threads, and are intended to be accessed concurrently
- Such concurrent accesses are not considered data races
- Thus, sequential consistency is guaranteed so long as the only conflicting concurrent accesses are to synchronization variables
- Any write to a synchronization variable establishes a *happens-before* relationship with subsequent reads of that same variable
- In C++ we can guarantee that our initial example program behaves as expected if we make both x and y synchronization variables.
  - In C++ we could use *atomic* types
  - Java: declare them *volatile*
- Declaring a variable as a synchronization variable
  - ensures that the variable is accessed indivisibly
  - prevents both the compiler and the hardware from reordering memory accesses in ways that are visible to the program.

# Using synchronization variables to ensure sequentially consistent execution

- Consider this example where x is of type int, x_ready is of type atomic<bool>,
- This program is free from data races
- Thread 2 is guaranteed not to progress to the second statement until the first thread has completed and set x_ready  There cannot be an interleaving of the steps in which the actions x = 42 and r1 = x are adjacent.
- Thus, we are guaranteed a sequentially consistent execution, which guaranteeing  that r1 = 42.
- Thus implementation must ensure arrange that
  - thread 1's assignments to x and x_init become visible to other threads in order
  - The assignment r1 = x operation in thread 2 cannot start until we have seen x_init set.
- In practice  these require the compiler to obey extra constraints and to generate special code to prevent potential hardware optimizations, such as thread 1 making the new value of x_init available before that of x because it happened to be faster to access x_init's memory

| Thread 1 | Thread 2 |
|---|---|
| x = 42;<br>x_ready = true; | while (!x_ready) {}<br>r1 = x; |

# Visibility

- Changes to fields made by one thread are guaranteed to be visible to other threads only under the following conditions
- A writing thread releases a synchronization lock and a reading thread subsequently acquires that same
  - Releasing a lock flushes all writes from the thread's working memory, acquiring a lock forces a (re)load of the values of accessible variables
  - While lock actions provide exclusion only for the operations performed within a synchronized block, these memory effects are defined to cover all variables used by the thread performing the action
- If a variable is declared as atomic
  - Any value written to it is flushed and made visible by the writer thread before the writer thread performs any further memory operation.
  - Readers must reload the values of volatile fields upon each access.
- As a thread terminates, all written variables are flushed to main memory. Thus, if one thread synchronizes on the termination of another thread using join, then it is guaranteed to see the effects made by that thread

# “Eventually can be a long time”

- The memory model guarantees that a particular update to a particular variable made by one thread will **eventually** be visible to another. But eventually can be an arbitrarily long time
  - Long stretches of code in threads that use no synchronization can be hopelessly out of synch with other threads with respect to values of fields
  - It is always wrong to write loops waiting for values written by other threads unless the fields are atomic or accessed via synchronization
- Rules do not require visibility failures across threads, they merely allow these failures to occur
- Not using synchronization in multithreaded code doesn't guarantee safety violations, it just allows them
- Detectable visibility failures might not arise in practice
- Testing for freedom from visibility-based errors impractical, since such errors might occur extremely rarely, or only on platforms you do not have access to, or only on those that have not even been built yet

# Memory fences

- How are we assured that a value updated within a critical section becomes visible to all other threads?
- With a *fence* instruction, e.g. MFENCE
- "A serializing operation guaranteeing that every load and store instruction that precedes, *in program order,* the MFENCE instruction is globally visible before any load or store instruction that follows the MFENCE instruction is globally visible." [Intel 64 & IA32 architectures software developer manual]
- Also see www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

```
mutex mtx;
…
mutex.mtx.lock();
sum += local sum;
mutex.mtx.unlock();
```

# C++ Atomic types

- In terms of atomicity, visibility, and ordering, declaring a field as atomic is nearly identical in effect to accessing it within a critical section
  #include <atomic.h>
  atomic<int> x;
- Declaring a variable as *atomic* differs only in that no locking is involved
  - Includes operations such as +=, ++
  - But ordinary assignment is not performed atomically, nor read access that aren't made via operators such as ++
  - To this end, atomic types have load( ) and store( ) operations
  - Ordering and visibility effects surround only the single access or update to the atomic variable itself
- Simple atomic variable access is more efficient than accessing variables through synchronized code, but requires care to avoid memory consistency errors

# Implementing Synchronization Primitives

- We build mutex and other synchronization primitives with special atomic operations, implemented with a single machine instruction, e.g. CMPXCHG
- Do atomically: compare contents of memory location loc to expected; if they are the same, modify the location with newval

```
CAS (*loc ,  expected ,  newval  ) {
    if (*loc ==  expected ) {
        *loc =  newval;
         return 0;
    }
    else
      return 1
```

- We can then build mutexes with CAS

```
Lock( *mutex ) {
   while (CAS ( *mutex , 1, 0)) ;
}

Unlock( *mutex ) { *mutex =  1; }
```

# Building an atomic counter

- We'll implement an atomic integer counter that exports two operations, plus a constructor
  - getValue()
  - incr()

```
class AtomicCounter{
 private:
    std::atomic<int> _ctr;
 public:
    int getCtr() { return _ctr; }
   AtomicCounter(){ _ctr = 0; }
   int incr() {
       int oldCtr = getCtr();
       while (CAS(&_ctr, &oldCtr, oldCtr + 1) )
           oldCtr = getCtr();
       return oldCtr + 1;
   }
```

value ← updated
if current = **expected**