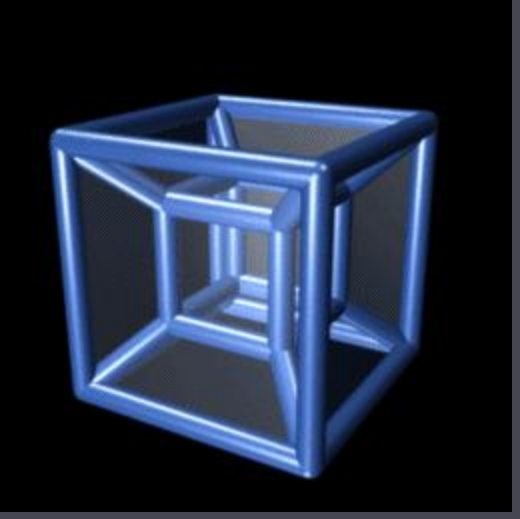
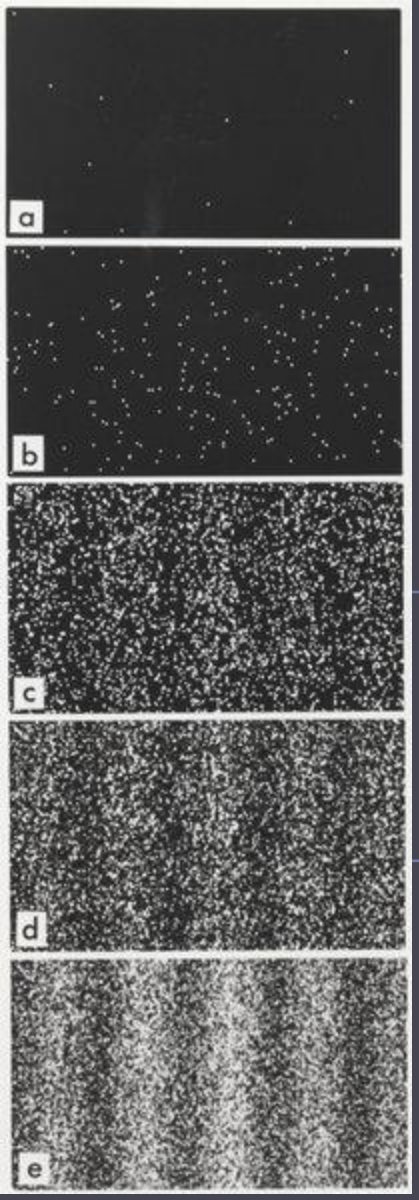


C++ Memory Model

Don’t believe everything you read (from shared memory)

The Plan

- Why multithreading is hard
- Warm-up example
- Sequential Consistency
- Races and fences
- The happens-before relation
- The DRF guarantee
- C++ memory model and atomics



Multithreading

From classical mechanics to quantum mechanics

Single-Threaded Programs

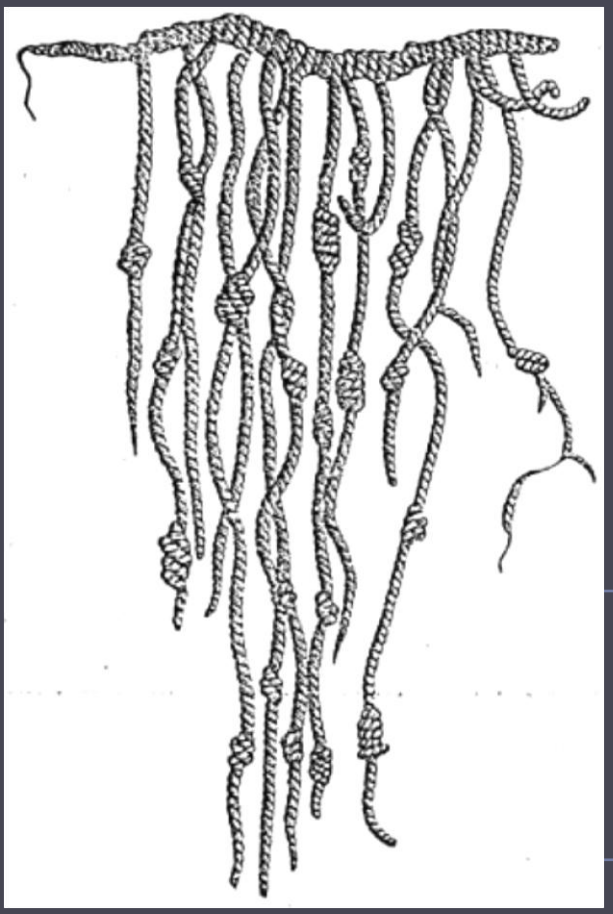
- Execute one statement after another
- One memory access after another
- A read after a write returns the last-written value
- Execution is deterministic (like classical mechanics)
  - Given the same inputs, the program will generate the same outputs—every time!
- Reasoning about programs relatively easy
- Testing and debugging relatively easy
  - Execute using a representative subset of possible inputs
  - Bugs easily reproducible

Multi-threaded Programs

- Statements executed by threads are interleaved
- Exponential number of possible interleavings
- Memory accesses interleaved
- A read may return a value written by another thread
- Execution nondeterministic (like quantum mechanics)
  - Same inputs but different interleavings may produce different output
- Reasoning about programs (even short ones) difficult
- Testing and debugging extremely difficult

Memory Model

- Makes reasoning about multithreaded programs possible (albeit not easy!)
- Tells the compiler what optimizations are not allowed
- Makes reasoning independent of the processor
- Gives multi-processor designers a set of requirements
- Java has one, C/C++ doesn't
- Current C++
  - If multithreaded programs work at all, they are not portable between compilers/processors
  - Vendor specific implementations (like volatile)



A “Simple” Example

The Infamous Double-Checked Locking Pattern

Single-threaded Singleton

```
class SingletonFoo
{
    private Foo _foo = null;
    public Foo getFoo ()
    {
        if (_foo == null)
            _foo = new Foo;
        return _foo;
    }
}
```

- new Foo is executed only once
- Returned _foo is fully constructed

Multi-threaded Singleton

```
class SingletonFoo
{
    private Foo _foo = null;
    public synchronized Foo getFoo ()
    {
        if (_foo == null)
            _foo = new Foo;
        return _foo;
    }
}
```

- Brute force approach: eliminate interleaving
- Only one thread at a time executes synchronized section
- About *two orders of magnitude* slower than single-threaded version

Clever (but wrong) Optimization

```
class SingletonFoo {
    private Foo _foo = null;
    public Foo getFoo () {
        if (_foo == null) {
            synchronized (this) {
                if (_foo == null)
                    _foo = new Foo;
            }
        }
        return _foo;
    }
}
```

Clever (but wrong) Optimization

```
class SingletonFoo {
    private Foo _foo = null;
    public Foo getFoo () {
        if (_foo == null) {
            synchronized (this) {
                if (_foo == null)
                    _foo = new Foo;
            }
        }
        return _foo;
    }
}
```

- Subtle problem: relaxed memory model
- _foo appears non-null before construction finished
- Another thread picks up uninitialized object

Volatile Solution

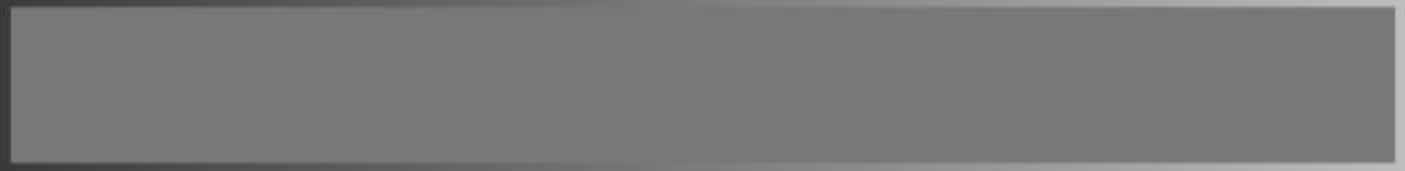
```
class SingletonFoo {
    private volatile Foo _foo = null;
    public Foo getFoo () {
        if (_foo == null) {
            synchronized (this) {
                if (_foo == null)
                    _foo = new Foo;
            }
        }
        return _foo;
    }
}
```

- Any modification to a *volatile* variable (conceptually) forces *all* previous writes to memory
- The snag: we might have just killed our optimization

Multi-threading in C++

- volatile keyword has no multi-threaded meaning
  - Some compilers cheat
  - Might work on some processors
- C++ 0x will have a memory model
  - *Race-free* programs appear *sequentially consistent*
  - Java *volatile* implemented as *atomic* library

Sequential Consistency



The Grand Illusion

Uniprocessor

- Instructions are executed in program order
- Every read returns the value last written to that location
- However
  - Compiler and processor are free to rearrange instructions
  - Optimizing compilers move or eliminate code
  - Reads and writes are buffered on a processor (memory access costs hundreds of cycles)
- Optimizations must preserve the illusion

Multiprocessor Sequential Consistency

- Each thread/processor executes instructions in program order
- Instructions performed by different processors are arbitrarily interleaved
- Each instruction is atomic and instantaneous wrt. other processors
- Conceptual switch connects one processor at a time to memory
- This is not how the program is executed in reality!
- *Compiler* must enforce the illusion (even though modern processors don't) for a reasonable subset of programs.

Breakdown of Sequential Consistency

```
Processor 1    Processor 2
x = 43          r1 = ready
ready = 1       if r1 == 1
                  r2 = x
```

- Initially x = 0, r1 = 0, r2 = 0, ready = 0
- r1 and r2 thread local (registers)
- Can r1 == 1 and r2 == 0?

Breakdown of Sequential Consistency

```
Processor 1    Processor 2
x = 43          r1 = ready
ready = 1       if r1 == 1
                  r2 = x
```

- Initially x = 0, r2 = 0, ready = 0
- r1 and r2 thread local (registers)
- Can r1 == 1 and r2 == 0?
- Yes, if processor/compiler reorders writes to *x* and *ready*
- *Compiler and processor optimizations are the enemies of sequential consistency*

Multithreaded Olympics

Races and fencing

Low-level Data Race

- Data conflict
  - Two processors access the same memory location
  - At least one access is a write
- Data race
  - There is no intervening synchronization

- Data races on both *x* and *ready*

```
Processor 1    Processor 2
x = 43          r1 = ready
ready = 1       if r1 == 1
                  r2 = x
```

Bad Bad Programmer!

- Data races are bugs!
- Data races break sequential consistency
- Language doesn't have to specify the behavior of buggy programs

- Warnng: presence of data races be used to maliciously attack a program

How to Beat Relaxed Memory Model into Submission

```
Processor 1     Processor 2
x = 43          r1 = ready
ready = 1       if r1 == 1
                  r2 = x
```

- Each thread may see results of writes in different order
- If the order is important, the *programmer* must force it
- Memory fence (or barrier)—commit pending writes

```
Processor 1     Processor 2
x = 43          fence
fence           r1 = ready
ready = 1       if r1 == 1
fence             fence
                  r2 = x
```

Happens Before

In the absence of simultaneity

Interleavings

```
Processor 1    Processor 2
x = 43
fence
ready = 1
fence          fence
               r1 = ready // 1
               if r1 == 1 // true
                   fence
                   r2 = x  // 43
```

- Without synchronization, no guarantee the reads will see last-written values
- If ready and x volatile (the fences), the guarantee holds
- With synchronization, the write "x = 43" ***happens before*** the read "r2 = x" in the second execution

Happens Before Relation

- On the same processor, A happens before B if A is earlier in the program order
- Between processors, *happens-before* defined by *synchronizing actions*
  - Write to a volatile variable happens before all subsequent reads of that variable—***in a particular execution***
  - Unlocking of a monitor happens before the subsequent locking of the same monitor
- Happens-before is a transitive closure of program order and synchronization order

DRF Guarantee

- A conflict is a ***race*** if there is at least one execution in which one action doesn't ***happen before*** another
- We want the language to guarantee the following

- ***Correctly synchronized programs (Data Race Free) have sequentially consistent semantics***

The C++ Memory Model

Do not write incorrect programs!

C++ Memory Model

- If a program is data race free, its execution is sequentially consistent.
- Otherwise it's undefined behavior

- Atomic library provides equivalents of Java "volatile"

Atomic

```
template <class T>
class atomic {
public:
    atomic(T); // No ordering semantics, constructor not atomic.
    void store(const T&);
    T load();
    bool cas(const T& old, const T& new_val);

    T operator T() { return load<acquire>(); }
    void operator=(const T& x) { return store<release>(x);
}
```

- In full version, load and store are templates
- CAS—atomic Compare And Swap

Atomic int and ptr

```
template <class T=int>
class atomic_int : public atomic<T> {
public:
    atomic_int(T);
    T fetch_add(T);
    T fetch_and(T);
    T fetch_or(T);
};

template <class T>
class atomic_ptr : public atomic<T> {
public:
    atomic_ptr(T);
    T fetch_and_add(ptr_diff_t);
};
```

Low-level Atomics

- Escape from sequential consistency
- Programmer may specify ordering constraints
  - *raw* (or *memory_model_relaxed)*: no constraints
- Sane programmers should never use those without drawing a bull's eye on their feet

```
enum ordering_constraint {raw, acquire, release, ordered};

template <ordering_constraint c>
   void store(const T&);
   // Compile-time error if c is neither raw nor release.
```

What Have We Learned?

- We want programs to behave in a sequentially-consistent way so that we can reason about them
- For reasons of performance, we'll settle for well-synchronized programs behaving in a sequentially consistent way
- Programs with data races are *not* well synchronized
- To understand data races we define synchronize-with and happens-before relations

Bibliography

- Scott Meyers and Andrei Alexandrescu. C++ and The Perils of Double-Checked Locking. Doctor Dobb's Journal, Jul 2004.
- Java Memory Model, http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf
- C++ MM, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2429.htm (this is a revision of a revision of a revision…)
- Less formal explanation, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2480.html
- FAQ on C++ MM, http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/