

**Implications of Memory Models  
(or Lack of Them)  
for  
Software Developers**

J. L. Sloan  
jsloan@diag.com

# A Typical Day in Product Development

## My Introduction to Modern Memory Models

- Parent thread calls `new` to create a C++ object.
- Pristine C++ object contains context of Child thread.
- Parent thread calls `pthread_create` to create Child.
- Child thread examines the C++ object...
- ... and crashes because the virtual table is uninitialized.

*This should be impossible!*

- But apparently it isn't...
- ... on a hyperthreaded P4 PC running RedHat Linux.

“Not only is the universe stranger than we imagine, it is stranger than we can imagine.”

-- Sir Arthur Eddington

# The Evolution of a Classic Design Pattern

## GOF Lazy Singleton Pattern in C++

```
Singleton* Singleton::pInstance = 0;

Singleton* Singleton::instance() {
    if (pInstance == 0) {
        pInstance = new Singleton();
    }
    return pInstance;
}
```

*But this isn't thread safe.*

## Thread-Safe Lazy Singletons in C++

```
Singleton* Singleton::pInstance = 0;
Mutex Singleton::mutex;

Singleton* Singleton::instance() {
    mutex.acquire();
    if (pInstance == 0) {
        pInstance = new Singleton();
    }
    mutex.release();
    return pInstance;
}
```

*But this acquires the mutex every time the singleton is accessed.*

## The Double-Checked Locking Pattern in C++

```
Singleton* Singleton::pInstance = 0;
Mutex Singleton::mutex;

Singleton* Singleton::instance() {
    if (pInstance == 0) {
        mutex.acquire();
        if (pInstance == 0) {
            pInstance = new Singleton();
        }
        mutex.release();
    }
    return pInstance;
}
```

## Singletons in Java

```
Class SingletonHelper {  
    public static Singleton pInstance = new  
        Singleton();  
}
```

*But this isn't lazy.*

## Thread-Safe Lazy Singletons in Java

```
Class SingletonHelper {  
    private Singleton pInstance = null;  
    public synchronized Singleton instance() {  
        if (pInstance == null) {  
            pInstance = new Singleton();  
        }  
        return pInstance;  
    }  
}
```

*But this acquires the mutex every time the singleton is accessed.*

## The Double-Checked Locking Pattern in Java

```
Class SingletonHelper {
    private Singleton pInstance = null;
    public Singleton instance() {
        if (pInstance == null) {
            synchronized (this) {
                if (pInstance == null) {
                    pInstance = new Singleton();
                }
            }
        }
        return pInstance;
    }
}
```

## “The Double-Checked Locking Pattern Is Broken”

... on many modern processors!

- In Java 1.5 and later unless using `volatile` or `synchronized`.
- In Java 1.4 and earlier no matter what.
- In vanilla C and C++ regardless.
- Other similar patterns that depend on consistent memory behavior among threads may not work either.
- Lots of very smart people have demonstrated that there are no simple fixes, because the causes are mainly in the hardware of modern microprocessors. [Meyers]

## Example Failure Modes of (Just) the DCLP

- Duplicated singletons
- Use of object before virtual table initialized

**There Ain't No Such Thing As a Free Lunch**

## Moore's Law Comes With A Price

In order to maintain Moore's Law (and revenue), modern microprocessor designers and compiler developers have resorted to increasingly complex

- compiler optimizations,
- processor optimizations,
- memory optimizations, and
- concurrency

with the result being that things are not what they seem.

## Moore's Law Comes With A Price

- This is okay if you are
  - on a single processor core
  - in a single thread of execution
  - accessing real memory.
- Increasingly this is not the case.
  - Multi-core processors are becoming the norm.
  - Threaded design necessary for performance.
  - More and more hardware is memory-mapped.

## Compiler and Processor Optimizations

- Register Use
  - particularly in RISC architectures
  - “rematerialization” by reloading from globals
  - redundant writes to globals
- Speculative Execution
- Pre-fetching
- *Machine Instruction Reordering*
  - by the compiler at compile time
  - by the processor at run time
  - maintaining “observable behavior” of a single thread

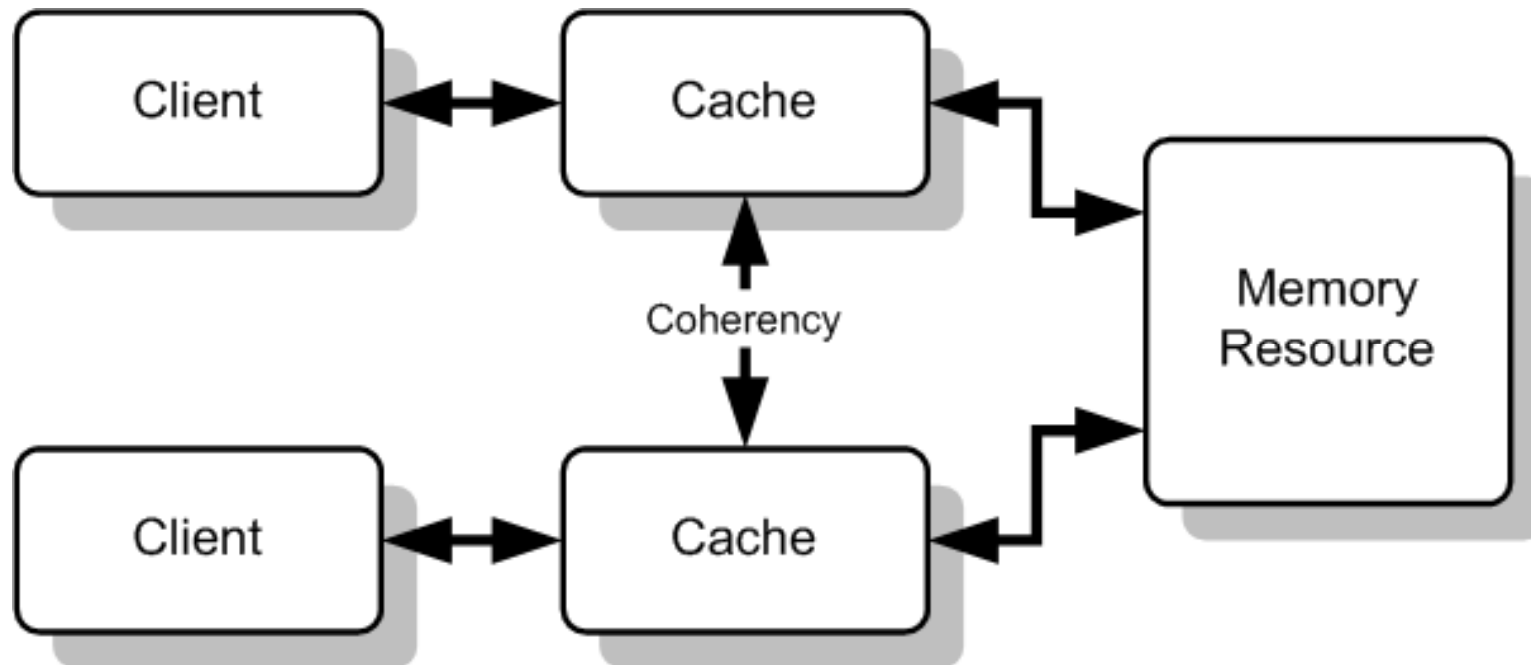
## Compiler and Processor Optimizations

- Actions may not occur in the order that you think they do.
- Looking at the assembler code is not sufficient.
  - Assembler only reveals compiler reordering.
- This can be seen with JTAG-based debuggers.
  - Program counter “jumps” around.

## Memory Optimizations

- Word Tearing
  - e.g. CRAY Y-MP
- Non-atomic Writes
  - 64-bit words on many systems
- Non-atomic Machine Instructions
- *Memory Caches*
  - long cache lines
  - multiple (levels of) caches
  - cache coherency (a.k.a. cache consistency) protocols
    - “A write must *eventually* be made visible to all processors.”
    - “Writes to the *same* location must appear to be seen in the same order by all processors.” [Adve] (Italics mine.)

# Cache Coherency



Public Domain 2006 Dennis Schmitz

## Cache Coherency

MESI Protocol (for example, there are others):

- **M**odified                   dirty but only in this cache
- **E**xclusive                 clean but only in this cache
- **S**hared                    clean and may be in other caches
- **I**nvalid                   no longer represents current value

(All cache lines start out as Invalid)

## Cache Coherency

“If multiple threads modify the same memory location concurrently, processors do not guarantee any specific result. This is a deliberate decision made to avoid costs which are unnecessary in 99.999% of all cases. For instance, if a memory location is in the ‘S’ state and two threads concurrently have to increment its value, the execution pipeline does not have to wait for the cache line to be available in the ‘E’ state before reading the old value from the cache to perform the addition. Instead it reads the value currently in the cache and, once the cache line is available in state ‘E’, the new value is written back. The result is not as expected if the two cache reads in the two threads happen simultaneously; one addition will be lost.”

[Drepper], 6.4.2, “Atomicity Operations”, p. 68

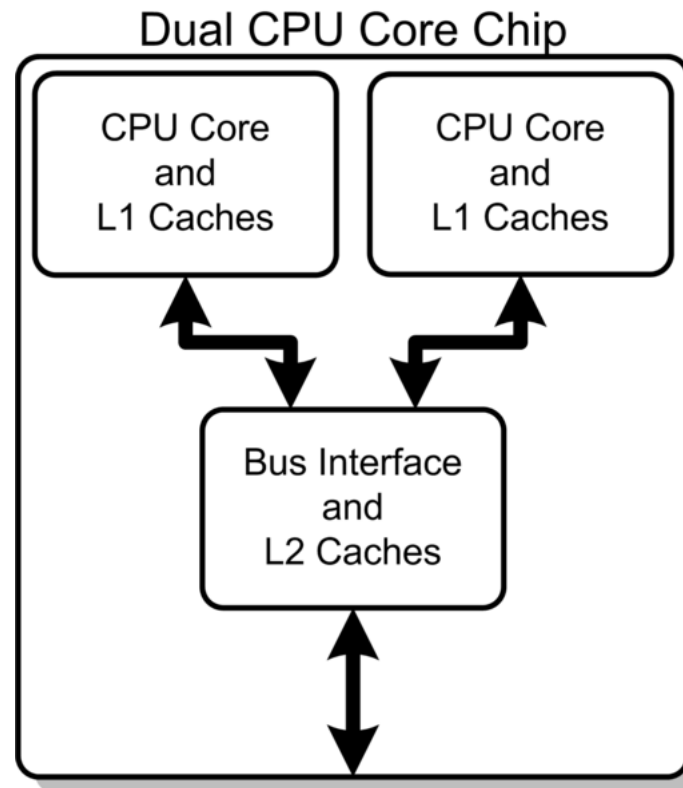
## Memory Optimizations

- Memory stores may not be happening in the order that you think they are.
- Memory may be written when you don't expect it.
- Memory may not be written when you do expect it.
- Different threads may have very different views of the state of a variable, or
- of a collection of related variables.

## Concurrency

- Multiple Execution Units within a Single Processor
  - ALU
  - FPU
- Multiple Concurrent Instruction Pipelines
- *Multiple Processor Chips*
  - SMPs
  - special purpose processors e.g. DSPs
- *Multiple Processor Cores on a Single Chip*
  - Dual, quad (or more) cores
  - special purpose cores e.g. DSPs, GPUs, CPMs
- *Multiple Hardware Contexts*
  - Pentium 4 “hyperthreading”

# Concurrency

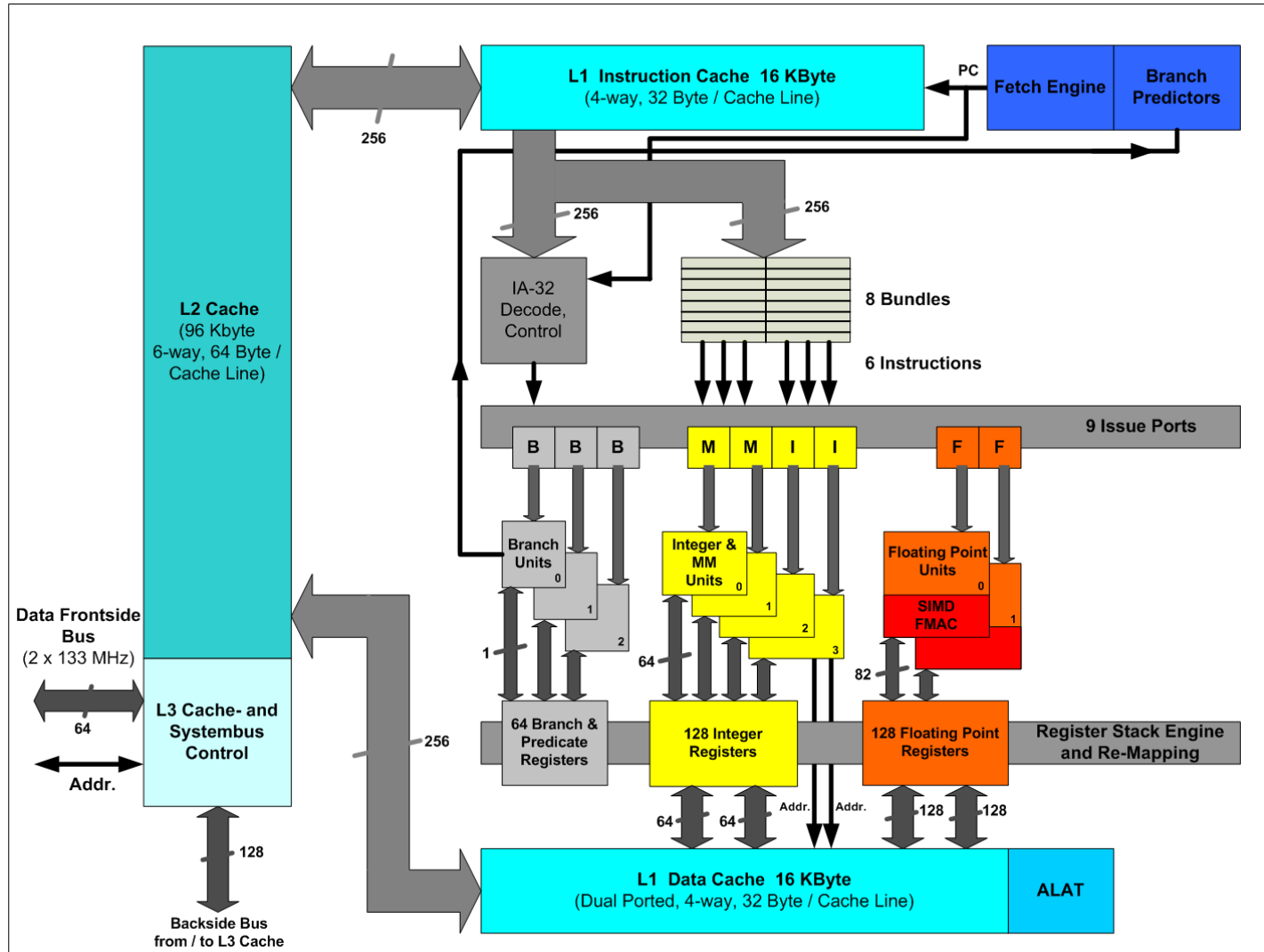


Public Domain 2006 Dennis Schmitz

## Concurrency

- You may effectively have concurrency.
- Even when you may not expect it.
- It may not be obvious from the source code.
- This may be true even on “single processor” systems.
- This has lead some to desperate measures:
  - unable to adapt legacy code to multi-core servers;
  - disabled hyperthreading on single-core servers;
  - freezing the product line on the CPU power curve.

# Intel Itanium Architecture



GNU FDL Appaloosa 2006

## Implications

- Memory optimizations give threads a potentially different concurrent view of the same variables.
- Concurrency gives them an opportunity to do so.
- Compiler and processor optimizations prevent you from fixing it in some “clever” way in your higher-level code.
  
- It can be very difficult to reason while debugging.
- And easy to draw the wrong conclusions.

# Memory Models and Execution Order

## Abstract Machines and Memory Models

- Higher level languages define an *abstract machine* on top of which the compiled code runs.
- The abstract machine implements a *memory model* that assumes how memory is accessed and modified.
- How one maps such an abstract machine and memory model onto actual hardware is not a new issue, e.g. [Lampport] circa 1979.
  
- Java and C# explicitly implement their abstract machine.
- C and C++ have an implicit abstract machine.

## Memory Model

- a.k.a. Shared Memory Consistency Model
- Defines how reads and writes to memory are ordered.
  - relative to one another in a single thread: *program order*
    - maintains the illusion of sequential execution
  - relative to one another across all threads: *total order*
    - determines how and when writes become visible to other threads
- Influences how you design your memory subsystem.
- And your cache coherency scheme.
- And your instruction pipeline.
- And what compiler optimizations you will permit.

## Question

```
int x, y;
```

Thread 1

```
x = x_update;  
y_copy = y;
```

Thread 2

```
y = y_update;  
x_copy = x;
```

Each thread has *one* possible program order.

There are *six* different possible total orders.

Who gets an updated value?

Total order shows that *at least* one thread does so.

## Classifications of Memory Models

- Sequential Consistency (SC) [Lamport]
  - “All actions occur in a total order that is consistent with program order and each read  $r$  of a variable  $v$  sees the value written by the write  $w$  to  $v$  such that
    - $w$  precedes  $r$  and
    - there is no  $w'$  such that  $w$  precedes  $w'$  and  $w'$  precedes  $r$ .”
  - (Java Language Specification 3<sup>rd</sup> Edition)*
- Processor Consistency (PC)
  - Relaxes write-to-read program order.
  - Writes to memory are delayed but program order is preserved.
- Weak Ordering (WO)
  - Relaxes all orders.
  - Writes to memory may occur in any order.

## Answer

```
int x, y;
```

Thread 1

```
x = x_update;  
y_copy = y;
```

Thread 2

```
y = y_update;  
x_copy = x;
```

SC: *at least one* of T1 or T2 gets an updated value.

PC: *neither* T1 nor T2 *may* get an updated value.

WO: *neither* T1 nor T2 *may* get an updated value.

## Memory Model Implementations

- Sequential Consistency (SC) [Lamport]
  - No commercial implementation on multi-core processors.
  - Although some do so for threads on a single core.
  - Too restrictive.
  - Would eliminate even many compiler optimizations.
- Processor Consistency (PC)
  - IBM 370
  - Intel Pentium 4
- Weak Ordering (WO)
  - DEC Alpha
  - IBM/Motorola/Freescale PowerPC
  - *(This is expected to be the trend.)*

# A Conspiracy of Physics and Economics

## Why Is This Suddenly an Issue?

- Mainstream clock rates have stalled around 3GHz.
  - heat
  - power consumption
  - cost of fabrication
- Increasingly complex cores too hard to test and debug.
  - Subtle bugs appearing in mainstream processors.
- Multiple CPU cores are the trend.
  - general purpose: Intel Quad-Core
  - System on Chip (SoC): Freescale PowerQUICC
- Such architectures have become mainstream (cheap).

# Memory Barriers

## Memory Barriers

- Special machine instructions
- Semantics
  - read fence (*acquire* semantics)
    - Addresses the visibility of read-after-write operations from the point of view of the reader.
  - write fence (*release* semantics)
    - Addresses the visibility of read-after-write operations from the point of view of the writer.
  - full fence
    - Insures that all load and store operations prior to the fence have been committed prior to any loads and stores following the fence.
- Using memory barrier instructions alone is *not* sufficient.
  - They only impact re-orderings performed by the processor, not by the compiler.

## Explicit Memory Barriers in Instruction Sets

- Pentium 4
  - cpuid, xchg, lock, mfence, lfence, sfence
- Power PC
  - sync, isync, lwsync, eieio
- Alpha
  - mb, wmb

## Memory Barriers in Higher Level Languages

- Linux: macros specific to each architecture
  - `mb( )`, `rmb( )`, `wmb( )`
  - `smp_mb( )`, `smp_rmb( )`, `smp_wmb( )`
- GCC: evolving built-in memory barrier functions
  - `__sync_synchronize`
  - `__sync_lock_test_and_set`
  - `__sync_lock_release`
- Java: 1.5 and later
  - `volatile`
  - `synchronized`
- .NET: V2 and beyond
  - `volatile`
  - `Thread.MemoryBarrier`

## Memory Barriers in Higher Level Languages

- POSIX Threads
  - `pthread_mutex_lock`
  - and others

## Memory Barriers

- Thread implementations *must* use these barriers.
  - Java 1.5 and beyond
  - POSIX Threads (“pthreads”)
  - RTOSes like vxWorks etc.?
- Applications *must* use thread synchronization primitives.
  - Java `volatile` (Java 1.5 and later) or `synchronized`
  - POSIX `pthread_mutex_lock` etc.
- Or else hand-code memory barrier instructions.
- Using `volatile` alone in C/C++ is *not* sufficient.
  - It only impacts re-orderings performed by the compiler,
  - not by the processor.

# Standards

## Java Language Specification (3<sup>rd</sup> Edition)

- “Compilers are allowed to reorder the instructions in either thread, when this does not affect the execution of that thread in isolation.” (17.3)
- “An inter-thread action is an action performed by one thread that can be detected or directly influenced by another thread.” (17.4.2)
- “Among all the inter-thread actions performed by each thread  $t$ , the program order of  $t$  is the total order that reflects the order in which these actions would be performed according to the intra-thread semantics of  $t$ .” (17.4.3)

## Java Language Specification (3<sup>rd</sup> Edition)

- Java 1.5 has an explicitly defined memory model.
- It implements `volatile` using memory barriers.
- `synchronized` uses memory barriers appropriately.
- Compiler is prevented from reordering inappropriately.
- Java has the advantage of
  - its compiler and virtual machine being developed in unison, and
  - having an integrated threading model.

## ECMA Common Language Runtime (Partition 1, 11.6)

- .NET appears to have addressed this as well.
- ECMA standard specifies sequential consistency.
- Microsoft guarantees SC in later versions (>V1?).
- .NET `volatile` has semantics similar to Java.
- Explicit `Thread.MemoryBarrier` provided.

## ANSI C Standard (1990)

- “issues of optimization are irrelevant” (5.1.2.3)
- “At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.” (5.1.2.3)
- “When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on. Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet.” (5.1.2.3)

## ISO C++ Standard (1998)

- “At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.” (1.9:7)
- “At sequence points, volatile objects are stable in the sense that previous evaluations are complete and subsequent evaluations have not yet occurred.” (1.9:11)
- “There is a sequence point at the completion of each full-expression.” (1.9:18)

## ISO C++ Standard (1998)

- “Between the previous and the next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.” (5:4)

## IEEE POSIX Standard (1996)

- “POSIX.1 as it stands now can be thought of as a C Language Binding.” (B.1.1)
- “The thread model is based on a well-understood synchronous, procedural model consistent with the C function model.” (B.1.1.2)
- “... a portable multithreaded program, or a multiprocess program that shares writeable memory between processes, has to use the synchronization primitives to synchronize data access. It cannot rely on modifications to memory being observed by other threads in the order written in the program or even on modification of a single variable being seen atomically.” (B.2.3.8)

## IEEE POSIX Standard (1996)

- “Formal definitions of the memory model were rejected as unreadable by the vast majority of programmers.”  
(B.2.3.8)

## ANSI C, ISO C++

- Neither C nor C++ define an explicit memory model.
- Nor any integrated threading model.
- They describe semantics in terms of observable behavior.
- And generally from the point of view of a single thread.
- Standards mostly deal with when side effects are visible.

```
i = 7, i++, i++;      /* specified */
i = i + 1;           /* specified */
i = v[i++];          /* unspecified */
i = ++i + 1;         /* unspecified */
```

## IEEE POSIX

- But POSIX threads *do* guarantee correct behavior,
- through the use of both compiler and memory barriers,
- if and only if you use mutexes appropriately.
- POSIX warns you explicitly not to make assumptions.
- It makes no guarantees about its use in C++.

## Some Related Issues

## When Is Memory Actually Accessed?

```
volatile int *shared;
*shared; /* read in ANSI C */

class Foo;
volatile Foo *shared;
*shared; // no read in ISO C++ (error in GCC)

class Foo { public: int bar; };
volatile Foo *shared;
*shared; // no read in ISO C++ (read in GCC!)
```

## **GCC 4.1.1**

- GCC defines built-in functions for atomic memory access. (5.4.4)
  - `__sync_synchronize`
  - `__sync_lock_test_and_set`
  - `__sync_lock_release`
- Compatible with *Intel Itanium Processor-Specific API*.
- Not all built-ins are defined on all processor targets.

## **GCC 4.1.1**

- GCC defines when a `volatile` is accessed. (6.1)
- Covers some subtle C versus C++ versus GCC cases.
- This is particularly important with memory mapped I/O.

## `volatile` in C and C++ is Problematic

- It is frequently miscompiled. [Eide and Regehr]
- Its use is still subject to processor reordering.
- `volatile` does not imply *atomic*.
  - word size and MMU
  - cache line size
  - word tearing

## Non-Portable Memory Barrier Implementations

- Linux 2.4, gcc 3.3.2, i386

```
#define mb() \  
asm volatile \  
( "lock; addl $0,0(%%esp)" : : : "memory" );
```

- Linux 2.6, gcc 4.1.0, IA32

```
#define mb() \  
asm volatile \  
( "mfence" : : : "memory" );
```

## POSIX Threads (libpthread) MP Implementation

- Linux 2.6, glibc 2.4, i386/i486

```
#define __arch_compare_and_exchange_val_32_acq\  
    (mem, newval, oldval) \  
( {  typeof(*mem) ret; \  
    asm volatile ("lock; cmpxchgl %2, %1" \  
        : "=a" (ret), "=m" (*mem) \  
        : "r" (newval), "m" (*mem), "0" (oldval)); \  
    ret; })
```

## Portable Memory Barrier Implementation

- Digital Aggregates open source Desperado library  
<http://www.diag.com/navigation/downloads/Desperado.html>
- Makes use of temporary POSIX thread mutex.
- Not terribly efficient.
- But not as inefficient as it may look.
- May be just as simple (and safer) just to use a mutex.

```
int desperado_portable_barrier() {
    int value = 0;
    int rc;
    int state;
    pthread_mutex_t mutex;
    rc = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &state);
    if (0 != rc) {
        value = -1;
    } else {
        rc = pthread_mutex_init(&mutex, 0);
        if (0 != rc) {
            value = -2;
        } else {
            rc = pthread_mutex_lock(&mutex);
            if (0 != rc) {
                value = -3;
            } else {
                rc = pthread_mutex_unlock(&mutex);
                if (0 != rc) {
                    value = -4;
                }
            }
        }
        rc = pthread_setcancelstate(state, 0);
        if (0 != rc) {
            value = -5;
        }
    }
    return value;
}
```

# Take Away

## Java

- Use Java 1.5 or later.
- Use `synchronized` for shared variables.
- This includes even just for reads.
- Or at least declare shared variables `volatile`.

## .NET

- Use `V2` with explicit memory model.
- Use `volatile` where appropriate.
- Use `Thread.MemoryBarrier` where appropriate.

## C/C++

- Use `pthread_mutex_t` etc. for shared variables.
  - This includes even *read-only* shared variables.
- Or at least declare shared variables `volatile` *and* code explicit memory barriers.
- *Is volatile necessary even when using mutexes?*
  - Probably too conservative.
  - But it depends on your implementation.
  - Yet it may be necessary for other reasons.
    - e.g. memory mapped I/O

## Embedded vs. Applications Developers

## Typical Reactions

- Embedded developers nod and yawn.
  - *Embedded developers have been dealing with concurrency, multiple processors, and shared memory issues for years.*
- Application developers clutch chests and seize.
  - *The multi-core future means that the free ride for application developers is over.*

“Multithreading is just one damn thing after, before, or simultaneous with another.”

-- Andrei Alexandrescu

## Principal Sources

David Bacon et al., *The “Double-Checked Locking is Broken” Declaration*,

<http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>

Ulrich Drepper, *What Every Programmer Should Know About Memory*, November 2007,

<http://people.redhat.com/drepper/cpumemory.pdf>

Eric Eide and John Regehr, “Volatiles are Miscompiled, and What to Do about It”, *Proceedings of the Eighth ACM and IEEE International Conference on Embedded Software*, October 2008,

<http://www.cs.utah.edu/~regehr/papers/emsoft08-preprint.pdf>

Scott Meyers et al., “C++ and the Perils of Double-Checked Locking”, *Dr. Dobb’s Journal*, #362, July 2004, #364, August 2004

David Patterson et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, EECS, U.C. Berkeley, UCB/EECS-2006-183, December 2006,

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>

Arch D. Robison, “Memory Consistency & .NET”, *Dr. Dobb’s Journal*, March 2003

Herb Sutter, “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”, *Dr. Dobb’s Journal*, 30.3, March 2005

## Additional Sources

- Sarita Adve et al., “Shared Memory Consistency Models: A Tutorial”, *IEEE Computer*, December 1996
- Andrei Alexandrescu et al., “Memory model for multithreaded C++”, WG21/N1680=J16/04-0120, SC22 POSIX Advisory Group, 2004
- Hans Boehm, “Memory model for multithreaded C++: August 2005 status update”, WG21/N1876=J16/05-0136, SC22 POSIX Advisory Group, 2005
- Hans Boehm, “Memory Model for C++: Status update”, WG21/N1911=J16/0171, October 2005
- Hans Boehm, “Memory Model Overview”, WG21/N2010=J16/0080, April 2006
- Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
- James Gosling et al., *The Java Language Specification*, Third Edition, Addison-Wesley, 2005
- Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual*, Intel, 2006
- ISO, *Information technology - Portable Operating System Interface (POSIX)*, ISO/IEC 9945-1:1996(E), IEEE 1003.1, ISO, 1996
- ISO, *Programming languages – C++*, ISO/IEC 14882:1998, ISO, 1998
- Leslie Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs”, *IEEE Transactions on Computers*, 9.29, 1979
- Nick Maclaren, “Why POSIX Threads are Unsuitable for C++”, WG21/N1940=J16/06-0010, February 2006
- Scott Meyers, “Double-Checked Locking, Threads, Compiler Optimizations, and More”, <http://www.aristeia.com>, 2004

## Additional Sources

William Pugh, "The Java Memory Model is Fatally Flawed", *Concurrency: Practice and Experience*, 12.6, 2000

Herbert Schildt, *The Annotated ANSI C Standard*, (incorporates ANSI/ISO 9899-1990), McGraw-Hill, 1990

Richard Stallman, *Using the GCC Compiler Collection (GCC)*, GCC 4.1.1, GNU Press, October 2003

Freescale, *e300 Power Architecture Core Family Reference Manual*, e300CORERM, Rev. 4, Freescale Semiconductor, 12/2007