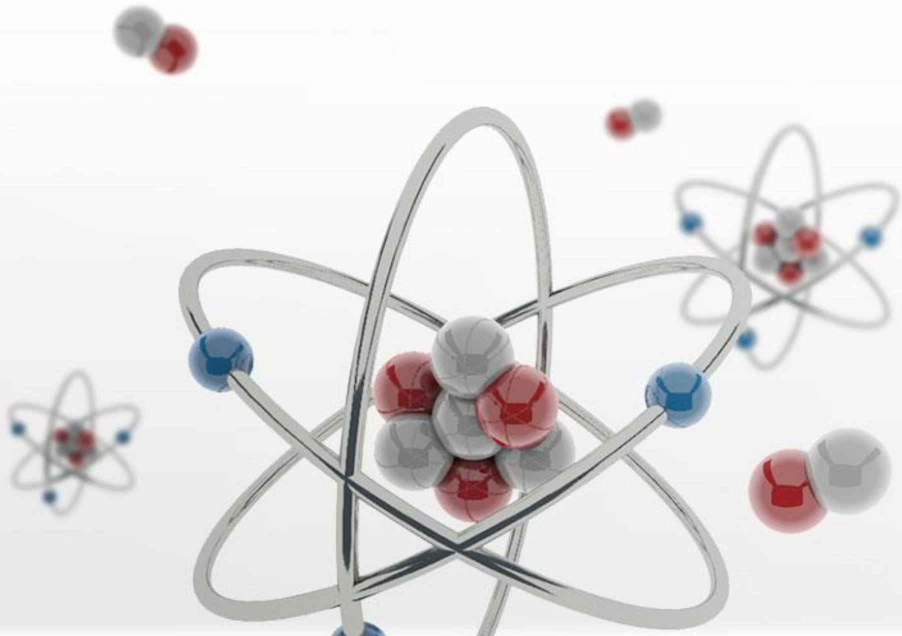
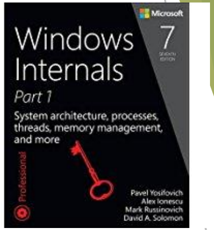


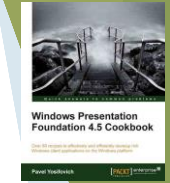
Concurrency and the C++ Memory Model

Pavel Yosifovich
@zodiacon

About Me

- Developer, trainer, author, speaker
- Author
  - Windows Internals 7th edition Part 1 (2017)
  - WPF 4.5 Cookbook (2012)
  - Mastering Windows 8 C++ App Development (2013)
- Pluralsight Author (www.pluralsight.com)
- Microsoft MVP
- Blog: http://blogs.Microsoft.co.il/pavely
- Open source projects on GitHub (http://github.com/zodiacon)





The C++ Standards

- Before the C++11 standard, the C++ standard was C++98
  - C++03 exists as well, with some fixes for C++ 98
- Since 2011, C++ standards have been making steady marches every 3 years
- C++ 17 is the latest approved C++ standard
- C++ 20 is already in the works

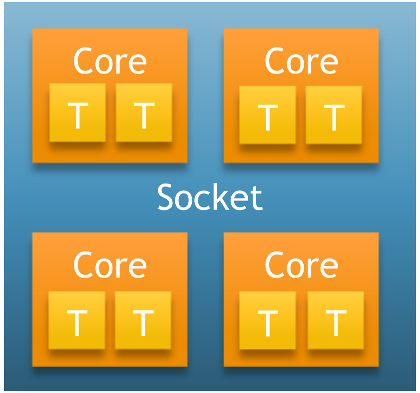
Concurrency and the C++ Standards

- In the C++ 98 standard, the word “thread” is never mentioned
  - Does this mean no threads were used?
- Many different libraries were used for threading
  - boost, TBB, OpenMP, MFC, …
- Starting from C++ 11
  - Threads are part of the standard
  - Including a memory model
  - Enhancements in C++ 14/17/20

Why Concurrency?

- Really just two possible reasons
  - Maximizing performance by the many CPU cores (and/or GPU threads) on the machine
  - Structural benefits
- Designing for concurrency
  - Need to think about the problem at hand before coding begins
  - Difficult to add concurrency at a later stage
    - May introduce subtle bugs and increase code complexity significantly

CPUs



- Socket
  - Physical chip placed on the motherboard
- Core
  - Separate computation unit
- Hardware thread
  - Partially separated computational unit (shares some cache with other HTs within the same core)
  - Several of those may be part of a single core
- Hyper-threading
  - Intel technology that provides two hardware threads per core
  - Similar technology exists in AMD processors
- Logical processor = hardware thread

(Simple?) Example

- Summing up matrix elements

```
long long SumMatrix1(Matrix<int>& m) {
    long long sum = 0;
    for (int r = 0; r < m.Rows(); ++r)
        for (int c = 0; c < m.Columns(); ++c)
            sum += m[r][c];

    return sum;
}
```

Row Major

```
long long SumMatrix2(Matrix<int>& m) {
    long long sum = 0;
    for (int c = 0; c < m.Columns(); ++c)
        for (int r = 0; r < m.Rows(); ++r)
            sum += m[r][c];

    return sum;
}
```

Column Major

Matrix Summation Results

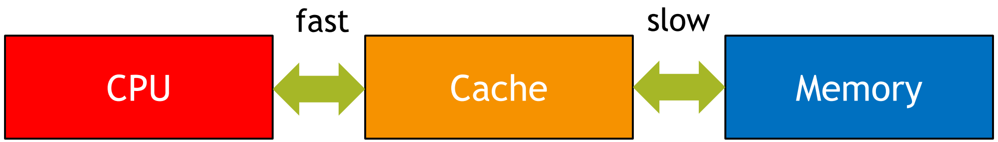
- Intel Core i7-7700HQ
- Visual Studio 2017 15.6 compiler
- x64

C:\WINDOWS\system32\cmd.exe

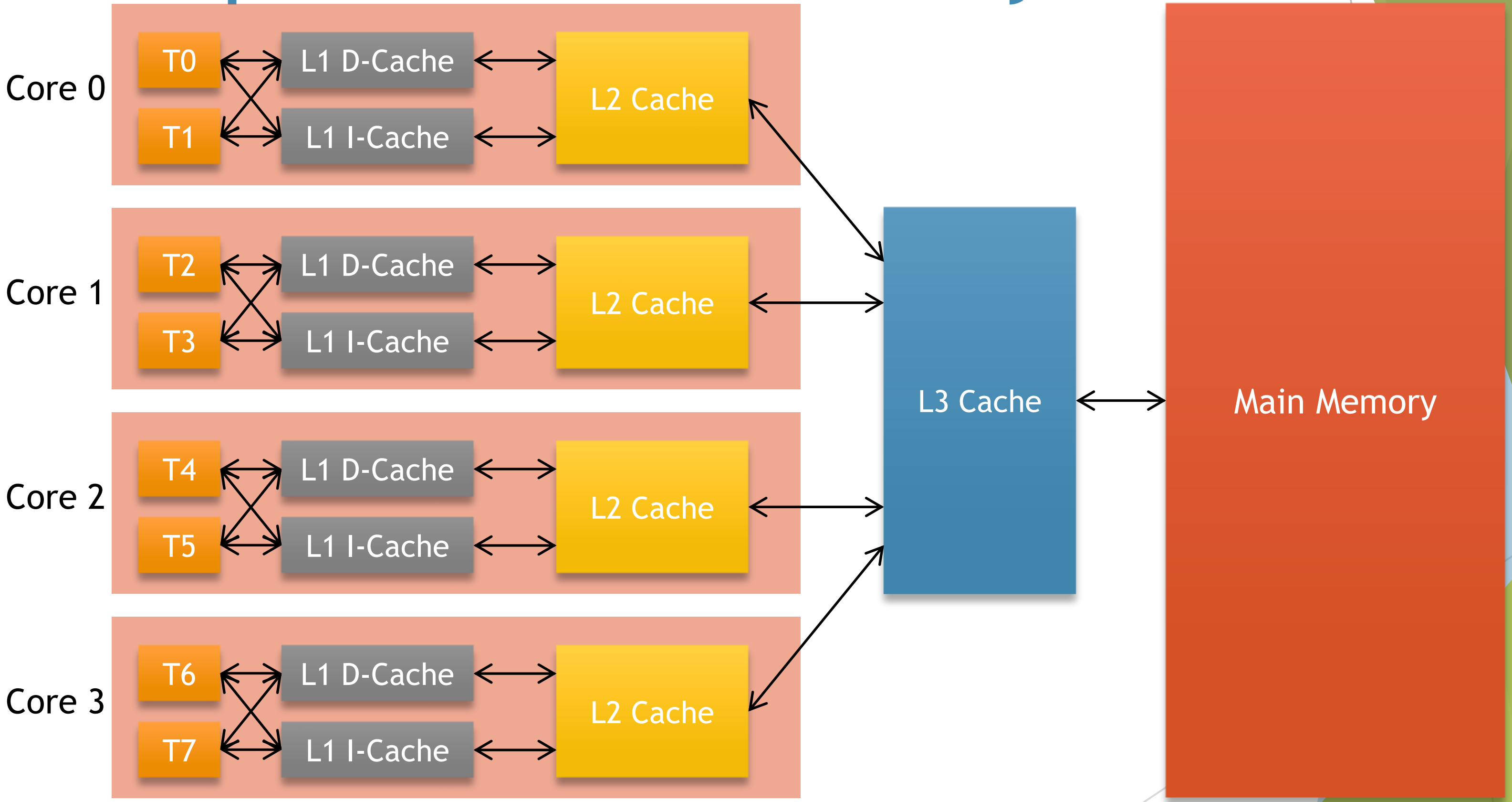
```
Type              Size                  Sum           Time (nsec)
-----------------------------------------------------------------
Row major      256 X 256           2147516416               24700
Col Major      256 X 256           2147516416              113900

Row major      512 X 512          34359869440              132500
Col Major      512 X 512          34359869440              600400

Row major    1024 X 1024         549756338176              453300
Col Major    1024 X 1024         549756338176             4612200

Row major    2048 X 2048        8796095119360             2709800
Col Major    2048 X 2048        8796095119360            56433500

Row major    4096 X 4096      140737496743936             8303600
Col Major    4096 X 4096      140737496743936           259841700

Row major    8192 X 8192     2251799847239680            45701700
Col Major    8192 X 8192     2251799847239680          1296682600

Row major  16384 X 16384    36028797153181696           142691700
Col Major  16384 X 16384    36028797153181696          7482542900

Row major  32768 X 32768   576460752840294400           655578300
Col Major  32768 X 32768   576460752840294400         46590117200

Press any key to continue . . .
```

CPU, Memory and Caches

- In earlier days of processors, CPU and memory speeds were comparable
  - This is no longer the case
- Cache(s) were introduced between CPU and memory
- Cache is small, fast memory
  - Holds recently accessed data/code



Example Cache Hierarchy



Cache Sizes and Cache Lines

- Example cache sizes
  - L1: 32 KB
  - L2: 256 KB
  - L3: 8 MB
- Caches don’t work on single byte entities
- Rather, work on cache lines
  - Typical size is 64 bytes
- Accessing a single byte reads/writes an entire cache line
  - i.e. arrays are fastest as far as hardware is concerned

Another Example

- Counting the number of even numbers in an array with parallel threads

```
int CountEvenNumbers1(const int* data, int size, int nthreads) {
    auto counters_buffer = make_unique<int[]>(nthreads);
    auto counters = counters_buffer.get();

    int chunk = size / nthreads;
    vector<thread> threads;

    for (int i = 0; i < nthreads; i++) {
        int start = i * chunk;
        int end = i == nthreads - 1 ? size : (i + 1) * chunk;

        thread t([data, counters](int index, int start, int end) {
            for (; start < end; ++start)
                if (data[start] % 2 == 0)
                    ++counters[index];
        }, i, start, end);

        threads.push_back(move(t));
    }
```

```
    for (auto& t : threads)
        t.join();

    int sum = 0;
    for (int i = 0; i < nthreads; i++)
        sum += counters[i];
    return sum;
}
```

```
1 threads count: 536870912 time: 1034540 usec
2 threads count: 536870912 time: 777712 usec
3 threads count: 536870912 time: 607431 usec
4 threads count: 536870912 time: 542888 usec
5 threads count: 536870912 time: 433128 usec
6 threads count: 536870912 time: 454097 usec
7 threads count: 536870912 time: 512473 usec
8 threads count: 536870912 time: 634788 usec
```

False Sharing

- Sharing cache lines being written by different threads

```cpp
thread t([data, counters](int index, int start, int end) {
    // use local counter
    int count = 0;
    for (; start < end; ++start)
        if (data[start] % 2 == 0)
            ++count;

    // write result just once
    counters[index] = count;
}, i, start, end);
```

```
1 threads count: 536870912 time: 470639 usec
2 threads count: 536870912 time: 251442 usec
3 threads count: 536870912 time: 220446 usec
4 threads count: 536870912 time: 194796 usec
5 threads count: 536870912 time: 173443 usec
6 threads count: 536870912 time: 170039 usec
7 threads count: 536870912 time: 165969 usec
8 threads count: 536870912 time: 164850 usec
```

Simple(?) Example

- What is the value of *b*?
  - 5 or 0?

```
int a = 0;
volatile int flag = 0;

thread t1([&]() {
      while (flag != 1)
         ;

      int b = a;
      cout << "b = " << b << endl;
});

thread t2([&]() {
      a = 5;
      flag = 1;
});

t1.join();
t2.join();
```

Some Definitions

- Byte
  - Smallest addressable unit of memory
- Memory location
  - An object of scalar type (arithmetic, pointer, enum or `nullptr_t`)
  - Or the largest contiguous sequence of non-zero length bit fields
- Thread
  - Independent flow of control within the program
- Accessing different memory locations concurrently by different threads is always safe
- Data race
  - When a thread writes to a memory location and another thread reads from the same memory location at the same time

Dekker's Algorithm

- Poor man's critical section

Thread 1

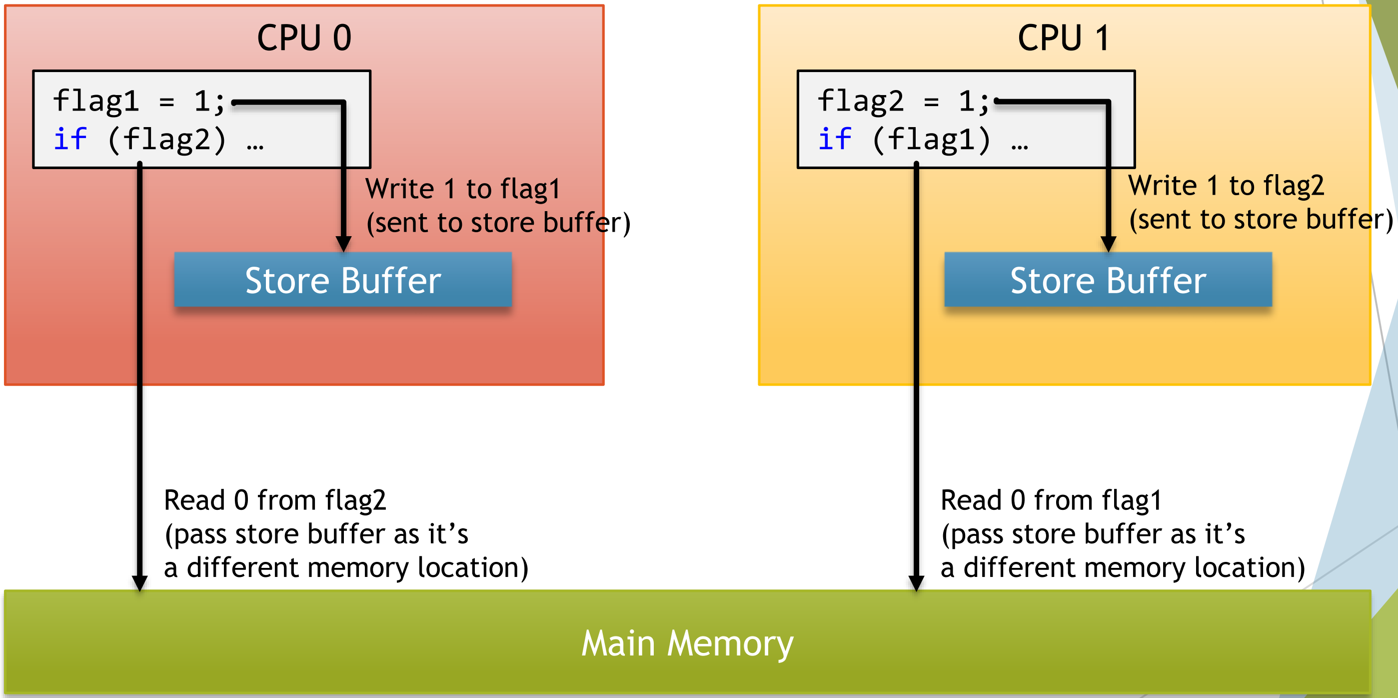
```
flag1 = 1;
if (flag2) {
        // back off
}
else {
        // enter critical section
}
```

Thread 2

```
flag2 = 1;
if (flag1) {
        // back off
}
else {
        // enter critical section
}
```

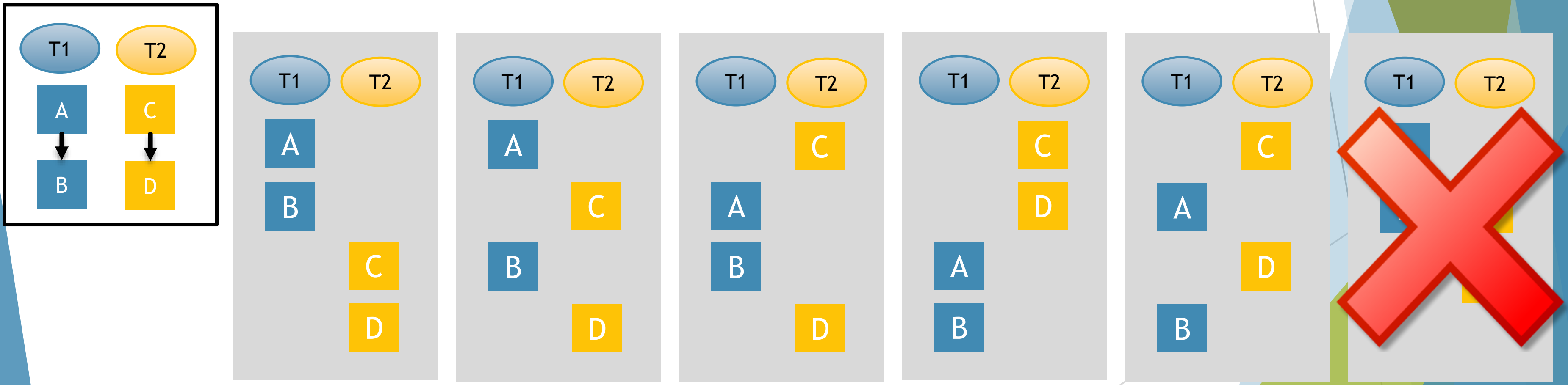
- Can thread 1 and thread 2 enter the critical section at the same time?

Dekker's Algorithm Executed



Sequential Consistency

- The result of any execution is the same as if
  - The operation of each thread appears as specified in program order
  - Operations of all threads were executed in some sequential order atomically



SC-DRF

- Sequential Consistency may be too strict to get without significant performance penalty
- Compromise
  - SC for Data Race Free programs
- In other words
  - If program guarantees no data races
  - Then compiler/runtime/hardware guarantee Sequential Consistency

Optimizations

- The complier knows
  - All memory operations in this thread, what they do, including any data dependencies
  - How to be conservative enough in face of possible aliasing
- The compiler does not know
  - Which memory locations are “mutable shared” between threads
  - Even if it did, it wouldn’t know the sharing semantics
  - How to be conservative enough in case of possible sharing
- Programmer must somehow let the compiler know

Optimization Examples

- Example single thread optimizations

```
x = 1;
s = "hello";
x = 2;
```

```
s = "hello";
x = 2;
```

```
for (int i = 0; i < len; i++)
    z += a[i];
```

```
r = z;
for (int i = 0; i < len; i++)
    r += a[i];
z = r;
```

```
s1 = "hello";
s2 = "cruel";
s3 = "thread";
```

```
s3 = "thread";
s2 = "cruel";
s1 = "hello";
```

Data Race Prevention

- A data race can be prevented by the following
  - Reads and writes are performed as atomic operations (`std::atomic<>`)
  - One of the conflicting operations *happens-before* another

Data Race

```
int count = 0;

auto inc = [&]() {
    for (int i = 0; i < 1000000; i++)
        count++;
};

thread t[]{ thread(inc), thread(inc),
    thread(inc), thread(inc) };
```

No Data Race

```
atomic<int> count = 0;

auto inc = [&]() {
    for (int i = 0; i < 1000000; i++)
        count++;
};

thread t[]{ thread(inc), thread(inc),
    thread(inc), thread(inc) };
```

Atomic Operations

- An atomic operation is indivisible
  - Partial change cannot be observed by any thread
- If all operations on an object are atomic, a read operation will receive the initial value of the object or one of the atomic modifications made to it
- Conversely, non-atomic operations might be seen as partial results from other threads
- C++ provides atomic types to perform atomic operations

Atomic Types

- The standard atomic types are defined in the `<atomic>` header
  - Template type is `std::atomic<T>`
- Many atomic operations within the atomic types use machine instructions that work atomically on the CPU level
  - Some are not (discussed later)
- The `is_lock_free()` member function indicates whether such operations use atomic CPU instructions
- `std::atomic<>` has specializations for specific types

`std::atomic<>` Member Functions

- The standard atomic types are not copyable or assignable in the conventional sense
- Support assignment operator from a non-atomic corresponding type
  - And an operator T to read the value stored in the atomic
- These are special cases for the `load()` and `store()` functions
  - Also support `exchange()`, `compare_exchange_weak()` and `compare_exchange_strong()`
- Support the compound assignment operators (+= etc.)
- The partial specialization for pointer types also supports the ++ and – operators

atomic<> Exchange Operations

```
T atomic<T>::exchange(T value)
```

- Set a new value and return the old value (atomically)

```
bool atomic<T>::compare_exchange_strong(T& expected, T desired)
```

- If the value is as expected, set to desired value and return true
  - Otherwise, return false (and update expected to the current value)
- `compare_exchange_weak()` allows for spurious failures
  - Always use if in a loop
- The fundamental building block in lock-free programming

Synchronizing Reads and Writes

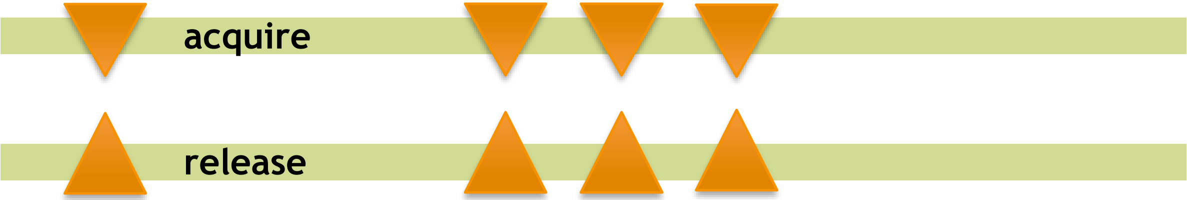
- Example: reading and writing from different threads

```
using namespace std;

vector<int> result;
atomic<bool> ready(false);

void reader_thread() {
    while (!ready.load()) {
        this_thread::sleep_for(chrono::milliseconds(1));
    }
    std::cout << "The answer is " << result[0] << endl;
}

void writer_thread() {
    result.push_back(42);
    ready = true;
}
```

- Why does this work?

Acquire and Release

- One way barriers
- Fundamental concepts of software and hardware
- Acquire == read (load) operation



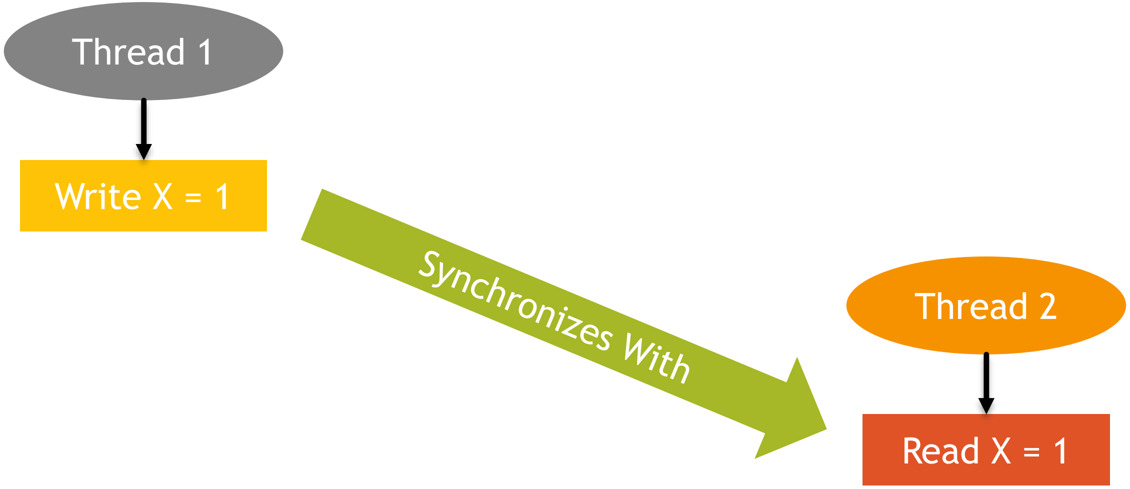
- Release == write (store) operation
- A release store operation makes its prior accesses visible to a thread performing an acquire load that pairs with that store

The *Synchronizes-With* Relationship

- Always comes from atomic types
- A “suitably tagged” write operation on a variable synchronizes-with a read operation on that variable stored by that write
  - Or a subsequent atomic write by the same thread
  - Or a sequence of atomic read-modify-write operations by any thread, where the value read by the first thread in the sequence is the value initially written
- “Suitably tagged” depends on the memory ordering semantics

Synchronizes-With

- The ordering imposed by one thread reading a value that was written by another thread



Fixed Dekker's Algorithm

```
#include <atomic>

std::atomic<int> flag1 = 0, flag2 = 0;

void Thread1() {
    flag1 = 1;
    if (!flag2) {
        // enter CS
    }
    else {
        // back off
    }
}

void Thread2() {
    flag2 = 1;
    if (!flag1) {
        // enter CS
    }
    else {
        // back off
    }
}
```

Synchronizes-with

Synchronizes-with

Memory Ordering for Atomics

- Each operation on the atomic type has an optional memory ordering argument (`memory_order` enum)
- Default is `memory_order_seq_cst` (Sequential Consistency)
  - Always used when invoked through the operators
- Store operations can use (`memory_order_xxx`)
  - `relaxed`, `release` or `seq_cst`
- Load operations can use
  - `relaxed`, `acquire`, `consume` or `seq_cst`
- Read-modify-write operations can use any memory order
  - `relaxed`, `consume`, `acquire`, `release`, `acq_rel`, or `seq_cst`

Relaxed Memory Order

- No global ordering of events
  - But all operations are still atomic
- Threads don’t have to agree on the sequence of events
  - Intra thread events still obey ***happens-before*** rules
- Better to wrap relaxed operations inside types that implement them

Relaxed Memory Order Example

```
#include <atomic>

std::atomic<int> count = 0;

// N workers
void WorkerThread() {
    while(…) {
        if (…) {
            ++count;
        }
    }
}

void main() {
    launch_workers();
    …
    join_workers();
    cout << count << endl;
}
```

```
#include <atomic>

std::atomic<int> count = 0;

// N workers
void WorkerThread() {
    while(…) {
        if (…) {
            count.fetch_add(1, memory_order_relaxed);
        }
    }
}

void main() {
    launch_workers();
    …
    join_workers();
    cout << count.load(memory_order_relaxed) << endl;
}
```

Other Memory Ordering Options

- Acquire/release (`memory_order_acq_rel`)
  - Just below SC
  - Acquire can move above (a previous) release
- Acquire (`memory_order_acquire`)
  - Load (read)
- Release (`memory_order_release`)
  - Store (write)
- Consume (`memory_order_consume`)
  - Most (all) compilers promote to `acquire`
  - Deprecated as of C++ 17 (may be removed in C++ 20)

Slightly Relaxed Dekker's Algorithm

```cpp
#include <atomic>

std::atomic<int> flag1 = 0, flag2 = 0;

void Thread1() {
    flag1 = 1;
    if (!flag2) {
        // enter CS
    }
    else {
        // back off
    }
}

void Thread2() {
    flag2 = 1;
    if (!flag1) {
        // enter CS
    }
    else {
        // back off
    }
}
```

```cpp
#include <atomic>

std::atomic<int> flag1 = 0, flag2 = 0;

void Thread1() {
    flag1.store(1, memory_order_release);
    if (!flag2.load()) {
        // enter CS
    }
    else {
        // back off
    }
}

void Thread2() {
    flag2.store(1, memory_order_release);
    if (!flag1.load()) {
        // enter CS
    }
    else {
        // back off
    }
}
```

The Double Checked Locking Algorithm

- Classic way to get a singleton object
- Fails in today's systems

```
struct widget {
    //...
};

widget* instance = nullptr;
mutex wmutex;

widget* getInstance() {
    if (instance == nullptr) {
        lock_guard lock(wmutex);  // lock_guard<mutex> lock(wmutex) in pre C++17
        if (instance == nullptr)
            instance = new widget();
    }
    return instance;
}
```

Double Checked Locking Algorithm Fixed

- Atomicity and ordering provided by atomics and the memory model

```
struct widget {
    //...
};

atomic<widget*> instance = nullptr;
mutex wmutex;

widget* getInstance() {
    if (instance == nullptr) {
        lock_guard lock(wmutex);
        if (instance == nullptr)
            instance = new widget();
    }
    return instance;
}
```

First check (atomic)

Then acquire lock

Then second check

Then create instance, then assign

Lazy Initialization Alternative

```
atomic<widget*> instance = nullptr;
atomic<bool> create = false;

widget* getInstance() {
    if (instance.load() == nullptr) {
        if (!create.exchange(true))
            instance = new widget();   // construct
        else
            while (instance.load() == nullptr) {}     // spin
    }
    return instance;
}
```

```
atomic<widget*> instance = nullptr;
atomic<bool> create = false;

widget* getInstance() {
    if (instance.load(memory_order_acquire) == nullptr) {
        if (!create.exchange(true))
            instance.store(new widget(), memory_order_release);
        else
            while (instance.load(memory_order_acquire) == nullptr) {}
    }
    return instance.load(memory_order_acquire);
}
```

Lazy Initialization with C++ 11

```
widget* instance = nullptr;

widget* getInstance() {
    static once_flag create;
    call_once(create, [] {
        instance = new widget();
    });
    return instance;
}
```

```
widget* getInstance() {
    static widget instance;
    return &instance;
}
```

- Uses `once_flag` behind the scenes

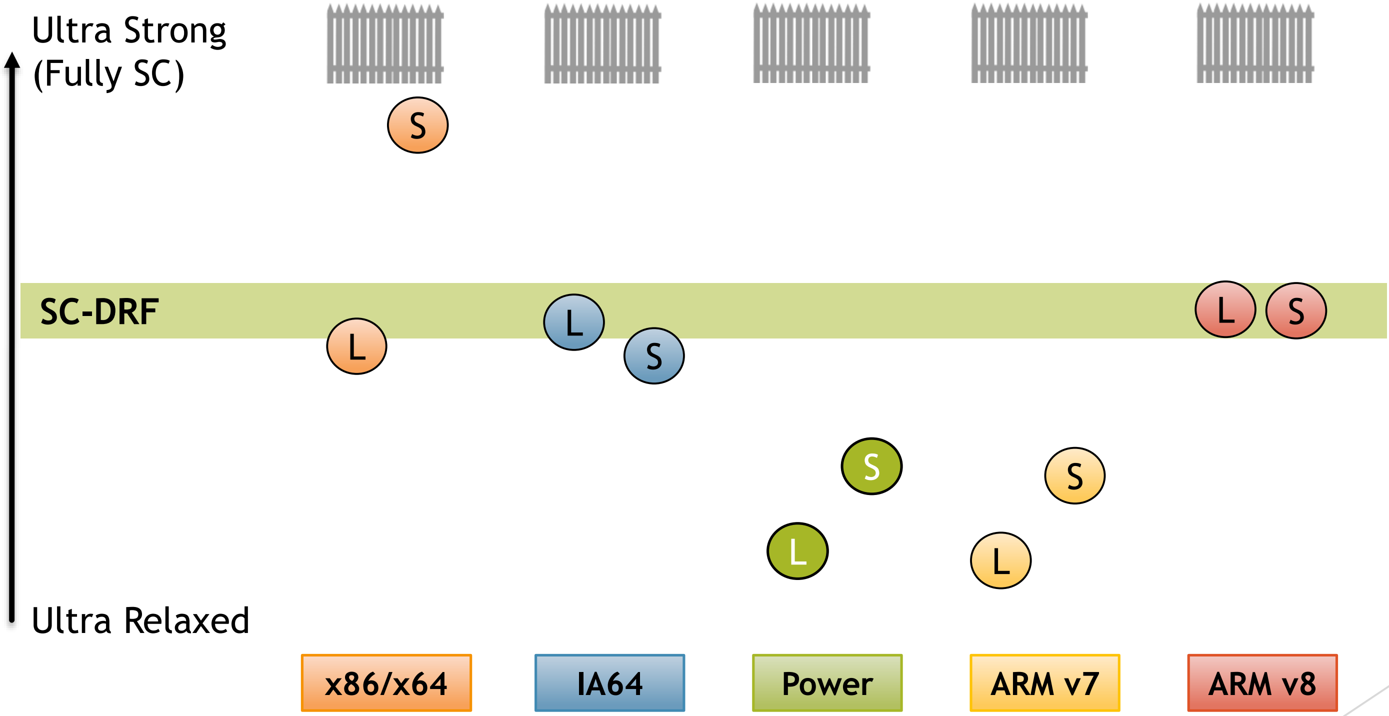
Fences

- Also known as memory barriers
- Prevent instruction moving across the barrier in both directions
- Mostly useful with `memory_order_relaxed`
- Unrelated to a specific memory location
- Can be used to enforce ordering for non atomic variables
- Usage: call the `atomic_thread_fence` function
- Prefer ordering with atomics

SC Atomic Implementation by CPU

CPU	Load Normal / SC atomic	Store Normal / SC Atomic	Compare-and-Swap (CAS)
x86/x64	`mov / mov`	`mov / xchg`	`cmpxchg`
IA 64	`ld / ld.acq`	`st / st.rel;mf`	`cmpxchg.rel;mf`
Power	`ld / sync;ld;cmp;bc;isync`	`st / sync;st`	`sync;_loop:lwarx;cmp;bc _exit;stwcx.;bc _loop;isync;_exit:`
ARM v7	`ldr / ldr;dmb`	`str / dmb;str;dmb`	`dmb;` (compare-exchange loop)
ARM v8	`ldr / ldra`	`str / strl`	

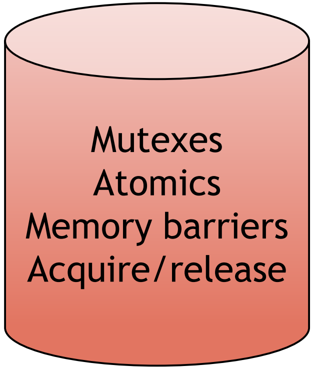
Memory Order Performance by CPU



The `volatile` Keyword

- Volatile in Java & .NET is not the same as C++ volatile
  - Java/.NET volatile is the same as atomic in C/C++

Inside Memory Model



Outside Memory Model

`volatile`

- Volatile variables are unoptimizable
  - Best to think of them as "I/O"

Thank You!