The C++ and CLR Memory Models

Sasha Goldshtein

CTO, Sela Group

@goldshtn

Assumptions

- You are a C++/C# developer
- You write multithreaded code (who doesn’t?)
- You care about the correctness of your code

- You might have gotten used to the nurturing embrace of x86, but now you have to make sure your code is correct in the fiery, dangerous pits of ARM as well

Agenda

- Atomicity, exclusivity, and ordering
- What does “memory model” even mean?
- Examples of memory reorderings
- Volatile and atomic variables
- Examples of broken code and how to fix it

*This is genuinely a level-400 talk. Viewer discretion is advised. Rated **R** because of frequent mentions of memory barriers and reorderings.*

Atomicity, Exclusivity, and Ordering

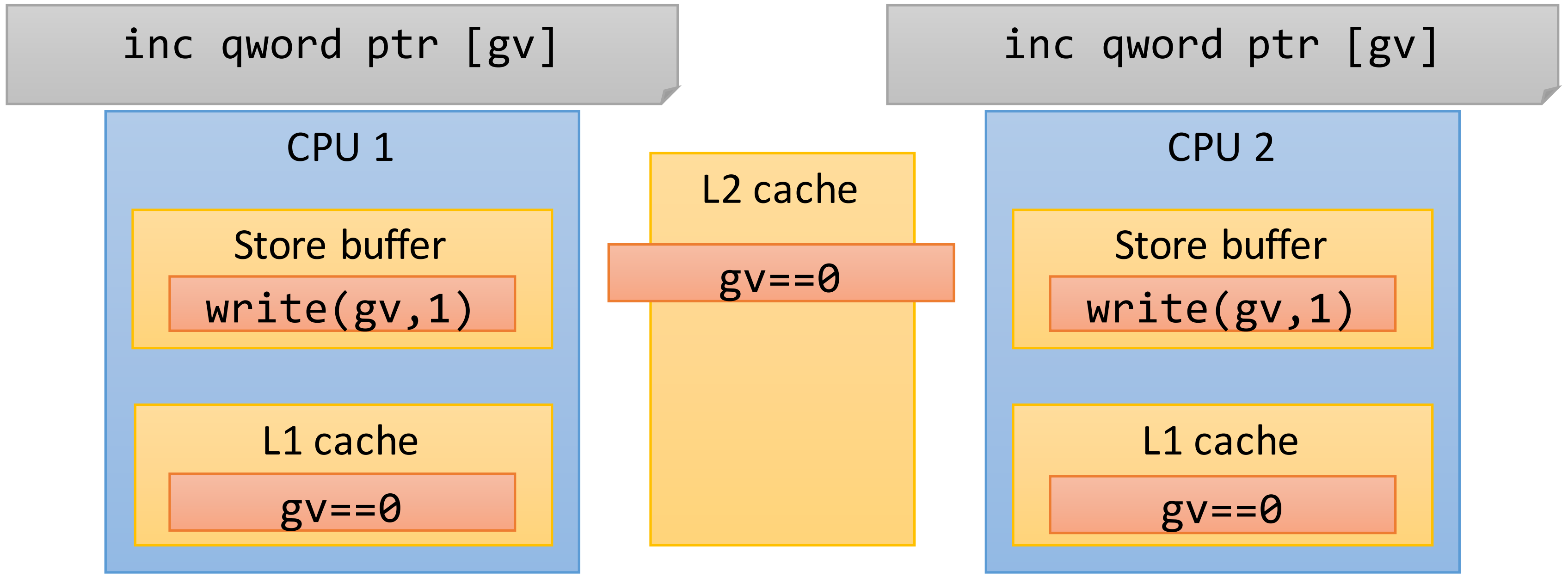
Atomicity

- An atomic operation is non-interruptible
  - Partial reads/writes or context switches aren't allowed
- On Intel x86-64 processors, *aligned* reads and writes of ≤64-bit values are atomic
- Many "trivial" operations, especially with non-optimizing compilers, are not atomic

Original source code	**Resulting x86-64 instructions**
`++globalVar;`	`mov rax, qword ptr [globalVar]`
	`add rax, 1`
	`mov qword ptr [globalVar], rax`

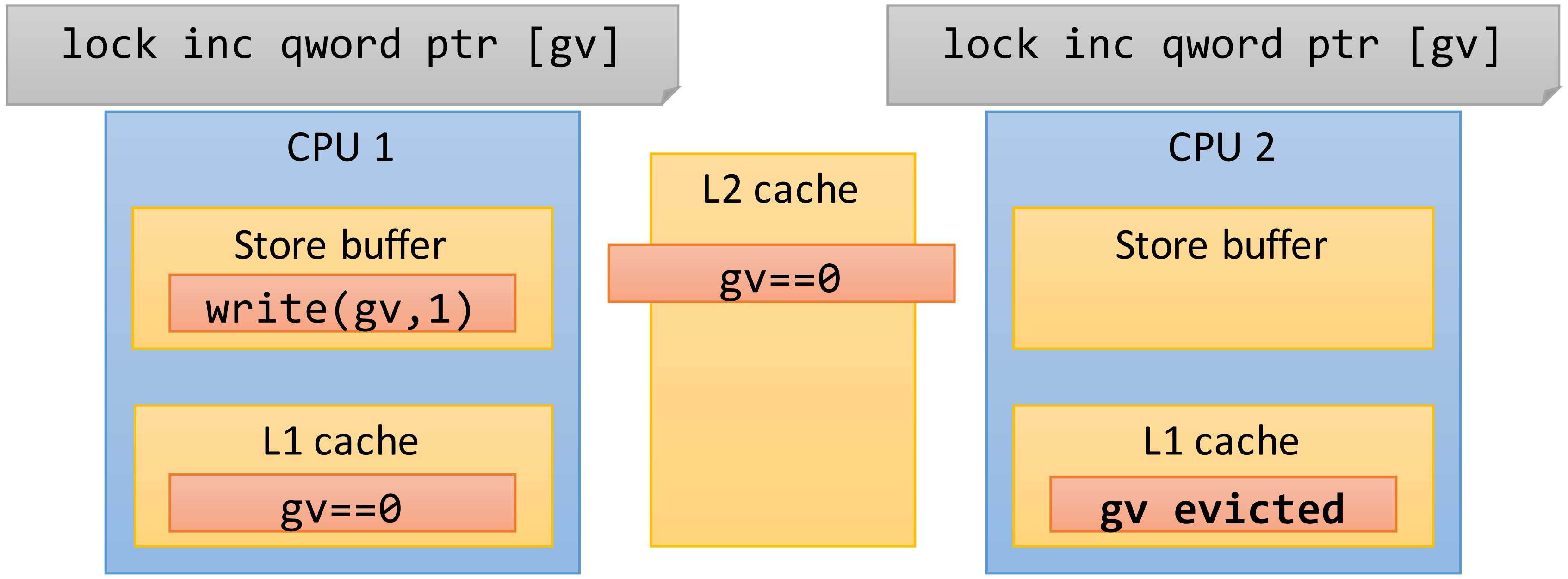
Atomicity Does Not Guarantee Exclusivity

- A non-interruptible operation still does not guarantee exclusive access to memory



Exclusivity

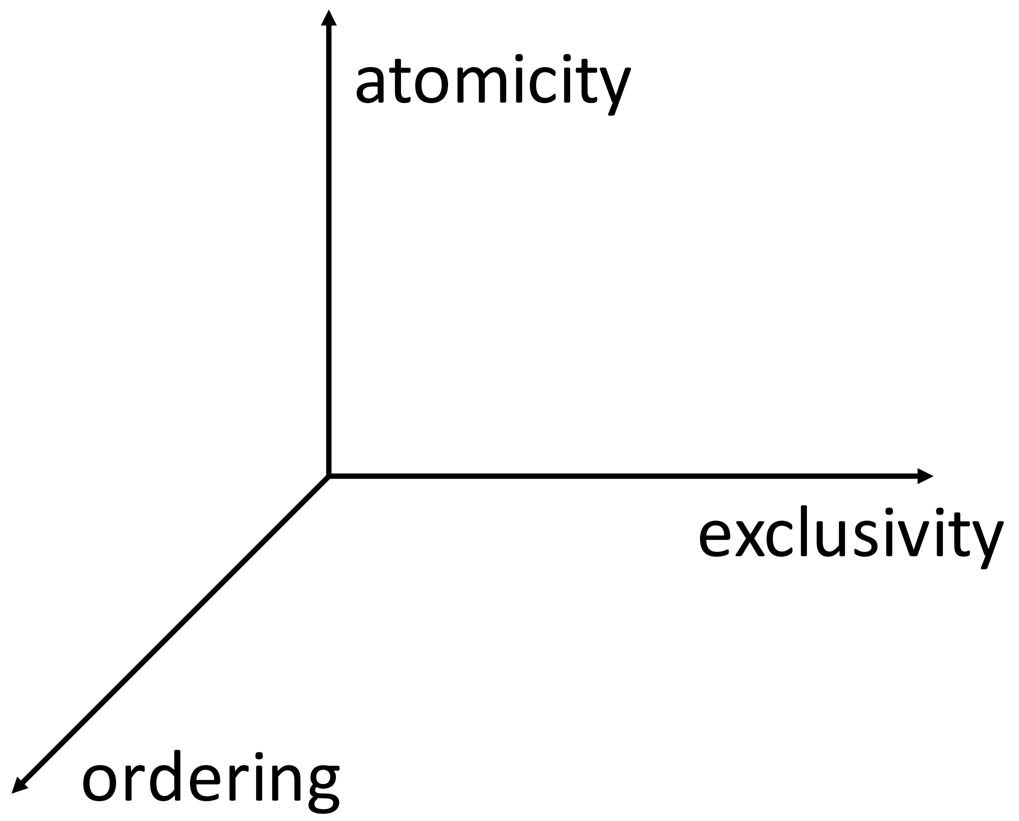
- LOCK: Require exclusive access to memory
  - In the past, achieved by locking the memory bus
  - Currently, achieved by marking the cache line in exclusive mode



Ordering

- Atomicity does not guarantee ordering
- As we will see later, some memory loads/stores may be reordered
  - E.g., stores may become visible to other processors *after* subsequent loads retire
- Processors may disagree on a variable's value
  - A processor may see its own writes *before* other processors see them

Complex Relationships



Memory Ordering

Memory Ordering

- *Question: Does the computer execute the program you wrote?*
- No
- Compilers, processors, and memory controllers issue memory operations in a different order
- It’s not malicious, it’s an optimization!

Compiler Reordering

- Hmm, it sounds like you’re reading that array element many times inside the loop…
- Let me hoist that read out of the loop for ya!
- Sounds like a nice optimization, right?

```
int sum = 0;
int p = get_pivot();
for (int i = 0; i < p; ++i)
{
  sum += data[i] * data[p];
}
```

Compiler Reordering

- Automatic vectorization is essentially compiler reordering

```
// original code:
for (int i = 0; i < n; ++i) {
  dst[i] = src[i];
}

// vectorized (and therefore reordered):
for (int i = 0; i < n; i += 16) {
  auto val = _mm_stream_load_si128((__m128i*)&src[i]);
  _mm_stream_si128((__m128i*)&dst[i], val);
}
```

Compiler Reordering

- Read elimination is a fairly common optimization

```
// original code:
if (arg1 == 0) throw new ArgumentException();
int val = arg2 + 1;
val += arg1;

// optimized (reordered) code:
int tmp = arg1;
if (tmp == 0) throw new ArgumentException();
int val = arg2 + 1;
val += tmp;
```

Compiler Reordering

- How about this reordering?

```
// original code:
void enqueue(X new_element) {
  bounded_queue_[++last] = new_element;
  locked_ = 0;
}

// reordered:
void enqueue(X new_element) {
  locked_ = 0;
  bounded_queue_[++last] = new_element;
}
```

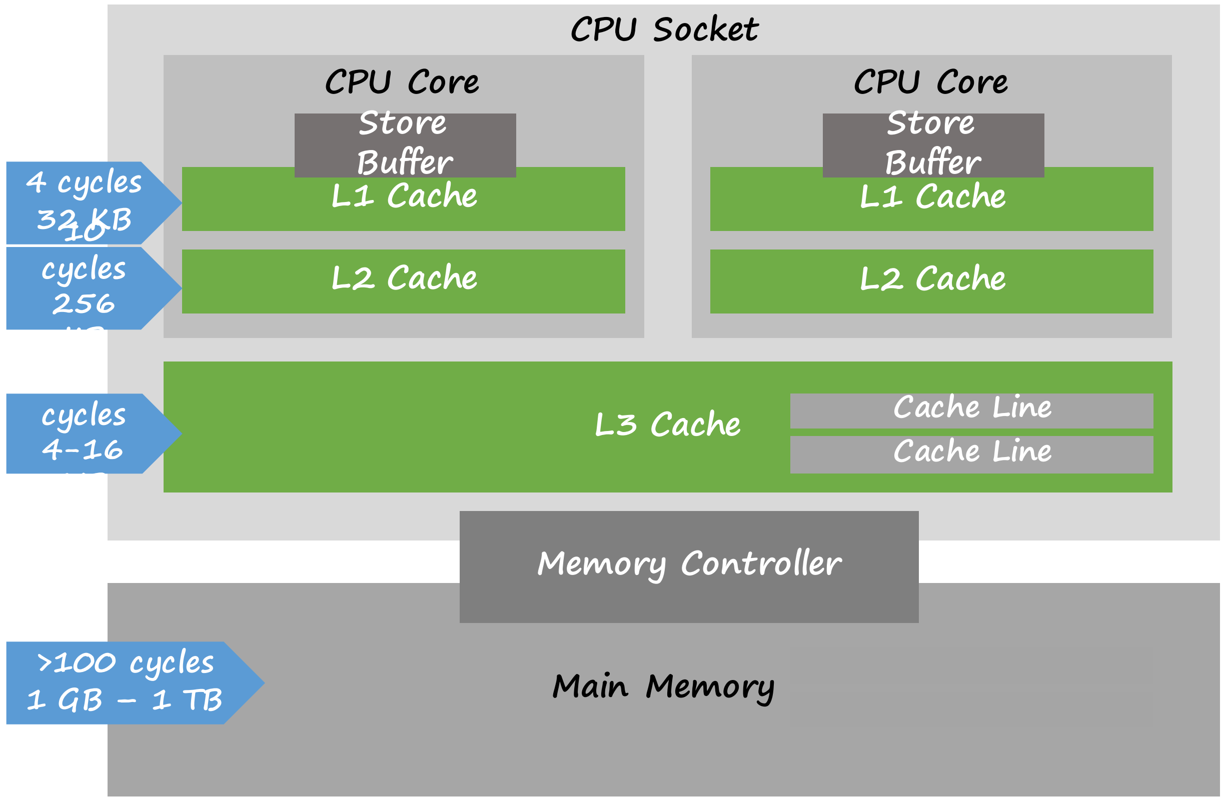
Out-Of-Order Execution

- Processors have deep execution pipelines designed to execute instructions in parallel
- This may cause reordering; specifically, reordering of memory operations

Time	0	1	2	3	4	5
`ADD DWORD PTR [EAX], ECX`	ML	ALU	MS			
`ADD ECX, DWORD PTR [EBX]`		ML	ALU			
`ADD DWORD PTR [EDX], ESI`			ML	ALU	MS	
`ADD EDI, 1`					ALU	

Caches

- Caches and store buffers can dramatically delay memory writes and speed up memory reads



What Kinds of Reorderings Are Permissible?

- Data dependencies must be honored
  - E.g., same thread writes X = 1, then reads X: gets 1
- **Compilers** may reorder any memory access under the *as-if* rule
- **Processors** have different guarantees

	x86, x86-64	**ARMv7, IA64**	**SPARC PSO**
Loads after loads	No	**Yes**	No
Loads after stores	No	**Yes**	No
Stores after stores	No	**Yes**	**Yes**
Stores after loads	**Yes**	**Yes**	**Yes**

Demo: Spin-lock sensitivity on ARM vs. x86-64

Reordering, Example 1

- Assuming g_p is `widget*`…
- Thread 1 may see g_p non-null, but only partially initialized

```
Thread 1                    Thread 2

if (g_p != nullptr)         g_p = new widget();
{
  g_p->do_work();
}
```

Writes performed by widget's constructor aren't necessarily visible here; broken on ARM and SPARC PSO

Reordering, Example 2

- Thread 2 may see `complete` as `true`, but `value` would not be initialized, or only partially initialized

```
Thread 1                              Thread 2


value = SomeComputation();            if (complete)
complete = true;                      {
                                              Use(value);
                                      }
```

Reordering, Example 3

- Peterson’s algorithm for synchronization
- Assuming **flag1**, **flag2** are initialized to 0

Store can pass load, so both threads see the other’s flag as 0 and enter the critical section; **broken even on x86!!!**

```
Thread 1                     Thread 2

START1: flag1 = 1;           START2: flag2 = 1;
if (flag2 == 0) {            if (flag1 == 0) {
  critical section             critical section
} else {                     } else {
  flag1 = 0;                   flag2 = 0;
  goto START1;                 goto START2;
}                            }
```

Demo: Peterson’s algorithm

Sequential Consistency

- Sequential consistency (SC)
  - The result of any execution (reads and writes) on multiple processors requires that the operations of each individual processor execute *in the order specified by the program*
- SC is often incredibly expensive and precludes important optimizations
- Modern compilers and processors do not offer sequential consistency by default

SC-DRF

- Race conditions
  - A memory location can be accessed *simultaneously* by two threads, one of which is a writer
- Sequential consistency for data race free programs (SC-DRF)
  - Executing reads and writes in program order, *as long as you don't have a race condition*
  - Hardware promises sequential consistency if you obey the constraints and don't write race conditions

Memory Models

- The C++11 memory model offers SC-DRF
    - Compiler doesn’t take care of preventing processor reorderings; semantics are hardware-dependent
    - Provides facilities for ensuring undesired reorderings do not occur
- The CLR memory model … which one? ☺
    - ECMA CLI allows all reorderings, Microsoft CLR implementation precludes store-store reorderings
    - Provides facilities for ensuring undesired reorderings do not occur

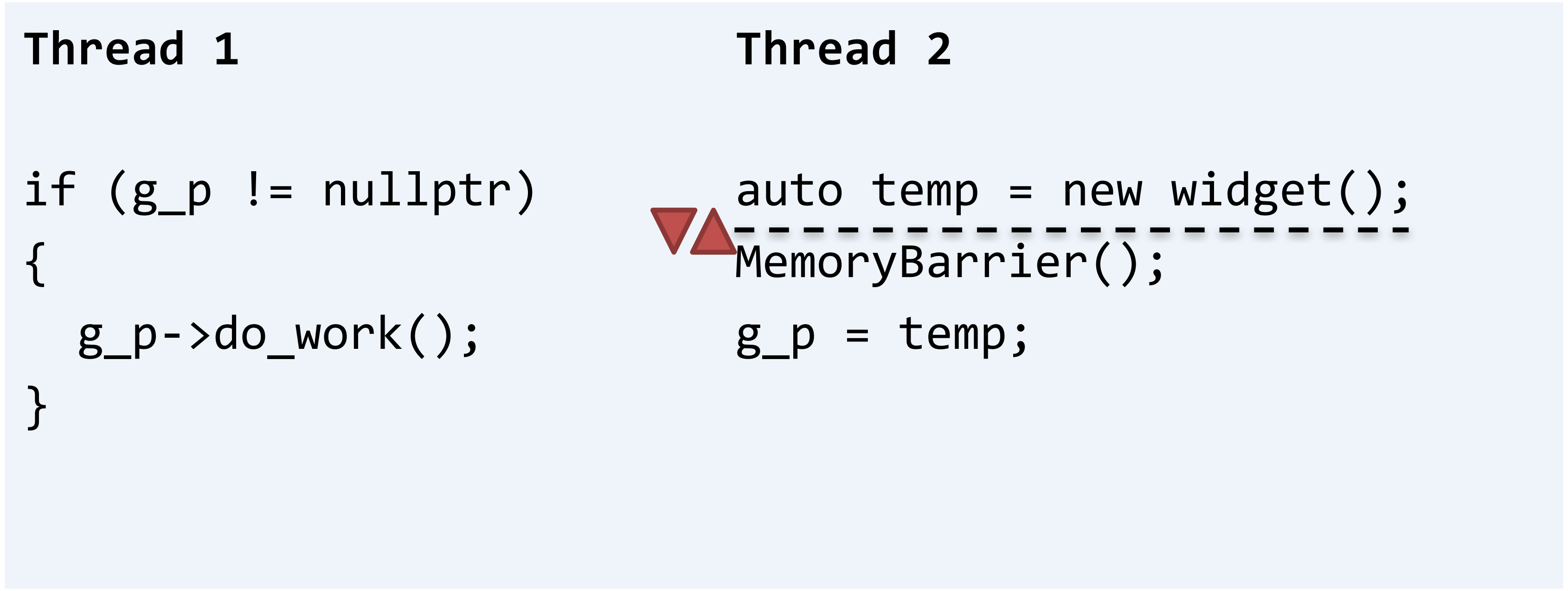
Good Fences Make Good Neighbors

`volatile`

- In C++:
  - Volatile variables prevent compiler reorderings of reads and writes (*to these variables*), and some other optimizations
  - Volatile variables do not prevent processor reorderings (although in some versions they used to ¯_(ツ)_/¯)
- In C#:
  - Volatile variables *additionally* prevent processor reorderings, producing unidirectional barriers

Processor Memory Barriers

- A full barrier prevents all reads and writes from passing the barrier (in any direction)
- That's more than what we need in this case:



```
Thread 1                        Thread 2

if (g_p != nullptr)             auto temp = new widget();
{                             ▼▲------------------------
  g_p->do_work();               MemoryBarrier();
}                               g_p = temp;
```

Processor Memory Barriers

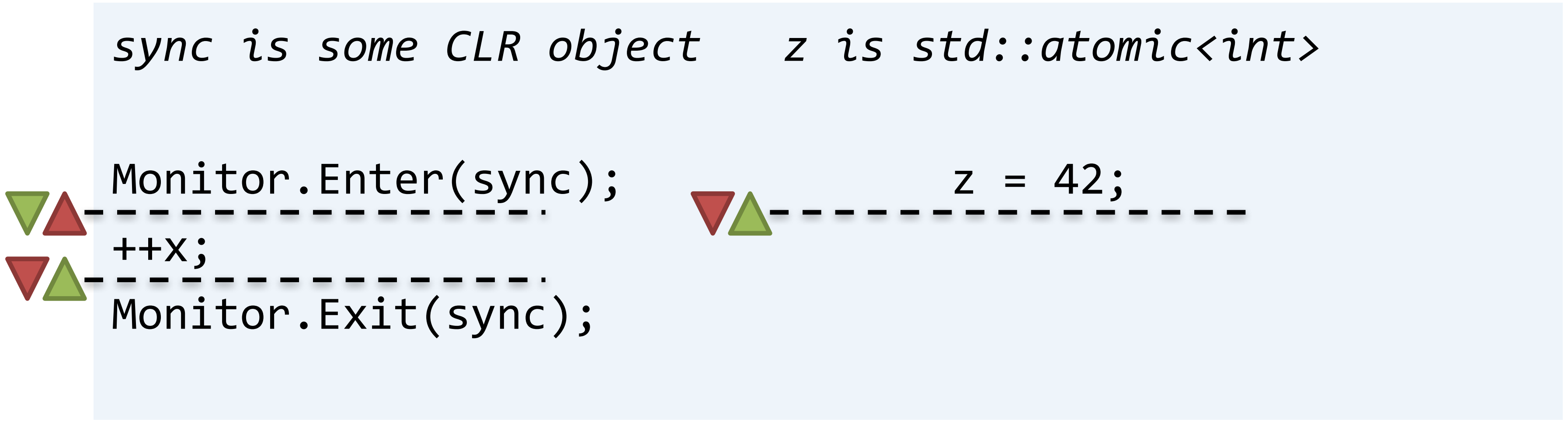
- In C#, `Thread.MemoryBarrier` is a full barrier
- `Volatile.Read`, `Volatile.Write`, and accessing `volatile` variables produce unidirectional barriers

```
Thread 1                       Thread 2


flag1 = 1;                     flag2 = 1;
-------------------            -------------------
Thread.MemoryBarrier();        Thread.MemoryBarrier();
if (flag2 == 0) {              if (flag1 == 0) {
  critical section               critical section
}                              }
else handle_contention();      else handle_contention();
```

Other Forms of Barriers

- Operations on synchronization mechanisms are memory barriers (usually unidirectional)
- Operations on `std::atomic` variables are memory barriers (usually unidirectional)



```
sync is some CLR object    z is std::atomic<int>


Monitor.Enter(sync);                z = 42;
------------------      ----------------------
++x;
------------------
Monitor.Exit(sync);
```

std::atomic

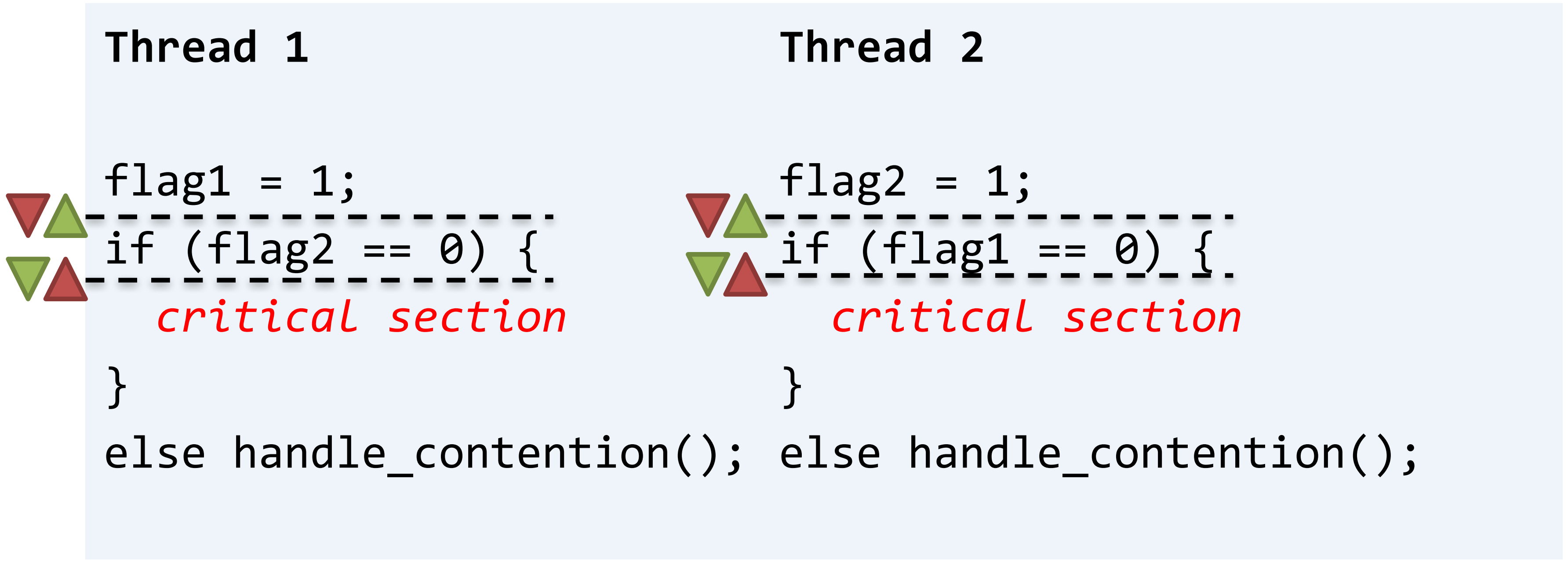
- Portable API for low-level memory operations
  - **Atomic**: no torn reads or torn writes
  - **Ordered**: acquire/release and additional memory ordering guarantees
- Suppose `value` and `done` are `std::atomics`:

```
Thread 1                                Thread 2


value = long_calculation();             if (done) {
-----------------                       -----------------
done = true;                                    do something with value
                                        }
```

Pitfall: Release Barrier Followed by Acquire Barrier

- Imagine `flag1` and `flag2` are `std::atomic<int>` or C# volatile variables
- **These instructions can still reorder!**



```
Thread 1                        Thread 2


flag1 = 1;                      flag2 = 1;
--------------------            --------------------
if (flag2 == 0) {               if (flag1 == 0) {
--------------------            --------------------
  critical section                critical section
}                               }
else handle_contention();       else handle_contention();
```

Bonus Examples

Thread-Safe Singleton: C++, BAD

```
static T* instance_ = nullptr;

static T* get_instance() {
  if (instance_ == nullptr) {
    instance_ = new T();
  }
  return instance_;
}
```

Thread-Safe Singleton: C++, STILL BAD

```
static T* instance_ = nullptr;
static std::mutex protector_;

static T* get_instance() {
  if (instance_ == nullptr) {
    protector_.lock();
    instance_ = new T();
    protector_.unlock();
  }
  return instance_;
}
```

Thread-Safe Singleton: C++, STILL BAD

```
static T* instance_ = nullptr;
static std::mutex protector_;

static T* get_instance() {
  if (instance_ == nullptr) {
    protector_.lock();
    if (instance_ == nullptr) {
      instance_ = new T();
    }
    protector_.unlock();
  }
  return instance_;
}
```

Thread-Safe Singleton: C++, STILL BAD

```
static T* instance_ = nullptr;
static std::mutex protector_;

static T* get_instance() {
  if (instance_ == nullptr) {
    std::lock_guard<std::mutex> lock{ protector_ };
    if (instance_ == nullptr) {
      instance_ = new T();
    }
  }
  return instance_;
}
```

Thread-Safe Singleton: C++, STILL BAD

```
static volatile T* instance_ = nullptr;
static std::mutex protector_;

static T* get_instance() {
  if (instance_ == nullptr) {
    std::lock_guard<std::mutex> lock{ protector_ };
    if (instance_ == nullptr) {
      instance_ = new T();
    }
  }
  return instance_;
}
```

Thread-Safe Singleton: C++, WHEW

```
static std::atomic<T*> instance_{ nullptr };
static std::mutex protector_;

static T* get_instance() {
  if (instance_ == nullptr) {
    std::lock_guard<std::mutex> lock{ protector_ };
    if (instance_ == nullptr) {
      instance_ = new T();
    }
  }
  return instance_;
}
```

Thread-Safe Singleton: C++

```
// Requires fully-conformant C++11 compiler
// Supported by VC++ since Visual Studio 2015

static T& get_instance() {
  static T instance;
  return instance;
}
```

One More Example…

```
struct SpinLock_DO_NOT_USE
{
  private bool _locked;
  public void Lock()
  {
    while (Interlocked.Exchange(ref _locked, true)) ;
  }
  public void Unlock()
  {
    _locked = false;
  }
}
```

Does `_locked` need to be volatile? Or, does this line require `Volatile.Write`?

“`volatile` Is Like `lock`”

- Myth: For small types like `int`s you don’t need a full-blown lock, just use `volatile`

- Reality: `protected` is like `delegate`

“Use `volatile` To Avoid Race Conditions”

- Myth: If you apply `volatile` to the right variables, you don’t need synchronization mechanisms and won’t have race conditions
- Reality: Use kerosene to put off a fire

Summary

- Here be dragons!
- If possible, try to hide behind someone else’s synchronization primitives

Thank You!

Sasha Goldshtein

@goldshtn