

# Multithreading for Gamedev Students

**Keith O'Connor**

3D Technical Lead  
Ubisoft Montreal

@keithoconor

- Who I am
  - PhD (Trinity College Dublin), Radical Entertainment (shipped Prototype 1 & 2), Ubisoft Montreal (shipped Watch\_Dogs & Far Cry 4)
- Who this is aimed at
  - Game programming students who don't necessarily come from a strict computer science background
  - Some points might be basic for CS students, but all are relevant to gamedev
- Slides available online at [fragmentbuffer.com](http://fragmentbuffer.com)

# Overview

- Hardware support
- Common game engine threading models
- Race conditions
- Synchronization primitives
- Atomics & lock-free
- Hazards

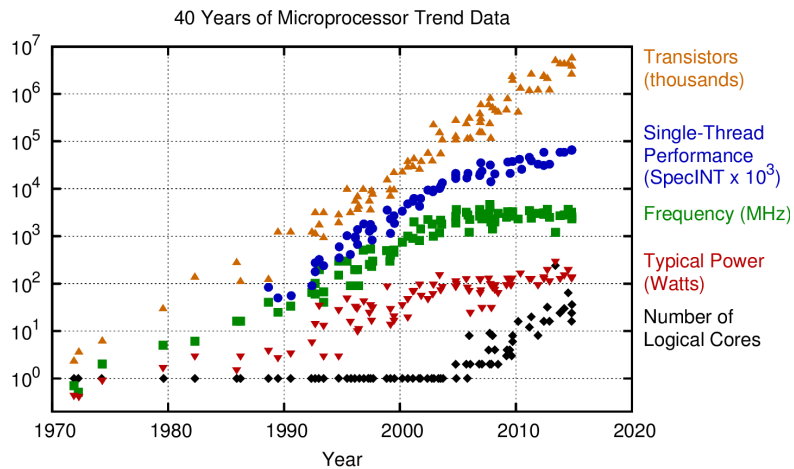
- Start at high level, finish in the basement
- Will also talk about potential hazards and give an idea of why multithreading is hard

# Overview

- Only an introduction
  - Giving a vocabulary
  - See references for further reading
  - Learn by doing, hair-pulling

- Way too big a topic for a single talk, each section could be its own series of talks

# Why multithreading?



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2015 by K. Rupp

- Hitting power & heat walls when going for increased frequency alone
- Go wide instead of fast
- Use available resources more efficiently
- Taken from <http://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/>

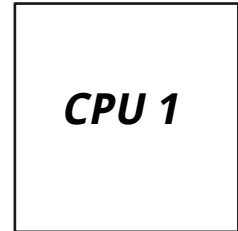
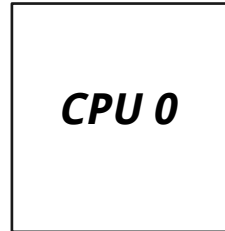
## *Hardware support*

- Before we use multithreading in games, we need to understand the various levels of hardware support that allow multiple instructions to be executed in parallel
- There are many more aspects to hardware multithreading than we'll look at here (processor pipelining, cache coherency protocols etc.)
- Again, going from high-level to low...

# Hardware support

## Multiple processors

- Expensive, high power consumption, latency between chips, cache coherency issues
- Generally restricted to high-end desktops & big iron

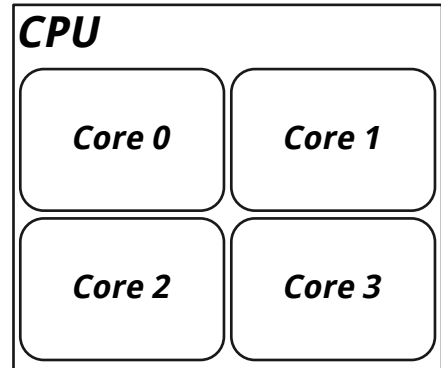


- At the highest level are multiple physical processors
- Uncommon for regular desktop PCs used for games, due to various cost & complexity factors

# Hardware support

## Multiple cores

- Multi-core allows for more efficient use of available hardware resources
- Cores might share L2/L3 caches, memory interface
- Most common setup for desktops and game consoles

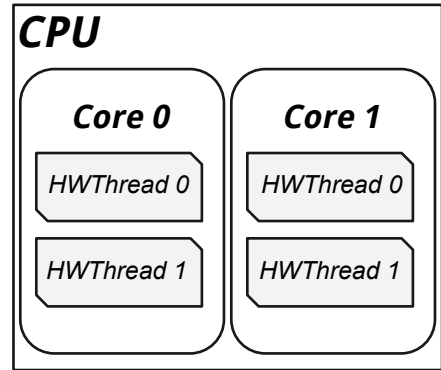


- A processor can have multiple cores
- The term 'core' refers to the package of ALUs, instruction pipeline, cache etc. that is used to do actual work (instruction processing)
- Cores share some processor resources, like cache & memory interface
- Shared resource usage can have a performance impact, must be taken into account when designing multithreaded systems
- Common current configurations are 2, 4 or even 6 cores per processor
- The vast majority of machines used for playing games are multicore single-processor systems with SMT...

# Hardware support

## Multiple hardware threads

- Simultaneous Multithreading (SMT)
  - “Hyper-threading” in Intel land
- Threads share core’s resources
  - Execution units, L1 cache etc.
- More efficient usage of core
  - Stalled threads don’t waste resources
- Typical 10%-20% faster vs. single hardware thread, but highly variable



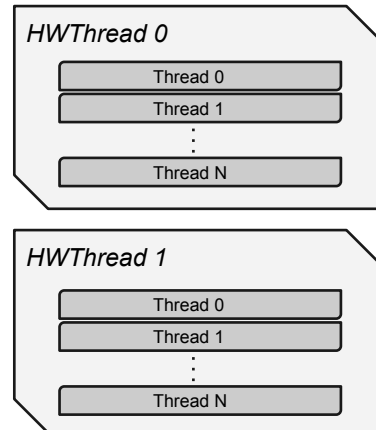
- A core can have multiple hardware threads
- The primary advantage of multiple hardware threads is the more efficient usage of the core’s hardware
- While one hardware thread is stalled, due to things like a cache miss or branch mispredict, the other thread(s) can use the core’s resources to do useful work
- Not as good as having separate cores, but gives better hardware utilization at relatively low cost - easy
- Most common current configuration is 2 hardware threads per core



# Hardware support

## Multiple software threads

- OS can create multiple processes
  - Games usually run as a single process
- A process can spawn multiple threads
  - Shared memory address space
- Threads can migrate between hwthreads
  - Or pinned to a specific hwthread with thread affinity



- Above the hardware level, a hardware thread can have multiple software threads
- Almost all games run as a single process, so all the game's threads share the same memory address space
- Alongside software threads are fibers - lightweight & cooperatively scheduled, reducing the need for complex synchronization
- Fibers outside the scope of this talk, but check out Christian Gyrling's excellent GDC talk on the use of fibers at Naughty Dog (see references at end)

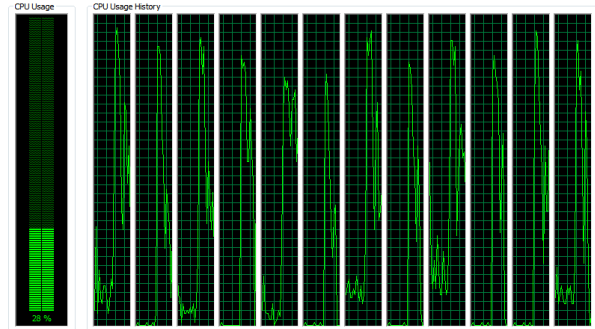
# *Hardware examples*

Let's look at a few configurations of multithreaded hardware that might be used for games

# Hardware examples

## Intel Xeon E5-1650

- 6 hyper-threaded cores
  - 12 'logical processors'
- L1 & L2 per-core
- Shared L3

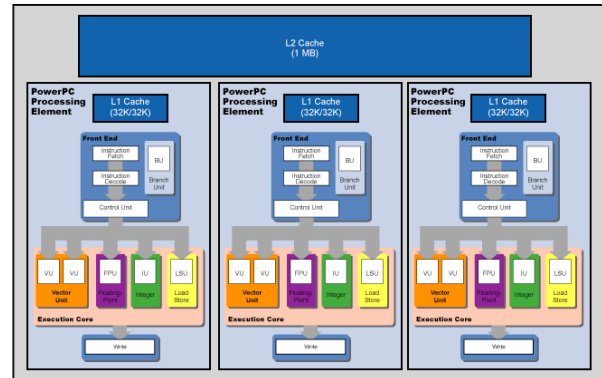


- My work machine
- Pretty standard architecture, although 12 hardware threads is higher than an average gaming desktop machine

# Hardware examples

## Xbox 360

- IBM Xenon CPU
  - PowerPC
- 3-core SMT
  - 6 hardware threads
- L1 per core, shared L2

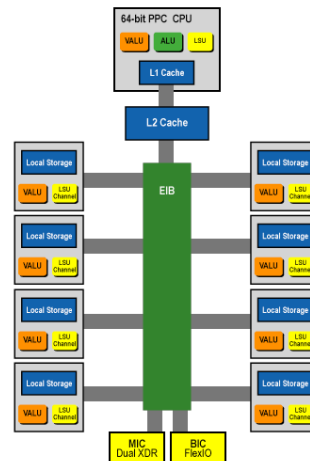


- Nice thing about working on consoles - fixed hardware setup, can tune for best performance
- Take care what runs on which hardware thread, taking into account which threads shared cores
- Try to match cache/processing-intensive threads on one hwthread with less intensive ones (not always possible)

# Hardware examples

## PlayStation 3

- Cell processor
  - Single SMT core, 2 hardware threads
- 1 PPU
  - Single core, high-speed 'local store' instead of RAM access
- 8 SPUs:
  - Single core, high-speed 'local store' instead of RAM access

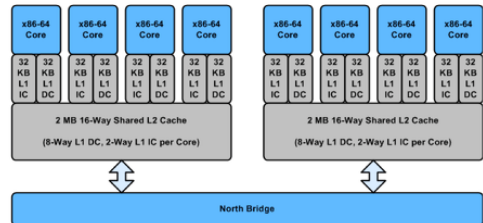


- The ugly duckling (or beautiful swan, depending on who you ask)
- Significantly different to regular PCs & 360, requiring lots of dedicated threading code (DMA data transfers, synchronization) to take advantage of all available cores
- 8 SPUs in chip design but 6 available for game use
- Intended usage was PPU setting up work and SPUs doing the heavy lifting (didn't always work out that way due to complexity)

# Hardware examples

## PlayStation 4 / Xbox One

- AMD Jaguar architecture
- 2 quad-core 'modules'
  - 8 hardware threads
  - L1 cache per core
  - L2 cache shared by all cores in module

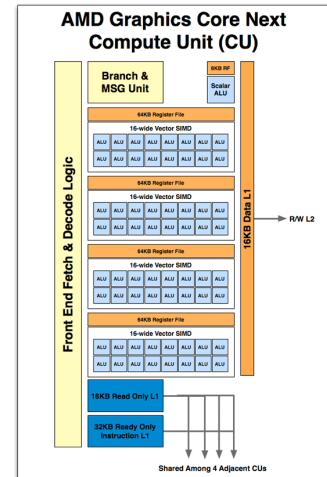


- Both chips based on the AMD Jaguar architecture - homogeneity across platforms is much nicer than previous gen
- 8 hardware threads with individual L1, split into two modules with L2 shared per module
- Cross-module cache access expensive

# Hardware examples

## AMD GCN GPUs

- Used by both PS4 & Xbox One
  - 18 & 12 compute units respectively
- Rendering is inherently parallel
  - Hardware exploits this to achieve high speed & throughput
- Extensible to non-graphics workloads
  - Compute & async compute



- And of course, we can't talk about parallel hardware in games without mentioning the Graphics Processing Unit
- AMD's GCN architecture used by both PS4 & Xbox One
- Graphics-specific processing means special hardware designs that allow for massive parallelism for that "embarrassingly parallel" domain (ie. non-branchy vertices, pixels), leading to teraflop-level processors
- ALUs all execute same instruction on different inputs (eg. 64 different pixels)
- Granularity means multiple workloads may be in flight at any one time - CUs can perform async compute while graphics work is stalled
- Massive area in itself - we'll concentrate on CPU parallelism

# *Multithreading in games*

- Now let's see how that multithreading hardware is put to use in game engine systems
- We'll then have a look at the low-level constructs used to coordinate data transfer



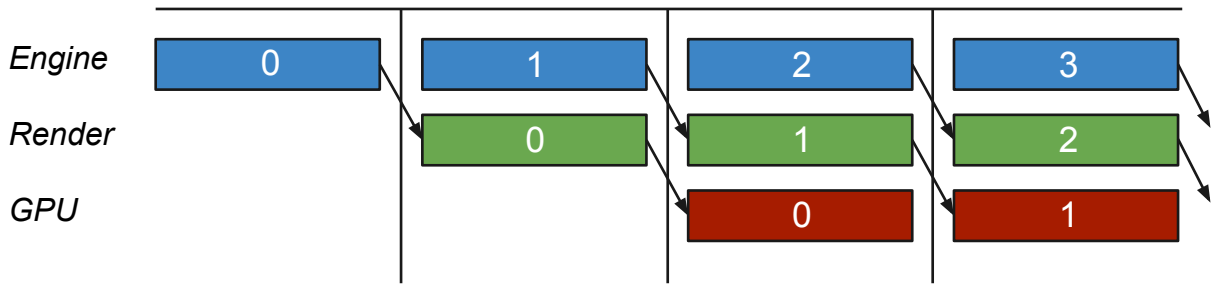
# Multithreading in games

- 30/60fps targets
  - Must use all available resources
- Many interacting systems
  - Restricted set of shared data
- Some common threading models

- High-performance games need to use all available processing power to reach target framerates
- While there are many interacting systems at work every frame, a lot of the work can be done in isolation with a restricted set of data needing to be shared
- There are a few common threading models in games, each suitable for different situations

# Multithreading in games

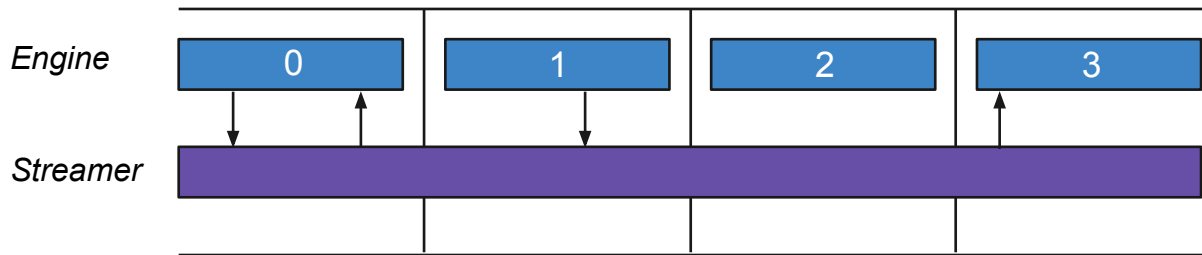
## Pipelining



- One thread does work, hands results to another thread to process resulting data in different way
- Example: Engine/render/GPU split - at the end of each frame, each thread hands off the work it has done to the next thread
- Each frame produces a frame on-screen, but with a few frames of latency

# Multithreading in games

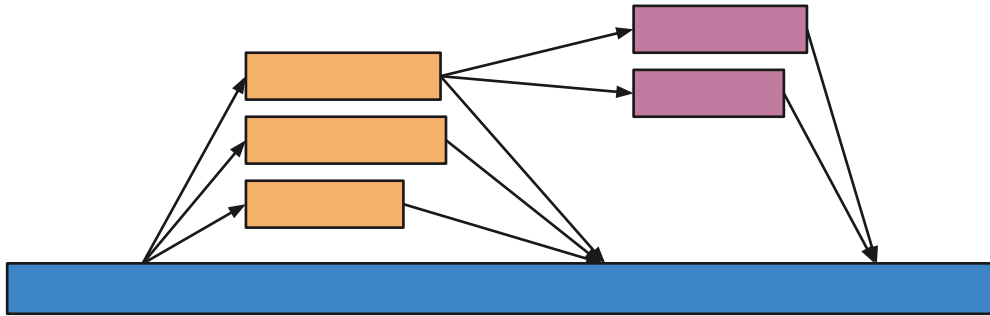
## Parallel threads



- Separate threads running concurrently, processing separate data but maintaining communication
- Example: streaming, audio
- Threads can be put to sleep if no useful work to do, woken by other threads
- When asleep, h/wthread can be used for other work - managed with thread priorities

# Multithreading in games

Job scheduler



- One thread managing & distributing work for other worker threads, collecting results afterwards
- Examples: particles, visibility system, draw call execution
- Common setup for processing data as quickly as possible, when results are needed for main thread to continue
- How do we manage the data & communication between threads?

# Multithreading is easy!...

- Good news! Multithreading is easy....
- Just run two independent bits of code on different threads.

# Multithreading is easy!...

... it's sharing data that makes it complicated.

- At the core of multithreaded programming is controlling access to shared data
- This means making sure data is accessed by one piece of code at a time, in the right order, while still staying performant.
- When you don't control this access properly, you get a race condition...

## ***Race conditions***

- Sometimes called a *data race*, It's named because the threads are basically racing each other to read and/or write the shared data. You don't know what thread is going to win that race, so you don't know what the outcome is going to be.

# Race conditions

```
int x = 0;
void threadFunc()
{
    for (int i = 0; i < 1000000; i++)
    {
        x++;
    }
}
// ...
std::thread t1(threadFunc);
std::thread t2(threadFunc);
```

Output?

- Here's a contrived but effective example
- Two threads spawned, both looping a million times and incrementing a variable
- The desired result is 2,000,000, but obviously that's not what's going to happen...



# Race conditions

```
int x = 0;
void threadFunc()
{
    for (int i = 0; i < 1000000; i++)
    {
        x++;
    }
}
// ...
std::thread t1(threadFunc);
std::thread t2(threadFunc);
```

1123412  
1539234  
1820120  
1738201  
1343722  
1259859  
...

- This is a perfect example of a race condition - every run produces wildly different results.
- We can in theory get any result from 1 to the full 2,000,000.

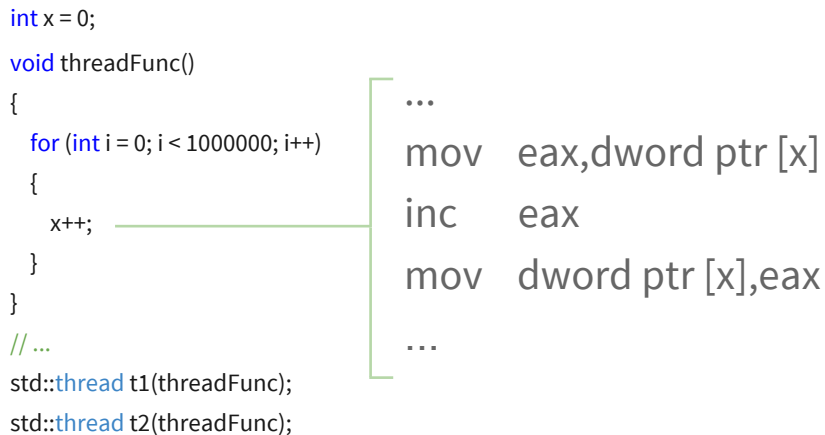
# Race conditions

- The system's output is dependent on timing
  - Timing influenced by many things
- Undefined behaviour - all bets are off
  - Random = unpredictable = wrong results
- Debugging nightmare

- With a race condition present, the exact same code when run multiple times can give completely different results due to any number of outside factors (OS thread scheduling, network traffic, disk I/O, cache usage, memory bandwidth)
- A race produces "undefined behaviour" - the C++ standard's phrase meaning "various bad things could happen, and it'll be your fault"
- A bug that manifests itself randomly is very difficult to track, difficult to diagnose, and being timing-dependent means it can stay hidden right up until you decide to try and ship your game
- That's a good reason to test in many different configurations (Debug, Release, Final etc.) throughout development - timings will differ wildly, and different issues will manifest themselves.

# Race conditions

```
int x = 0;
void threadFunc()
{
    for (int i = 0; i < 1000000; i++)
    {
        x++;
    }
}
// ...
std::thread t1(threadFunc);
std::thread t2(threadFunc);
```



```
...
mov  eax,dword ptr [x]
inc  eax
mov  dword ptr [x],eax
...
```

- To see why we got a race condition, let's look at the disassembly.
- The action we care about is happening on the increment itself, which compiles down to three instructions (with compiler optimizations disabled, for demonstration purposes)

# Race conditions

```
int x = 0;
void threadFunc()
{
    for (int i = 0; i < 1000000; i++)
    {
        x++;
    }
}
// ...
std::thread t1(threadFunc);
std::thread t2(threadFunc);
```

...

**mov eax,dword ptr [x]**

inc eax

mov dword ptr [x],eax

...

- We load the value of x from memory into the register eax...

# Race conditions

```
int x = 0;
void threadFunc()
{
    for (int i = 0; i < 1000000; i++)
    {
        x++;
    }
}
// ...
std::thread t1(threadFunc);
std::thread t2(threadFunc);
```

...

mov eax,dword ptr [x]

**inc eax**

mov dword ptr [x],eax

...

- ...increment the value in that register...

# Race conditions

```
int x = 0;
void threadFunc()
{
    for (int i = 0; i < 1000000; i++)
    {
        x++;
    }
}
// ...
std::thread t1(threadFunc);
std::thread t2(threadFunc);
```

...

mov eax,dword ptr [x]

inc eax

**mov dword ptr [x],eax**

...

- ...and then store the result from the register back to memory.

# Race conditions

```
int x = 0;
void threadFunc()
{
    for (int i = 0; i < 1000000; i++)
    {
        x++;
    }
}
// ...
std::thread t1(threadFunc);
std::thread t2(threadFunc);
```

```
...
mov  eax, dword ptr [x]
inc  eax
mov  dword ptr [x], eax
...
```

- A very important thing to remember about threads is that after every instruction, anything can happen
- The thread could stall for a millisecond, or go to sleep for an hour

# Race conditions

```
int x = 0;
void threadFunc()
{
  for (int i = 0; i < 1000000; i++)
  {
    x++;
  }
}
// ...
std::thread t1(threadFunc);
std::thread t2(threadFunc);
```

...

mov eax,dword ptr [x] ← [x] updated by t2

inc eax

mov dword ptr [x],eax → STOMP!

...

- So what happens when another thread updates the memory address [x] between the first two instructions?
- That other update gets stomped when the third instruction is executed!



# *Synchronization primitives*

- We can protect from data races like these with various established synchronization primitives
- I won't go in-depth on any of these, because there is plenty of literature out there that describes their syntax & use in much better detail than I could here

# Synchronization

## Spinlock

- Spin in a tight loop trying to acquire lock
  - Usually via atomic variable - see next section
- Can cause problems
  - CPU & memory bandwidth usage
- Lightweight when used correctly

- The most straightforward thing to do is a spinlock - basically spin in a loop and repeatedly trying to acquire a lock (eg. an atomic variable used as a flag) until it succeeds
- Needs to make sure the correct memory barrier is being used, either explicitly or via an atomic with correct ordering
- Can be effective when the spin is only for a short amount of time, as it avoids the overhead of context switches or thread scheduling

# Synchronization

## Mutex

- Lock/unlock pair
- Protects critical section of code, provides single-threaded access (mutual exclusion)

- A mutex is probably the most commonly used sync primitive, protecting a 'critical section' of code that contains accesses to shared data
  - Not to be confused with the unfortunately-named Windows CRITICAL\_SECTION object, which is itself a user-space mutex
  - Support provided in C++11 standard library
- Only one thread can lock the mutex at any time, so any data access done inside the mutex is guaranteed to be thread-safe
- General OS mutexes are kernel objects, so can synchronize between processes
- But games run in a single process, so don't want to pay the (sometimes heavy) cost for the OS context switch needed to handle these. We prefer to use lightweight mutexes like CRITICAL\_SECTION and other OS-specific primitives that only offer in-process synchronization.
- A good example of mutex use in game code is in a memory allocator, where only a single piece of code can be allowed to allocate memory at any one time

# Synchronization

## Semaphore

- Maintains internal counter
- Wait (decrement) & Signal (increment) operations
  - $\leq 0$  thread sleeps
  - $> 0$  waiting threads proceed
- Signaling can wake up threads
- Used for signaling between threads
  - Or controlling the number of threads that can perform a task

- Where mutexes are used for protecting resources, semaphores are used for signaling between threads
- A binary semaphore can be similar to a mutex, but without the concept of ownership (locked mutex must be unlocked by the thread that locked it)
- Typical example: producer/consumer queue. Producers signal, consumers wait.
- Game use example: signaling the render thread that the engine thread has finished its frame

# Synchronization

## Condition variables

- Threads wait until condition is met
- Monitor: mutex + condition variable
- Platform-specific Events used in games

- Allows a thread to sleep until a specific condition has been filled
- A 'monitor' is a construct where a condition variable is used to wake a single thread, and that thread then owns the monitor's mutex in order to do some work on a critical section of code
- Game platforms often provide auto-resetting events that provide similar functionality - thread notifies the event when a condition has changed, waking the waiting thread
- Another inclusion in C++11's multithreading support

# Synchronization

## GPU Fences

- For CPU  $\leftrightarrow$  GPU interaction
  - Knowing when shared data has been produced or consumed
- Platform-specific APIs
  - Core in DX12 & OpenGL since 3.2

- GPU fences are often used in 3D engines to allow the CPU to know when the GPU is finished using a certain resource, or vice versa
- Eg. compute shader being used to offload some CPU work to the GPU

# *Memory ordering*

- There are unseen forces at work which can make life much more complicated than simple data races from simultaneous access...

# Memory ordering

```
// global  
int data = 0;  
int readyFlag = 0;
```

```
// thread A  
data = 32;  
readyFlag = 1;
```

```
// thread B  
if(readyFlag == 1)  
{  
    Output( data );  
}
```

Output?

- Pretty straightforward code
- One thread writes a piece of data and then sets a flag to 1 to signal the other thread to do some work
- If the other thread sees the flag set, it prints out the data
- What output will it give?



# Memory ordering

```
// global  
int data = 0;  
int readyFlag = 0;
```

```
// thread A  
data = 32;  
readyFlag = 1;
```

```
// thread B  
if(readyFlag == 1)  
{  
    Output( data );  
}
```

*0 or 32!*

- Depending on the hardware and compiler, this code can easily produce either 0 or 32, contrary to expectations and common sense
- Even though it looks like a bug, it's happening because both the compiler and the CPU are busy behind the scenes performing various optimizations for you

# Memory ordering

- The compiler can reorder instructions
- The CPU can reorder instructions
- The CPU can reorder memory accesses

- The compiler can reorder instructions, eg. to hide stalls or perform various optimizations such as using registers to store intermediate results
- The CPU can reorder instructions, eg. speculative branch prediction, executing instructions before the branch test in order to avoid stalls & pipeline bubbles
- The CPU can reorder memory accesses, eg. using store buffers or write combining to improve the speed of memory writes & caching
- These optimizations are all very important for achieving high performance
- Any optimization is fair game for the compiler & CPU, as long as they adhere to their memory models

# Memory ordering

- Memory models
  - Determine what reads & writes can be reordered relative to others
- Hardware & software
  - Processor has one memory model
  - Language may have another

- For single thread, the effects of each memory access is seen in the order they're written - otherwise it would be impossible to reason about the code
- But behind the scenes the compiler or processor may decide to reorder them for better performance
- Other threads viewing the accesses may see unrelated reads & writes happen in a different order
- The rules of these reorderings - which accesses can be reordered - follow the memory model adhered to by the processor and the language

# Memory ordering

Type	Alpha	ARMv7	POWER	SPARC PSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y			Y		Y	
Loads reordered after stores	Y	Y	Y			Y		Y	
Stores reordered after stores	Y	Y	Y	Y		Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y

- Here we can see the effects of the memory model used by various processors [taken from the Wikipedia article on memory ordering]
- Gives an extra dimension to race problems - what works correctly on one processor might not work on another

# Memory ordering

- Sequential consistency
  - WYSIWYG for memory accesses
  - No apparent reordering
  - Subsequent limit on possible optimizations
  - Use unless performance dictates otherwise
- There are various memory models which allow different levels of reordering, but the most straightforward memory ordering is Sequential Consistency
- This basically removes reordering so that reads & writes are seen to happen in the order they were written, even when viewed from outside that thread
- Code still isn't necessarily executed in program order, but the important thing is that the memory effects are indistinguishable
- Makes it possible to easily reason about cause and effect when writing multithreaded code
- Limits the optimizations possible by the compiler & CPU due to the extra ordering restrictions

# Memory ordering

## Memory barriers

- Used for enforcing memory ordering on compiler & CPU
- Implicit in certain functions
  - Eg. `std::atomic<>` operations with ordering other than `memory_order_relaxed`
- Explicit acquire & release fences

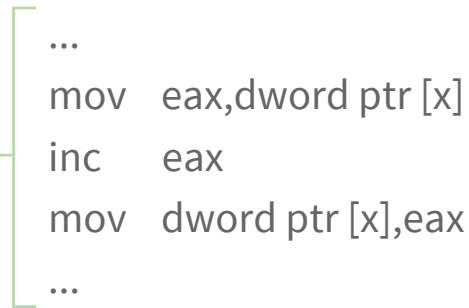
- The C++ *volatile* keyword can prevent compiler reordering, but not necessarily CPU reordering (compiler-dependent)
- When dealing with a memory model that has less than full sequential consistency, memory barriers (or fences) must be used to impose ordering
- Implied by using certain instructions like the C++11 atomic types with default ordering (`memory_order_seq_cst` - sequential consistency)
- For high-performance lock-free programming, explicit fences can be specified to impose the minimum necessary ordering - see references for more details (in particular Jeff Preshing's blog)
- In practice for game engines, we only really need to worry about memory models when writing lock-free code, and even then only when we really need highest performance possible - given the complexity and potential for bugs, this is relatively rare and only used by experienced engine programmers. But as a game programmer, it's always important to be aware of what's going on under the hood.

## ***Atomics & lock-free***

- Sometimes locks can be too heavyweight, or introduce problems with scalability and contention. In these cases we can turn to atomics operations.
- An atomic operation is an indivisible one; hardware-level implementations ensure that the results are either seen completely or not at all - never partially

# Atomics & lock-free

```
int x = 0;
void threadFunc()
{
    for (int i = 0; i < 1000000; i++)
    {
        x++;
    }
}
// ...
std::thread t1(threadFunc);
std::thread t2(threadFunc);
```



The diagram illustrates the decomposition of the `x++` operation into three assembly instructions:

- `mov eax, dword ptr [x]`
- `inc eax`
- `mov dword ptr [x], eax`

- Let's look back at our first race condition
- The problem here was that the increment is split up into its constituent read/modify/write operations
- A simple change to the first line is needed to make this whole operation atomic



# Atomics & lock-free

```
std::atomic<int> x = 0;
```

```
void threadFunc()
```

```
{
```

```
  for (int i = 0; i < 1000000; i++)
```

```
  {
```

```
    x++;
```

```
  }
```

```
}
```

```
// ...
```

```
std::thread t1(threadFunc);
```

```
std::thread t2(threadFunc);
```

```
...
```

```
mov     eax, 1
```

```
lock xadd dword ptr [x],eax
```

```
...
```

- This causes the compiler to emit a special instruction that ensures the atomicity of the operation
- C++11 is the first version of C++ to include atomics as part of its standard library
- Atomic operations not only prevent any other thread from interfering with the memory modification, but they also prevent against torn writes - ie. a type that needs to be written in two or more parts (eg. writing a 64-bit type on a 32-bit ISA)

# Atomics & lock-free

Variety of atomic operations available

- Add/subtract
- Exchange
- ...
- Compare & exchange
  - CAS loops

- Various atomic operations are available on different platforms, even bitwise operations
- Exchange: atomically assign a new value to the variable, returning the old value
- Compare & exchange: exchange, but only if the current value is equal to a given value
- CAS (Compare And Swap) loops employ compare & exchange operations to make a whole group of operations atomic

# Atomics & lock-free

```
while( true )
{
    int oldFoo = m_foo;
    int newFoo = oldFoo;
    DoStuff( &newFoo );
    if( CompareExchange(&m_foo, newFoo, oldFoo) == oldFoo )
    {
        break;
    }
}
```

- Here's an example of a CAS loop, in pseudo-C
- *m\_foo* is a variable that may be accessed by multiple threads. Shown here as an int here for simplicity, but commonly a pointer when we're modifying whole objects
- DoStuff() can be complex and doesn't need to be atomic, as it's only working on a local variable that isn't shared data
- CompareExchange() does an atomic compare & exchange, executing [*m\_foo* = *newFoo*] only if [*m\_foo* == *oldFoo*] and returning the old value of *foo*
- The actual compare & exchange is the only part that needs to be atomic, and the exchange only happens if *foo* hasn't changed
- If it has changed, we loop again and redo the operations with the new updated value
- DANGER! Can suffer from the ABA problem, depending on the use case; see hazards section at the end

# Atomics & lock-free

## Example: thread-safe linear allocator

```
void* Allocate(int size)
{
    void* mem = AtomicAdd(&m_heapPointer, size);
    return ( (mem + size) > m_endOfHeap ) ? nullptr : mem;
}
void Clear()
{
    AtomicAssign(&m_heapPointer, m_startOfHeap);
}
```

- A common example use of atomics in game code is making a simple thread-safe linear allocator
- Linear allocators are used for cheap, temporary memory allocations that can be thrown away after a time, usually at the end of the frame
- Used to avoid things like fragmentation, overhead of free block management, allocator lock contention, and the other problems commonly associated with generalized memory allocation
- A simple atomic add of the heap pointer is all that's needed to make an allocation, since the atomic operation will return the value before the add
- Similarly, clearing the allocator at the beginning of the frame is a simple atomic assignment

# Atomics & lock-free

## Lock-free programming

- Implementing a multithreaded algorithm without locks, and without blocking
- No thread can stop global progression by being interrupted

- All of these together - understanding reordering, using memory barriers, and atomic operations - can be used to make an algorithm fast and lock-free

# Atomics & lock-free

## Why lock-free?

- Freedom from lock contention
- Scalability
- Performance\*
  - Uncontented locks can be very performant

- The main reason to use lock-free is when locks have already proven to be a problem
  - A heavily-contented lock can degrade performance to that of a single-threaded program, or worse
  - A thread being preempted while holding a lock can block global progression
- Performance is often cited as a good reason to go lock-free, but that's not always necessarily the case
  - Uncontented locks can be very fast if used correctly, and much simpler than lock-free
  - Atomic operations can be >10x as slow as their non-atomic equivalents
  - Additional details (such as cacheline contention) can slow things down even further, depending on the implementation
  - As with every optimization, always profile before & after

# Atomics & lock-free

## Why not lock-free?

- Complexity
- Complexity
- Complexity

- As you've hopefully gathered by now, lock-free programming can be extremely complex, and requires a thorough understanding of the hardware and careful design
- When first faced with multithreaded bottlenecks related to shared data, the first thing to do is try to reduce the amount of data sharing
  - Fewer & shorter locks, per-thread memory instead of shared, fewer sync points
  - Always profile with real-world data - algorithms can perform very differently with high vs. low lock contention
- Only go lock-free if nothing else works, and even then do so with extreme care
- Things can go wrong in all sorts of subtle, non-obvious ways that are very hard to debug

# ***Hazards***

- Finally, let's look at some of the varied and horrible ways things can go wrong in a multithreaded environment



# Hazards

- Deadlocks
  - Two locks acquired, but in different orders
  - One thread locks A and waits for B
  - Another locks B and waits for A

- To avoid deadlocks, locks should always be acquired in the same order

# Hazards

- Livelocks
  - Multiple threads making local progress
  - Activity of each thread causes others to repeatedly not make global progress
  - Classic analogy: two people in a corridor

# Hazards

- Priority inversion
  - The low-priority thread takes a lock which the high-priority one needs, and then goes to sleep because of its priority
  - System performance ends up being dictated by the low-priority thread instead of higher ones
- Priority inversion can be worked around by randomly boosting thread priorities (Windows does this), or keeping track of the owners of locks in order to detect such a case

# Hazards

- False sharing
  - Multiple threads modifying memory in the same cacheline
  - Causes constant cache invalidation & unnecessary memory traffic
  - Can significantly impact performance
- “False” sharing because the data isn’t actually shared, but its locality and the way cache works causes writes by one thread to invalidate the cache on others
- Can be worked around by ensuring that if multiple threads that are working on the same object, they are accessing separate parts of it
- Needs to be taken into account when designing the algorithm & datastructures in the first place

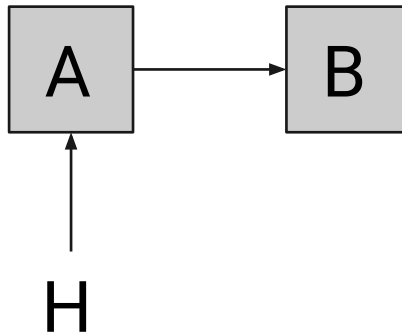
# Hazards

## ABA problem

- The final hazard we'll look at is the ABA problem
- This problem is one that can easily crop up when using CAS loops, or other similar multithreaded constructs that rely on checking a previous value
- Best demonstrated by the example of adding an object onto a linked list

# Hazards

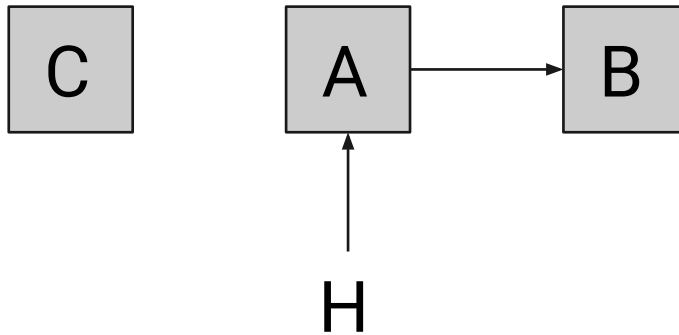
## ABA problem



- Let's say we have two nodes of a linked list, A and B, and the Head pointer pointing to A
- The list can be accessed from multiple threads, so we want to make sure it can never be in an inconsistent state

# Hazards

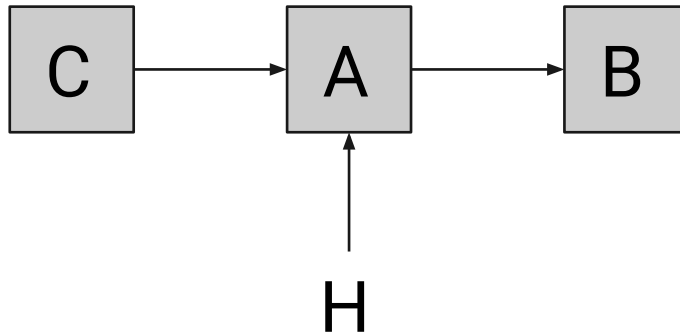
ABA problem



- We want to add a new node C atomically using a CAS loop

# Hazards

ABA problem

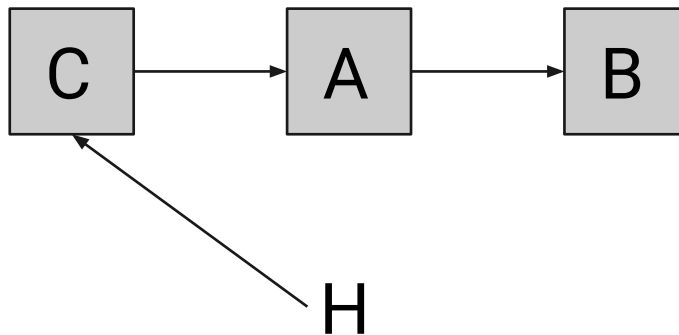


- We create the new node and point it at A
- This doesn't need to be atomic, as we haven't altered the state of the list itself yet - only a newly allocated node local to one thread



# Hazards

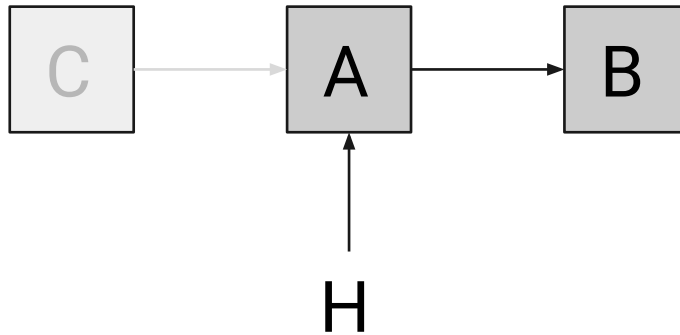
## ABA problem



- Finally, in an atomic operation we would use CompareExchange to change the Head to point at C
- If the state of the list had changed, the exchange wouldn't happen and we would start again
- We have our new head node, and everything happened atomically we we're good.
- But there's a problem - let's roll back a step and see where this could go wrong

# Hazards

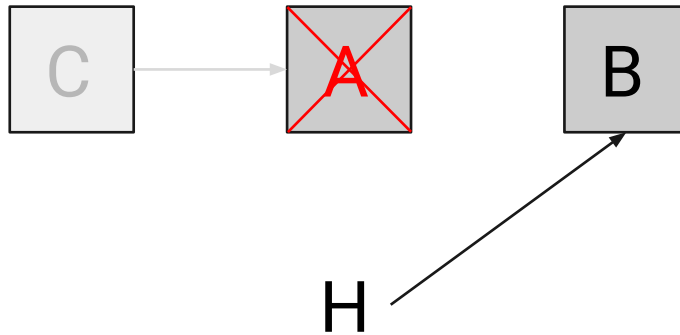
ABA problem



- Let's say, before performing the last step of exchanging the head pointer, the thread gets preempted by the OS and goes to sleep

# Hazards

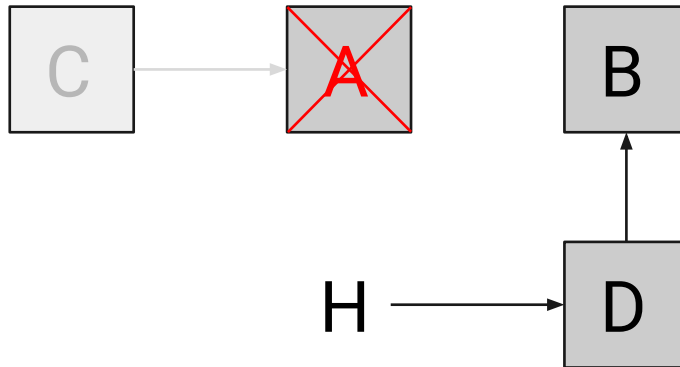
## ABA problem



- During that time, another thread comes in and decides to delete A from the list.
- It deletes A, and points the head at B.
- Since C is still local to the other thread, it doesn't know or care about it.

# Hazards

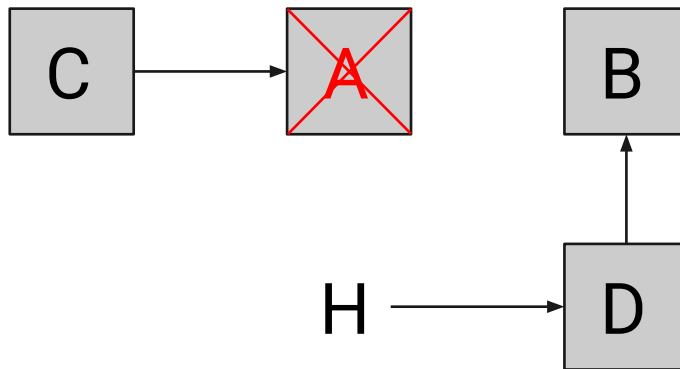
## ABA problem



- It then allocates a new node D, and adds it to the list
- However - and this is the important bit - the memory allocator reuses the memory for the recently-deleted A for the new node, so D has the same address as A did

# Hazards

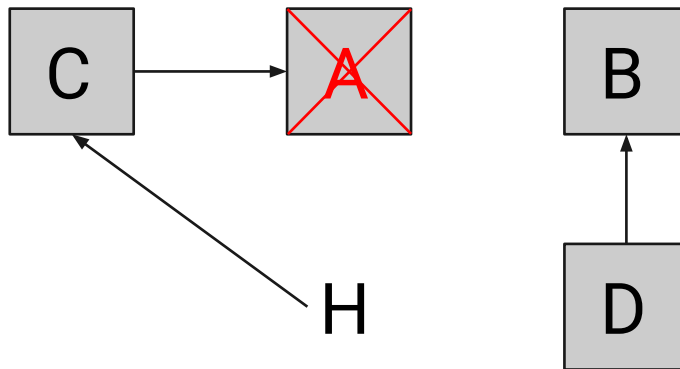
ABA problem



- The first thread now wakes up again
- It performs the CompareExchange, which succeeds as H still points to the same memory address as when it started

# Hazards

## ABA problem



- Our linked list is left in ruins!
- Even worse is the fact that we won't crash straight away - C effectively points at D since it's the same memory address - but it's a different object than it should be and this inconsistency will eventually cause problems that aren't immediately identifiable
- A common way to work around this is to use a unique identifier for each node (eg. a union of an array index & unique counter), and CAS loop on that instead of the raw pointer
- This is one example of the insidious problems that can creep into lock-free programming

***Final thoughts***

# Complexity

- A little knowledge is a dangerous thing
- Mistakes are easy to make, but hard to debug



# Complexity

- No such thing as unlikely
  - “One in a million”
  - 50 times a frame, 30fps... ~11 minutes
- Bad things will happen, *if you're lucky*

- One thing to be wary of in multithreading is the “edge case”
- Something that happens exceedingly rarely is your greatest enemy
- Because of the variable timing inherent in multithreading, catching these issues is very hard
- If you're lucky, crashes and instability will be frequent. Unlucky: once a week, in a particular build configuration, only for certain testers.

# Complexity

- Even simple things can cause problems

```
enum{ EValueA, EValueB };
```

```
//...
```

```
Assert( foo == EValueA || foo == EValueB );
```

- Here's a final example that I came across recently
- At first glance there's not much that can go wrong, but it started occasionally asserting
- My first thought was memory corruption or some other memory issue like alignment (the GPU was setting the value of *foo* via a fence), since the value could only be one of those two
- What further stumped me was that whenever the assert was reported, *foo* was indeed equal to EValueA
- I soon realized *foo* was being changed from EValueB to EValueA in the middle of the assert logic - after the first test but before the second
- This shows how things can easily go wrong when you let your guard down

# Debuggability

- Given all this complexity, plan ahead
- Always try to keep a single-threaded path alive
  - So you know if problems are logic or threading
  - Runtime-switchable if possible
- Sometimes just thinking is better than debugging

***Questions?***

# References

<http://preshing.com/20120612/an-introduction-to-lock-free-programming/>

The blog of Jeff Preshing (Technical Architect, Ubisoft Montreal), a goldmine for learning about lock-free programming, memory models, and many other related topics. Start with this post and then work your way through the whole blog.

<https://www.youtube.com/watch?v=X1T3IQ4N-3g>

Preshing's "How Ubisoft Develops Games for Multicore" talk from CppCon 2014

[https://msdn.microsoft.com/en-us/library/windows/desktop/ee418650\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ee418650(v=vs.85).aspx)

"Lockless Programming Considerations for Xbox 360 and Microsoft Windows" - Bruce Dawson. A great intro to multithreading & lock-free programming with concrete examples and explanations

# References

<https://www.youtube.com/watch?v=c1gO9aB9nbs>

<https://www.youtube.com/watch?v=CmxkPChOcvw>

“Lock-Free Programming (or, Juggling Razor Blades)”, a talk from Herb Sutter at CppCon 2014.

<http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/>

“atomic<> Weapons: The C++ Memory Model and Modern Hardware” - Another in-depth talk from Herb Sutter on atomics and C++11

<http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf>

“Memory Barriers: a Hardware View for Software Hackers” - Paul E. McKenney. A great low-level view of how caches work and why memory barriers are necessary. An appendix of the book “Is Parallel Programming Hard, And, If So, What Can You Do About It?": <https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html>

# References

<http://www.gdcvault.com/play/1022186/Parallelizing-the-Naughty-Dog-Engine>

“Parallelizing the Naughty Dog Engine Using Fibers” - GDC 2015 talk from Christian Gyrling

<https://msdn.microsoft.com/en-us/magazine/dn973015.aspx>

“What Every Programmer Should Know About Compiler Optimizations” - Hadi Brais, Microsoft. An interesting look at the optimization & instruction reordering that can be done by compilers

<http://www.gotw.ca/publications/concurrency-ddj.htm>

“The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software” - an oft-cited 2005 Dr. Dobbs article by Herb Sutter on how multithreaded is the way forward.

# References

<https://fgiesen.wordpress.com/2014/08/18/atomics-and-contention/>

"Atomic operations and contention" - Fabian "ryg" Giesen, RAD Game Tools. A good article on the actual cost of atomic operations. Another blog worth browsing in full for good low-level details.

<http://developer.amd.com/resources/documentation-articles/conference-presentations>

"The AMD GCN Architecture: A Crash Course" - Layla Mah, AMD. An in-depth look at the GCN architecture used by the GPUs of the PS4 and Xbox One.

<http://stackoverflow.com/questions/6319146/c11-introduced-a-standardized-memory-model-what-does-it-mean-and-how-is-it-g>

One of the better SO answers on the implications of C++11's memory model