ECE4893A/CS4803MPG: Multicore and GPU Programming for Video Games

Introduction to Multithreading

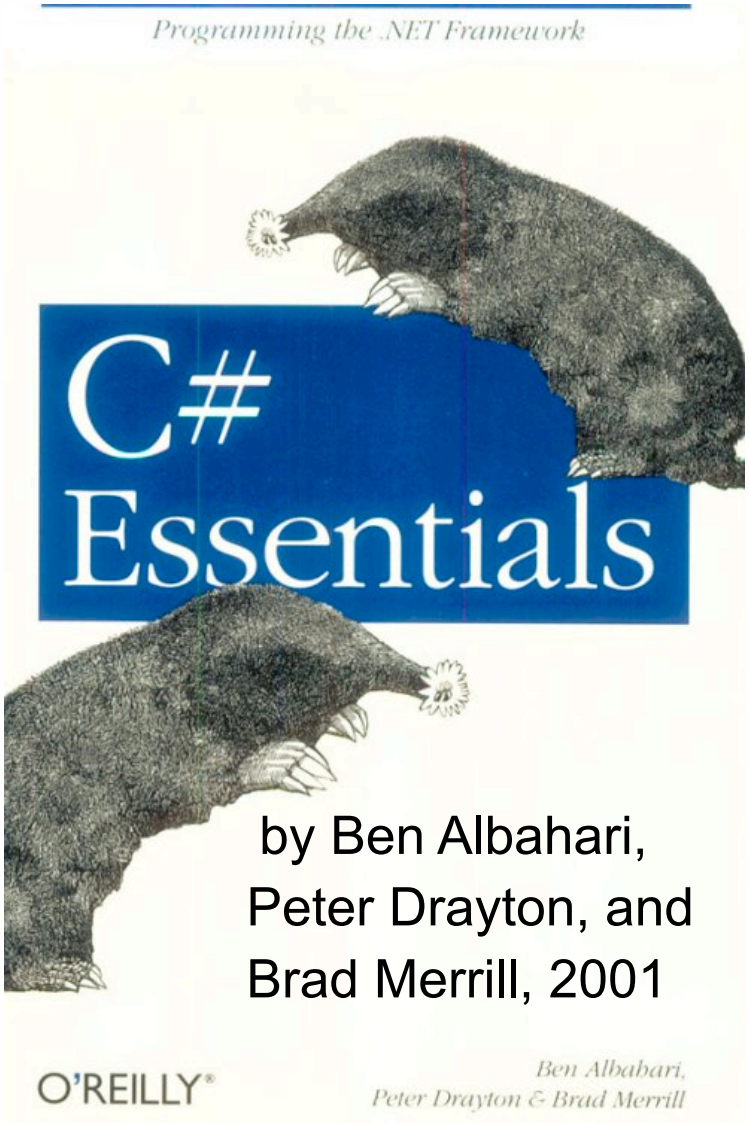
Prof. Aaron Lanterman

School of Electrical and Computer Engineering

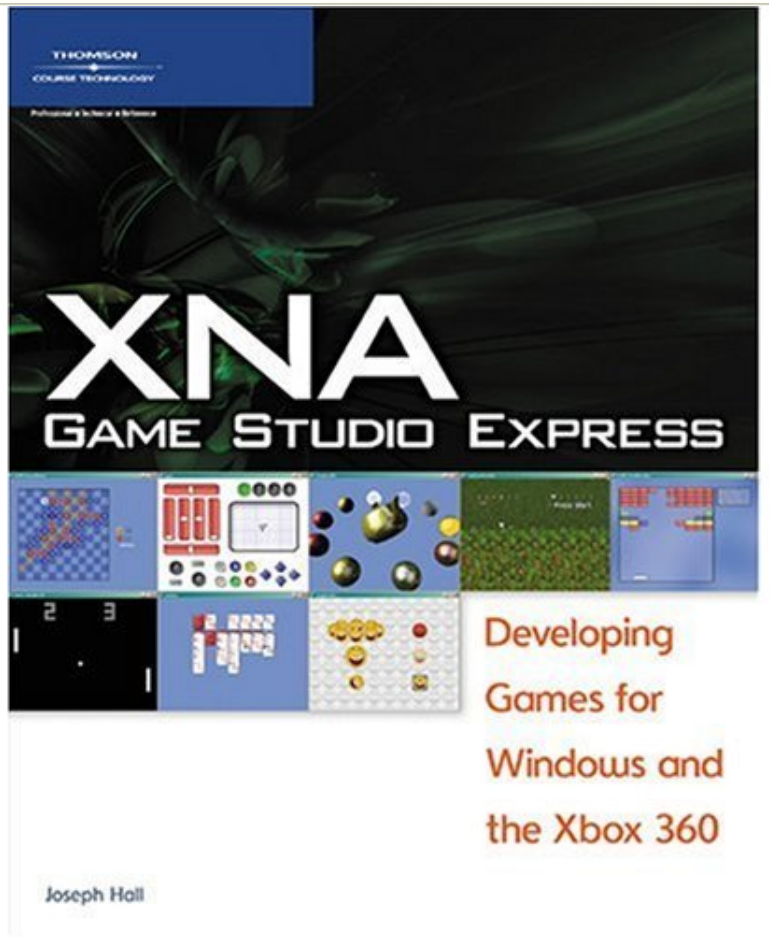
Georgia Institute of Technology

Georgia Institute of Technology

References (1)



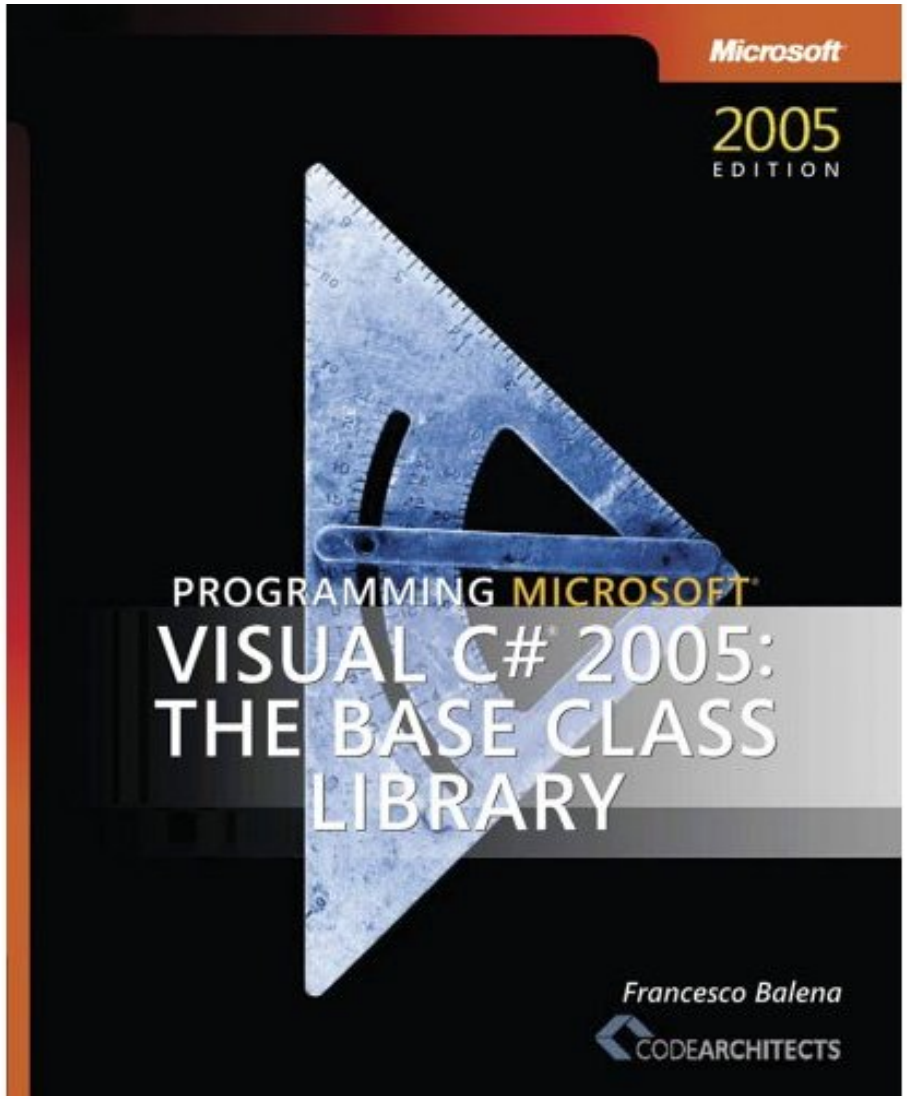
by Ben Albahari, Peter Drayton, and Brad Merrill, 2001



by Joseph Hall, 2008

Georgia Institute of Technology

References (2)



Francisco Balena

2006

Microsoft Press

References (3)

Tons of stuff from Microsoft's Bruce Dawson

Threading example

```
using System;
using System.Threading;
class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread(new ThreadStart(Go));
    t.Start();
    Go();
  }
  static void Go()
  {
    for (char c='a'; c <= 'z'; c++)
      Console.Write(c);
  }
}
```

static methods are part of the class, not particular instances

Example from "C# Essentials," pp. 107-108.

Georgia Institute of Technology

Threading example output

```
using System
using System.Threading;
class ThreadTest
{
  static void Main()
  {
     Thread t = new Thread(new ThreadStart(Go));
    t.Start();
    Go();
   }
  static void Go()
  {
      for (char c='a'; c <= 'z'; c++)
      Console.Write(c);
  }
}
```

Output:

```
abcdabcdefghijklmnopqrsefg
hjiklmnopqrstuvwxyztuvwxyz
```

Example from "C# Essentials," pp. 107-108.

Georgia Institute of Technology

Lock example

```
using System;
using System.Threading;
class LockTest {
  static void Main() {
    LockTest lt = new LockTest();
    Thread t = new Thread(new ThreadStart(lt.Go));
    t.Start();
    lt.Go();
  }
  void Go() {
    lock(this)
      for (char c='a'; c <= 'z'; c++)
        Console.Write(c);
  }
}
```

Example from "C# Essentials," p. 108

this references the current instance of the class (can't use **this** in static methods)

lock takes a reference type; if another thread has already acquired a lock, this thread halts until the other thread lets it go

Locks example output

```
using System;
using System.Threading;
class LockTest {
   static void Main() {
     LockTest lt = new LockTest();
     Thread t = new Thread(new ThreadStart(lt.Go));
    t.Start();
    lt.Go();
   }
  void Go() {
    lock(this)
      for (char c='a'; c <= 'z'; c++)
        Console.Write(c);
  }
}
```

Example from "C# Essentials," p. 108

Output:

```
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
```

Georgia Institute of Technology

Lock: behind the curtain

```
lock(expression)
{
  //mycode
}
```

is syntactic sugar for

```
System.Threading.Monitor.Enter(expression);
try {
  // mycode
}
finally {
  System.Threading.Monitor.Exit(expression);
}
```

From “C# Essentials,” pp. 108-109

Lock advice from MSDN

- "In general, avoid locking on a public type, or instances beyond your code's control…
  - **lock**(**this**) is a problem if the instance can be accessed publicly.
  - **lock**(**typeof**(MyType)) is a problem if MyType is publicly accessible.
  - **lock**("myLock") is a problem since any other code in the process using the same string, will share the same lock."

http://msdn.microsoft.com/en-us/library/c5kehkcz(VS.80).aspx

Lock advice from Rico Mariani

```
class MyClass {
   private static String myLock = “MyLock“;
   public void Foo() {
      lock(myLock) { ... }
    }
}
```

- “This is bad because string literals are normally interned, meaning that there is one instance of any given string literal for the entire program. The exact same object represents the literal…on all threads. So if someone else comes along and locks a literal named “MyLock” his literal will interfere with yours.
- Recommendation:

```
private static Object myLock = new Object();
```

http://blogs.msdn.com/ricom/archive/2003/12/06/41779.aspx

Don’t lock on value types

- Value types can be “boxed” to act as reference types…
- …but each lock construct will create a **different** box

Grrrrrrrrrrr!!!!!

- XNA on Xbox 360 uses Compact Framework, not full .NET like on Windows
- Compact Framework has a Monitor class (so can use locks), but it doesn't implement Pulse/Wait and their variations ☹
- Also missing Semaphores
- Not sure about "pro Xbox 360 development," i.e. C++ XDK

One Mutex

```
// This Mutex object must be accessible to all threads.
Mutex m = new Mutex();

public void WaitOneExample();
{
  // Attempt to enter the synchronized section,
  // but give up after 0.1 seconds
  if (m.WaitOne(100, false))
  {
    // Enter the synchronized section.
    …
    // Exit the synchronized section, and release the Mutex.
    m.ReleaseMutex();
  }
}
```

A mutex is called “signalled” if no thread currently owns it

From F. Balena, “Visual C# 2005: The Base Class Library,” p. 478.

Georgia Institute of Technology

Many Mutexes - WaitAny

```
static Mutex[] mutexes =
          { new Mutex(), new Mutex(), new Mutex() };

public void WaitAnyExample();
{
  // Wait until a resource becomes available.
  // (Returns the index of the available resource.)
  int mutexNdx = Mutex.WaitAny(mutexes);
  // Enter the synchronized section.
  // (This code should use only the
  // resource corresponding to mutexNdx.)
  …
  // Exit the synchronized section, and release the Mutex.
  mutexes[mutexNdx].ReleaseMutex();
}
```

From F. Balena, “Visual C# 2005: The Base Class Library,” p. 479.

Many Mutexes - WaitAll

```
Mutex.WaitAll(mutexes)
```

- Wait until all resources have been released

- Useful if you can't proceed until all the other threads are done

From F. Balena, "Visual C# 2005: The Base Class Library," pp. 480.

Naming a Mutex (available on Windows)

```
Mutex m = new Mutex(false,"mutexname");
```

- If a Mutex with that name already exists, caller gets a reference to it; otherwise a new Mutex is created
- Lets you share Mutex objects among different applications
  - Not too relevant to video game programming

From F. Balena, "Visual C# 2005: The Base Class Library," pp. 480.

Mutexes vs. Monitor locks

- Mutexes slower than locks (around 20 times slower!)
  - Monitor locks operating at the level of the CLR
  - Mutexes operate at the OS level
- Mutexes generally reserved for interprocess communications (vs. interthread)

Info from B. Dawson, “Coding For Multiple Cores on Xbox 360 and Microsoft Windows,” http://msdn2.microsoft.com/en-us/library/bb204834.aspx

Thread safety

- Some .NET objects are thread-safe
- Some aren't
- Some .NET objects have some method that are thread safe and some that aren't
- Check the documentation
- If using on Xbox 360, be careful to note .NET & Compact .NET differences

Info from F. Balena, "Visual C# 2005: The Base Class Library," pp. 473-474.

Synchronized types

- Some .NET types that aren't ordinarily thread-safe offer thread-safe version

```
// Create an ArrayList object, and add some values to it
ArrayList al = new ArrayList();
Al.Add(1); al.Add(2); al.Add(3);
// Create a synchronized, thread-safe version
ArrayList syncAl = ArrayList.Synchronized(al);
// Prove that the new object is thread-safe
Console.WriteLine(al.IsSynchronized);     // => False;
Console.WriteLine(syncAl.IsSynchronized); // => True;
// You can share the syncAl object among different
// threads
```

From F. Balena, "Visual C# 2005: The Base Class Library," pp. 477-478.

Synchronized types - disadvantages

- Accessing synchronized objects is slower than accessing the original nonsynchronized object
- Generally better (in terms of speed) to use regular types and synchronize via locks

Info from F. Balena, “Visual C# 2005: The Base Class Library,” p. 474.

Problems with locks (1)

- **Overhead:** acquiring and releasing locks takes time
  - So don't acquire locks too often
- **Deadlocks:** lock acquisition order must be consistent to avoid these
  - So don't have very many locks, or only acquire one at a time
- **Contention:** sometimes somebody else has the lock
  - So never hold locks for too long
  - So have lots of little locks

From B. Dawson, "Lockless Programming in Games," http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Problems with locks (2)

- **Priority inversions:** if a thread is swapped out while holding a lock, progress may stall
  - –Changing thread priorities can lead to this
  - –Xbox 360 system threads can briefly cause this

From B. Dawson, “Lockless Programming in Games,” http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Sensible reaction

- Use locks carefully
  - –Don't lock too frequently
  - –Don't lock for too long
  - –Don't use too many locks
  - –Don't have one central lock
- Or, try lockless

From B. Dawson, “Lockless Programming in Games,” http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Lockless programming

- Techniques for safe multi-threaded data sharing without locks
- Pros:
  - May have lower overhead
  - Avoids deadlocks
  - May reduce contention
  - Avoids priority inversions
- Cons
  - Very limited abilities
  - Extremely tricky to get right
  - Generally non-portable

Polling

- Main thread checks flag variables set by the worker threads when they finish
- Useful if main thread can do some stuff (e.g., eye-candy animation in a turn-based strategy game) independently of the worker threads (e.g. AI), but needs worker threads to finish before continuing (e.g. making the computer's move)

Polling example

Code from Joseph Hall, "XNA Game Studio Express," p. 608

```
bool done = false;
while (!done)
{
  Thread.Sleep(0);
  done = true;
  for int(i = 0; i < m_ThreadDone.Length;
                                    i++)
  {
    done &= m_ThreadDone[i];
  }
}
```

Worker thread i sets `m_ThreadDone[i]=true` before it exits

The problem with polling

- Polling takes up “C# cycles”
- If your main thread only needs to wait until its worker threads are done, the Wait/Pulse approach is better
  - Let the .NET runtime handle it!
  - Uh… oh, but only on Windows. ☹

True or False?

"If all you are doing is reading or writing a shared integer variable, nothing can go wrong and you don't need any lock blocks, since reads and writes correspond to a single CPU instruction… right?"

Info from F. Balena, "Visual C# 2005: The Base Class Library," p. 472.

Beware enregistering

```
private bool Done = false;

void TheTask();
{
  // Exit the loop when another thread has set the Done
  // flag or when the task being performed is complete.
  while (this.Done == false)
  {
    // Do some stuff
    if (nothingMoreToDo)
    {
      this.Done = true;
      break;
    }
  }
}
```

Enregistering: compiler caches variable in a register, not in L2 or main memory

From F. Balena, “Visual C# 2005: The Base Class Library,” p. 472.

`volatile` fields

```
private volatile bool Done = false;
```

- `volatile` tells compiler other threads may be reading or writing to the variable, so don't enregister it
- Does not ensure operations are carried out atomically for classes, structs, arrays…
- Does not ensure atomic read+write for anything
  - Increment, decrement
  - Test & Set
- Can still be problematic when doing "real C++ XDK" Xbox 360 programming (we'll return to this later)

Info from F. Balena, "Visual C# 2005: The Base Class Library," p. 474.

Interlocked.X (1)

Atomic increment and decrement:

```
int lockCounter = 0;

// Increment the counter and execute some code if
// its previous value was zero
if (Interlocked.Increment(ref lockCounter) == 1)
{
  …
}
// Decrement the shared counter.
Interlocked.Decrement(ref lockCounter);
```

Can also increment or decrement by an arbitray amount with a second argument

From F. Balena, “Visual C# 2005: The Base Class Library,” p. 485.

Interlocked.X (2)

- Can assign a value and return its previous value as an atomic operation:

```
string s1 = "123";
string s2 = Interlocked.Exchange(ref s1, "abc");
```

After execution, s2 = "123", s1 = "abc"

- Variation to the assignment if a and c are equal (reference equality in the case of objects):

```
Interlocked.CompareExchange(ref a, b, c)
```

From F. Balena, "Visual C# 2005: The Base Class Library," p. 485.

Out-of-order read/writes (1)

- “CPUs employ performance optimizations that can result in out-of-order execution, including memory load and store operations.”

- “Memory operation reordering normally goes unnoticed within a single thread of execution, but causes unpredictable behaviour in concurrent programs and device drivers unless carefully controlled.”

http://en.wikipedia.org/wiki/Memory_barrier

Out-of-order read/writes (2)

- "When a program runs on a single CPU, the hardware performs the necessary book-keeping to ensure that programs execute as if all memory operations were performed in program order, hence memory barriers are not necessary."

http://en.wikipedia.org/wiki/Memory_barrier

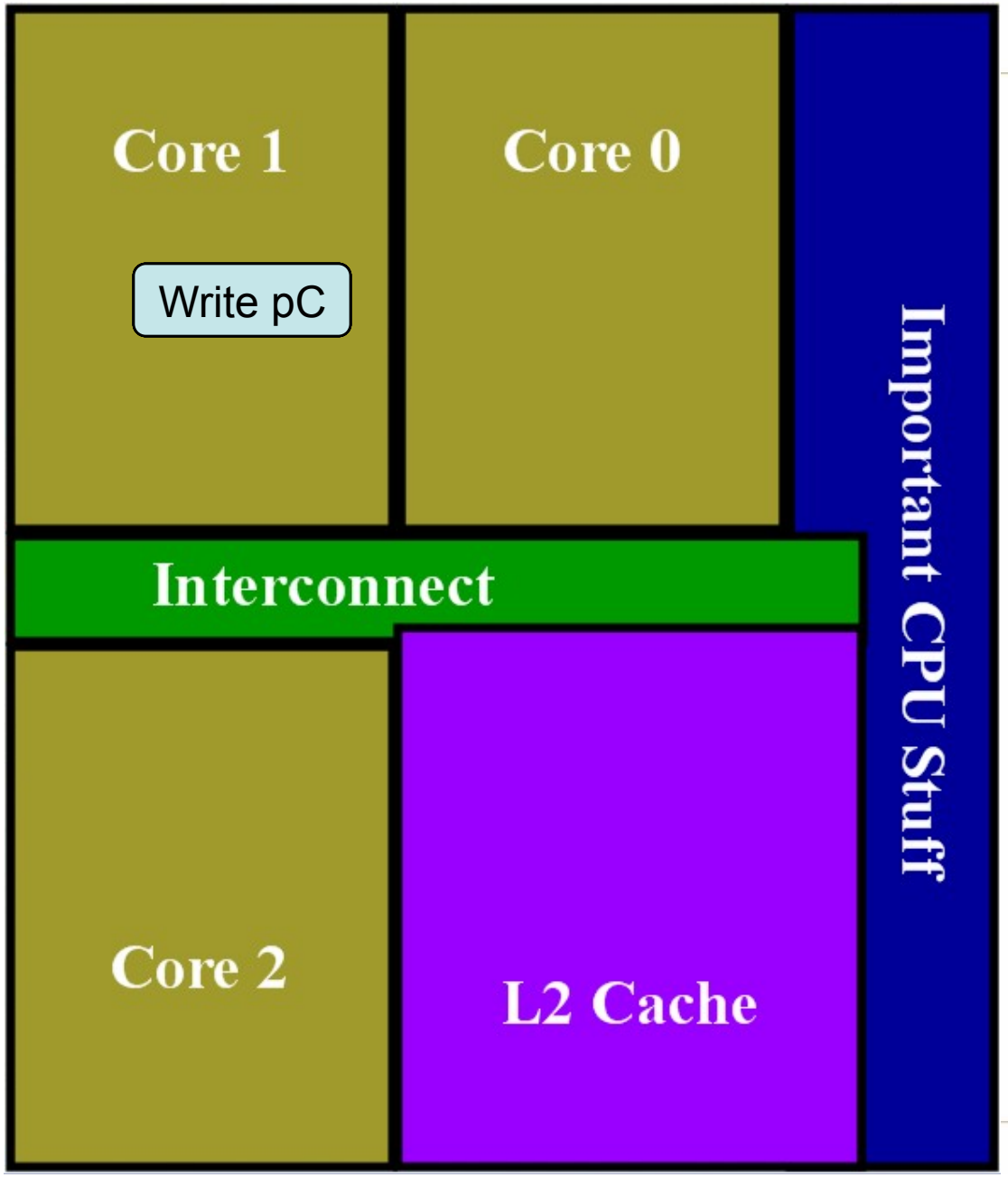
Out-of-order read/writes (3)

- “However, when the memory is shared with multiple devices, such as other CPUs in a multiprocessor system, or memory mapped peripherals, out-of-order access may affect program behavior.”

- “For example a second CPU may see memory changes made by the first CPU in a sequence which differs from program order.”

http://en.wikipedia.org/wiki/Memory_barrier

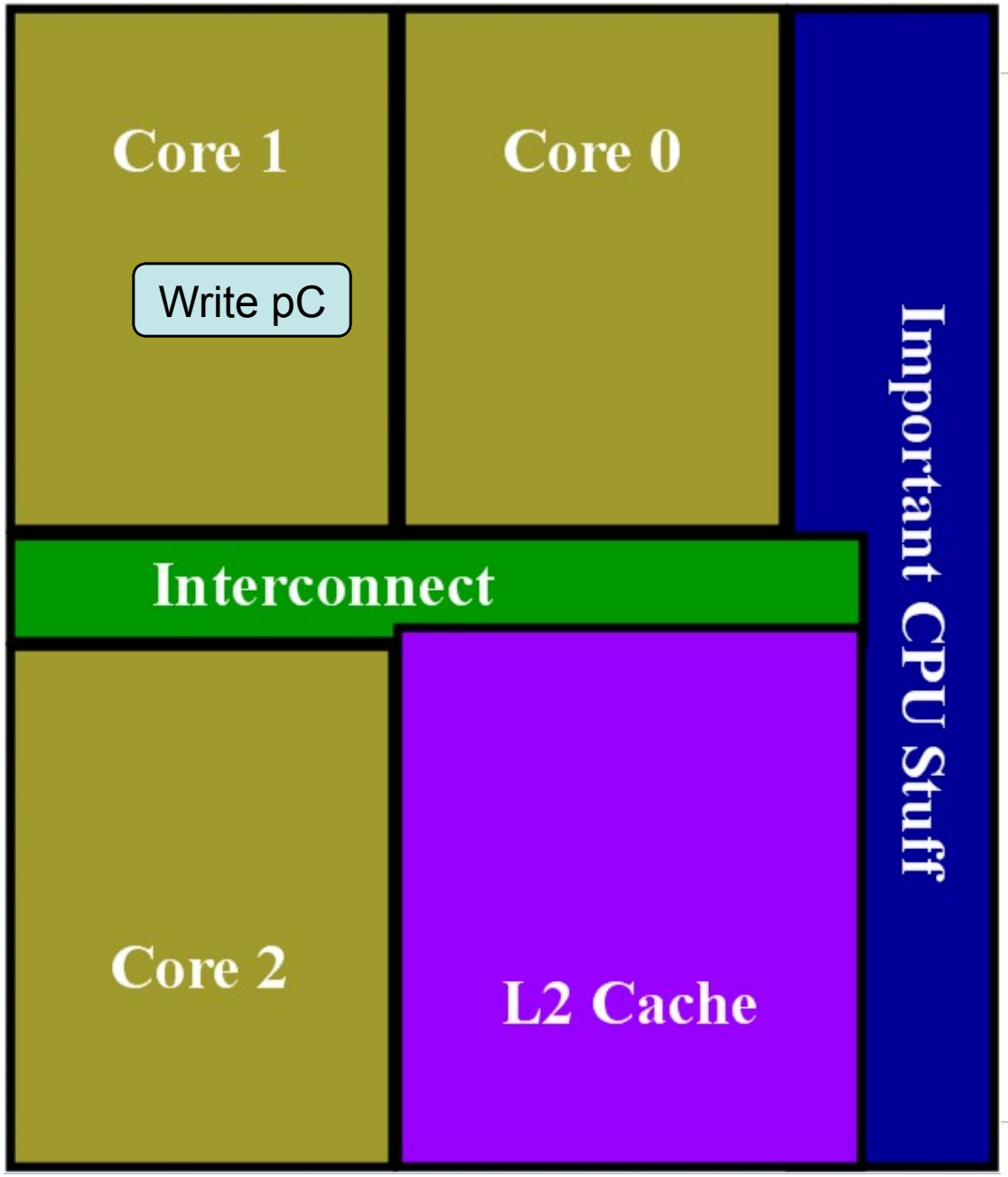
Simple CPU/Compiler Model



Read pC
Write pA
Write pB
Read pD
Write pC

From B. Dawson, “Lockless Programming in Games,” http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Alternate CPU Model – Writes pass Writes

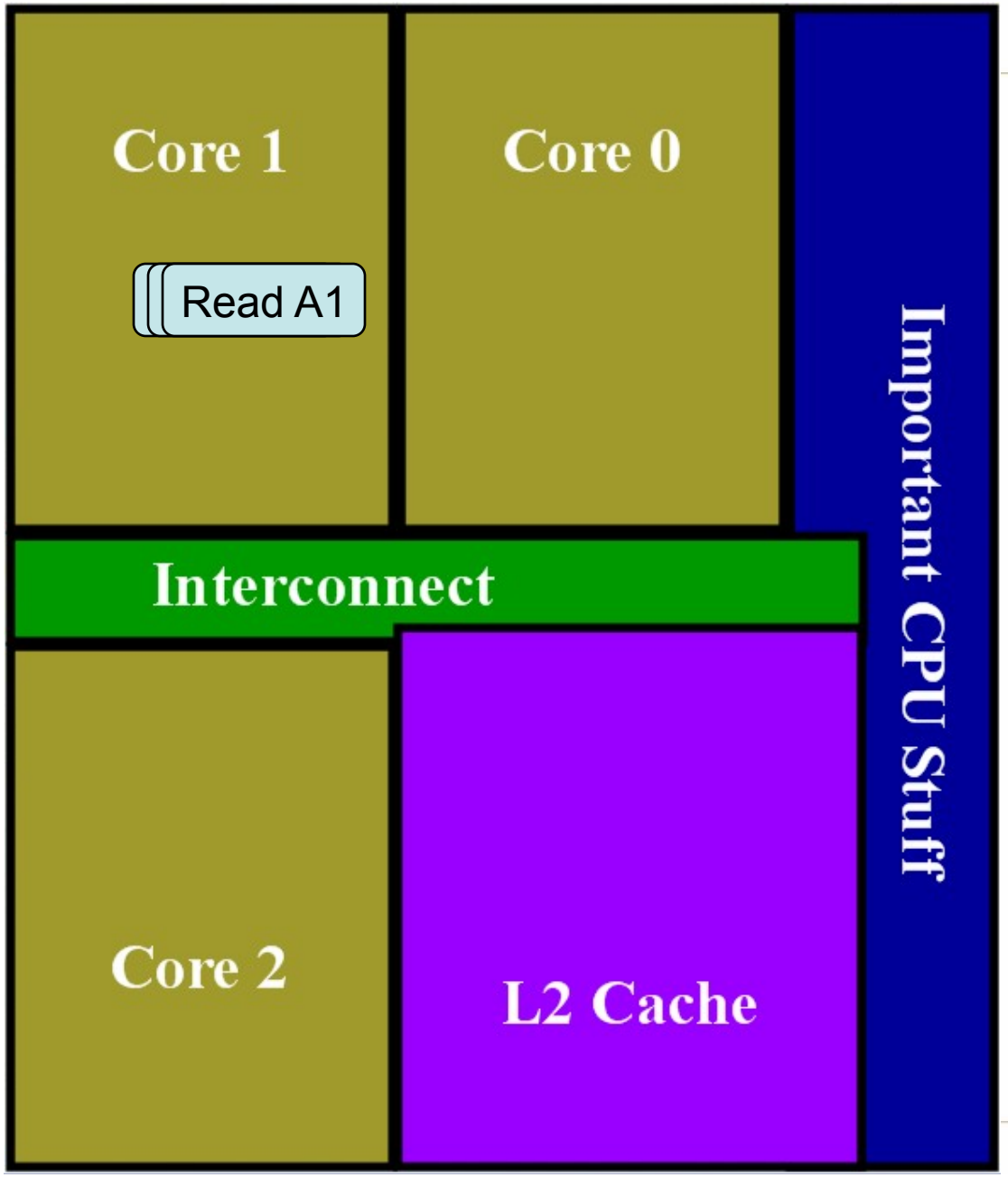


Write pA
Write pB
Write pC

Visible order:
Write pA
Write pC
Write pB

From B. Dawson, “Lockless Programming in Games,” http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Alternate CPU – Reads Pass Reads

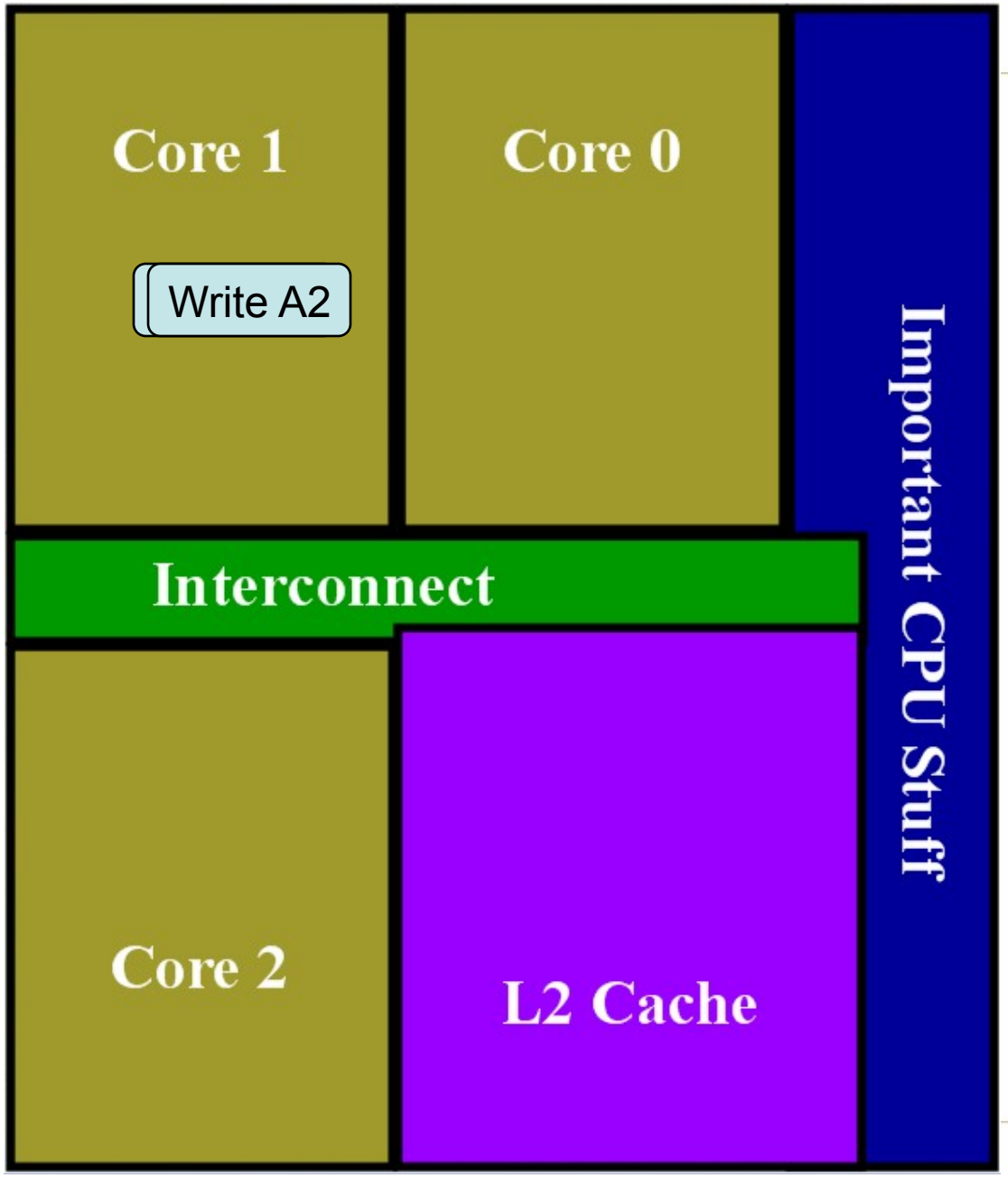


Read A1
Read A2
Read A1

Visible order:
Read A1
Read A1
Read A2

From B. Dawson, “Lockless Programming in Games,” http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Alternate CPU – Writes Pass Reads

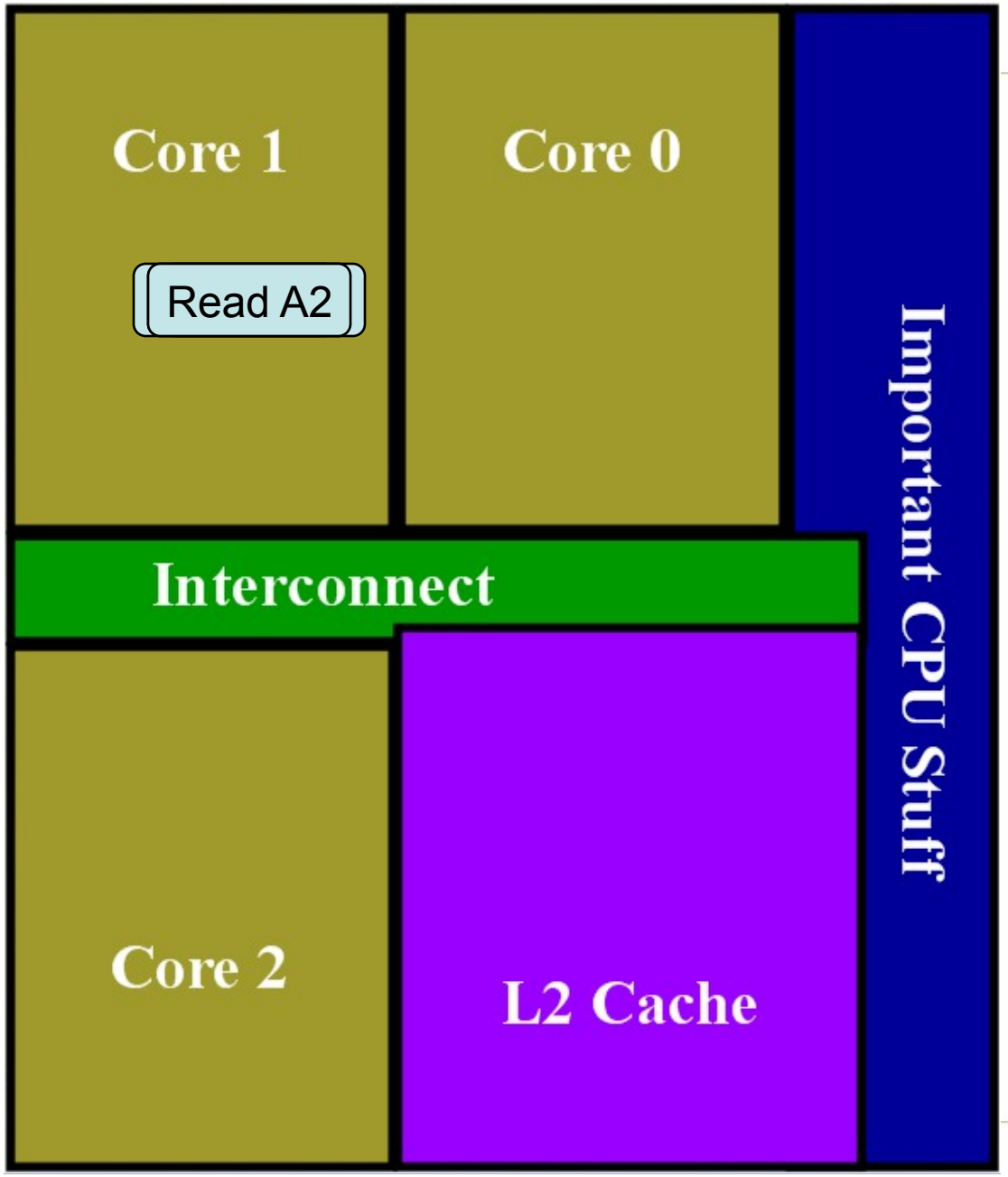


Read A1
Write A2

Visible order:
Write A2
Read A1

From B. Dawson, “Lockless Programming in Games,” http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Alternate CPU – Reads Pass Writes



From B. Dawson, "Lockless Programming in Games," http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Read A1
Write A2
Read A2
Read A1

Visible order:

Read A1
Read A1
Write A2
Read A2

Memory Models

From B. Dawson, “Lockless Programming in Games,” http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

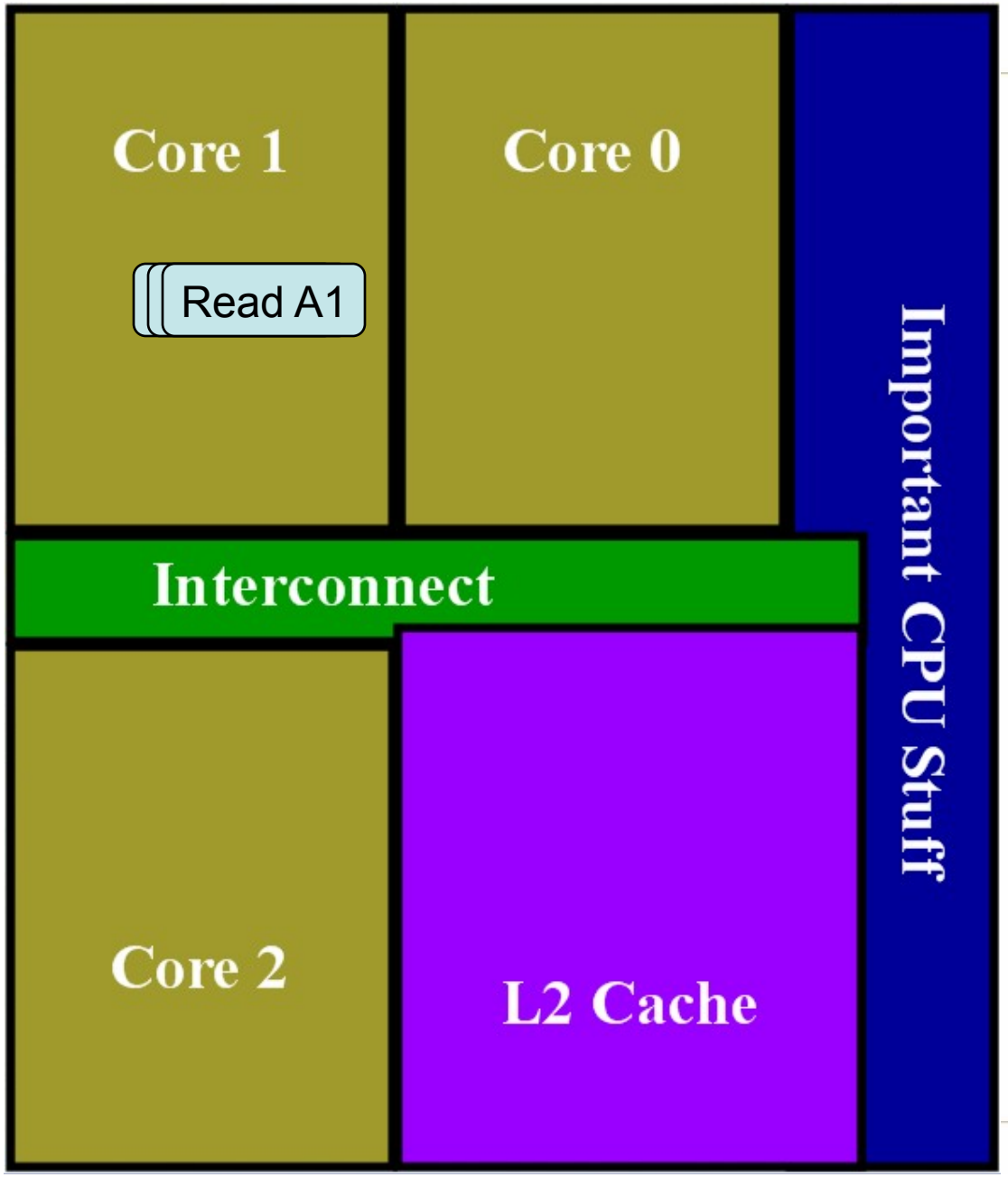
	x86/x64	PowerPC	ARM	IA64
store can pass store?	No	Yes*	Yes*	Yes*
load can pass load?	No	Yes	Yes	Yes
store can pass load?	No	Yes	Yes	Yes
load can pass store?**	Yes	Yes	Yes	Yes

- "Pass" means "visible before"
- Memory models are actually more complex than this
  - May vary for cacheable/non-cacheable, etc.
- This *only* affects multi-threaded lock-free code!!!

* Only stores to different addresses can pass each other

** Loads to a previously stored address will load that value

Improbable CPU – Reads *Don't* Pass Writes



Read A1
Write A2
Read A1

From B. Dawson, "Lockless Programming in Games," http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Reads must pass writes!

- Reads not passing writes would mean L1 cache is frequently disabled
  - Every read that follows a write would stall for shared storage latency
- Huge performance impact
- Therefore, on x86 and x64 (on *all* modern CPUs) reads can pass writes

From B. Dawson, “Lockless Programming in Games,”
http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Memory barriers

- “a class of instructions which cause a central processing unit (CPU) to enforce an ordering constraint on memory operations issued before and after the barrier instruction.”

- PowerPC: sync, lwsync, eieio assembly instructions

http://en.wikipedia.org/wiki/Memory_barrier

PowerPC memory barriers

- Assembly instructions:
  - lwsync: lightweight sync (still lets reads pass writes)
  - sync, i.e. hwsync: heavyweight sync (stops all reordering)
  - eieio: “Enforce In-Order Execution of I/O”

http://en.wikipedia.org/wiki/Memory_barrier

Further information from an e-mail from Bruce Dawson

MyExportBarrier();

- Prevents reordering of writes by compiler *or* CPU
  - Used when handing out access to data
- x86/x64: _ReadWriteBarrier();
  - Compiler intrinsic, prevents compiler reordering
- PowerPC: __lwsync();
  - Hardware barrier, prevents CPU write reordering
- ARM: __dmb(); // Full hardware barrier
- IA64: __mf(); // Full hardware barrier

- Positioning is crucial!
  - Write the data, MyExportBarrier, write the control value
- Export-barrier followed by write is known as write-release semantics

From B. Dawson, “Lockless Programming in Games,”
http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

MyImportBarrier();

- Prevents reordering of reads by compiler *or* CPU
  - Used when gaining access to data
- x86/x64: _ReadWriteBarrier();
  - Compiler intrinsic, prevents compiler reordering
- PowerPC: __lwsync(); or isync();
  - Hardware barrier, prevents CPU read reordering
- ARM: __dmb(); // Full hardware barrier
- IA64: __mf(); // Full hardware barrier

- Positioning is crucial!
  - Read the control value, MyImportBarrier, read the data
- Read followed by import-barrier is known as read-acquire semantics

From B. Dawson, “Lockless Programming in Games,”
http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Full Memory Barrier

- MemoryBarrier();
  - x86: __asm xchg Barrier, eax
  - x64: __faststorefence();
  - Xbox 360: __sync();
  - ARM: __dmb();
  - IA64: __mf();
- Prevents all reordering – including preventing reads passing writes
- Most expensive barrier type

From B. Dawson, “Lockless Programming in Games,”
http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Reordering implications

- Publisher/Subscriber model
- Thread A:

```
g_data = data;
g_dataReady = true;
```

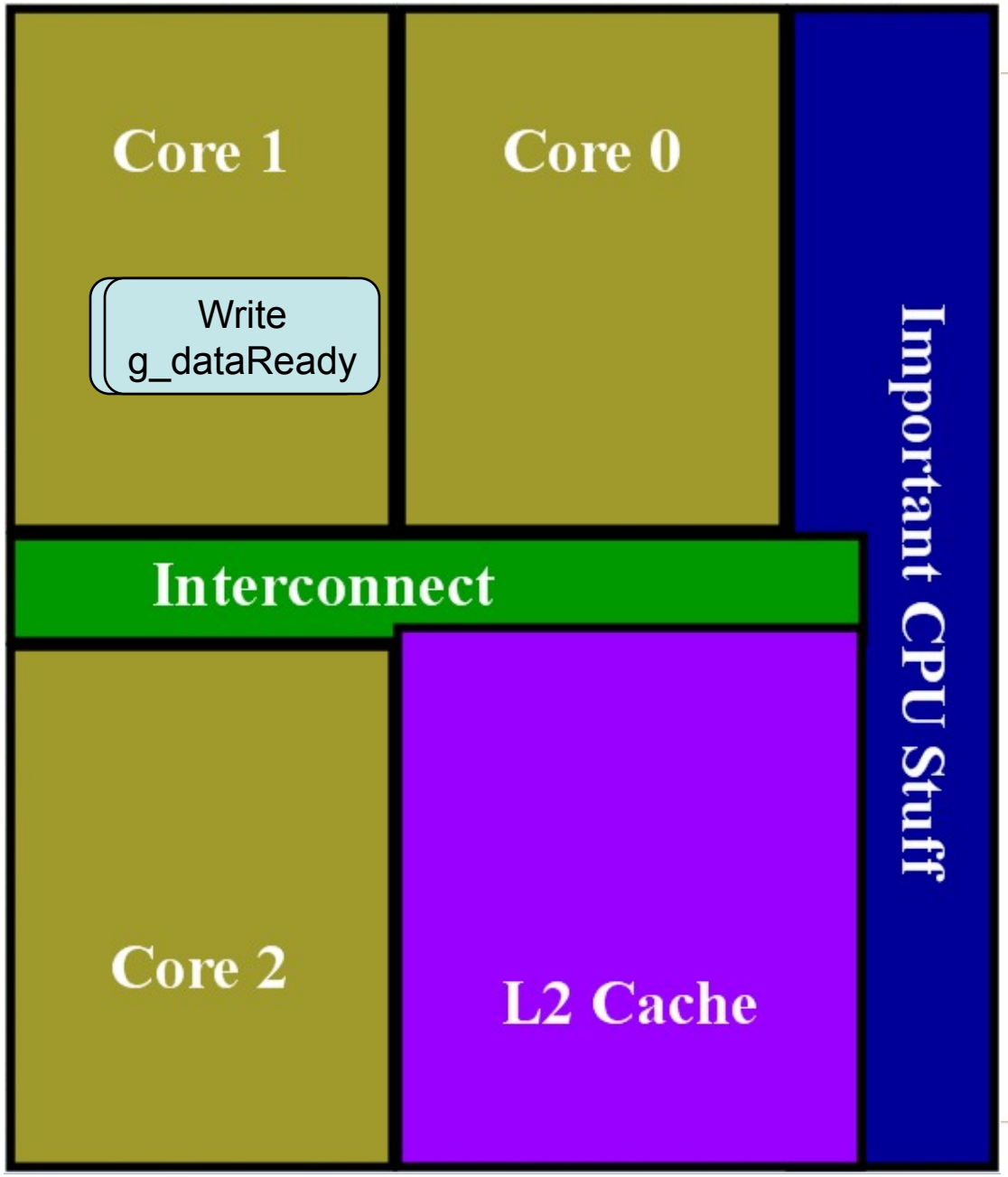
- Thread B:

```
if( g_dataReady )
    process( g_data );
```

- Is it safe?

From B. Dawson, “Lockless Programming in Games,” http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Publisher/Subscriber on PowerPC



From B. Dawson, "Lockless Programming in Games," http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Proc 1:

Write g_data

Write g_dataReady

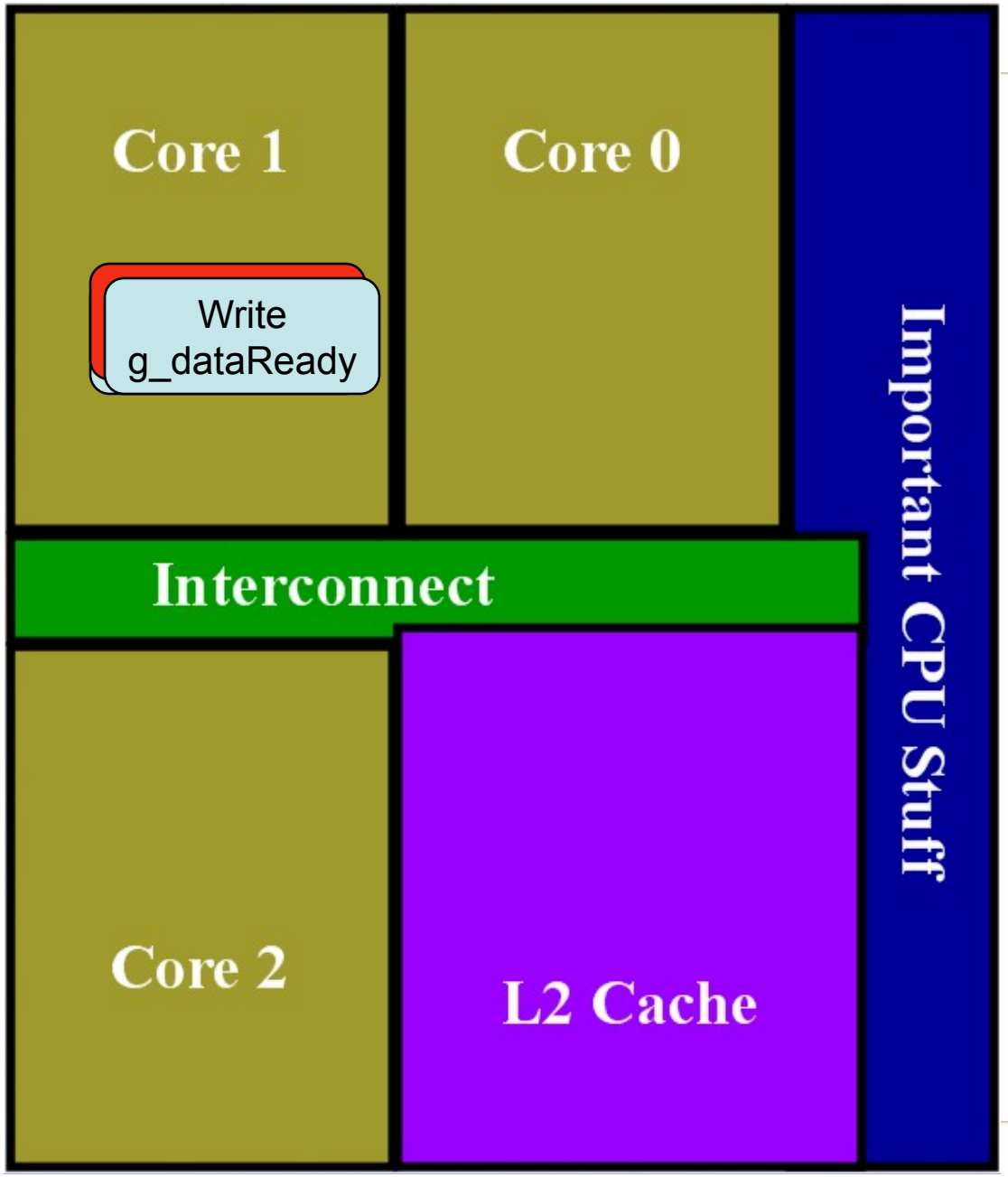
Proc 2:

Read g_dataReady

Read g_data

- Writes may reach L2 out of order

Publisher/Subscriber on PowerPC



From B. Dawson, "Lockless Programming in Games,"
http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Proc 1:

Write g_data

MyExportBarrier();

Write g_dataReady

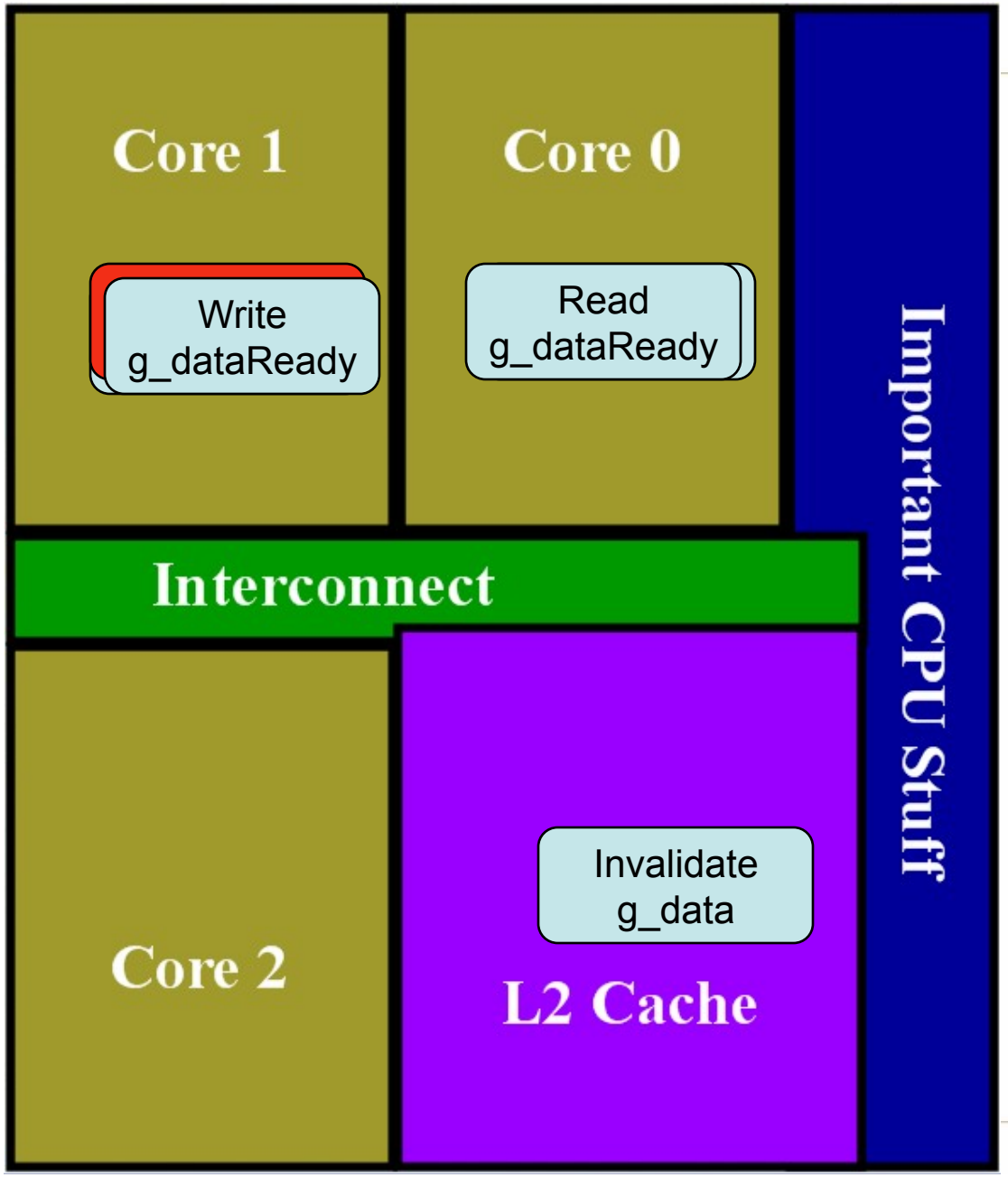
Proc 2:

Read g_dataReady

Read g_data

- Writes now reach L2 in order

Publisher/Subscriber on PowerPC



From B. Dawson, "Lockless Programming in Games,"
http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Proc 1:

Write g_data

MyExportBarrier();

Write g_dataReady

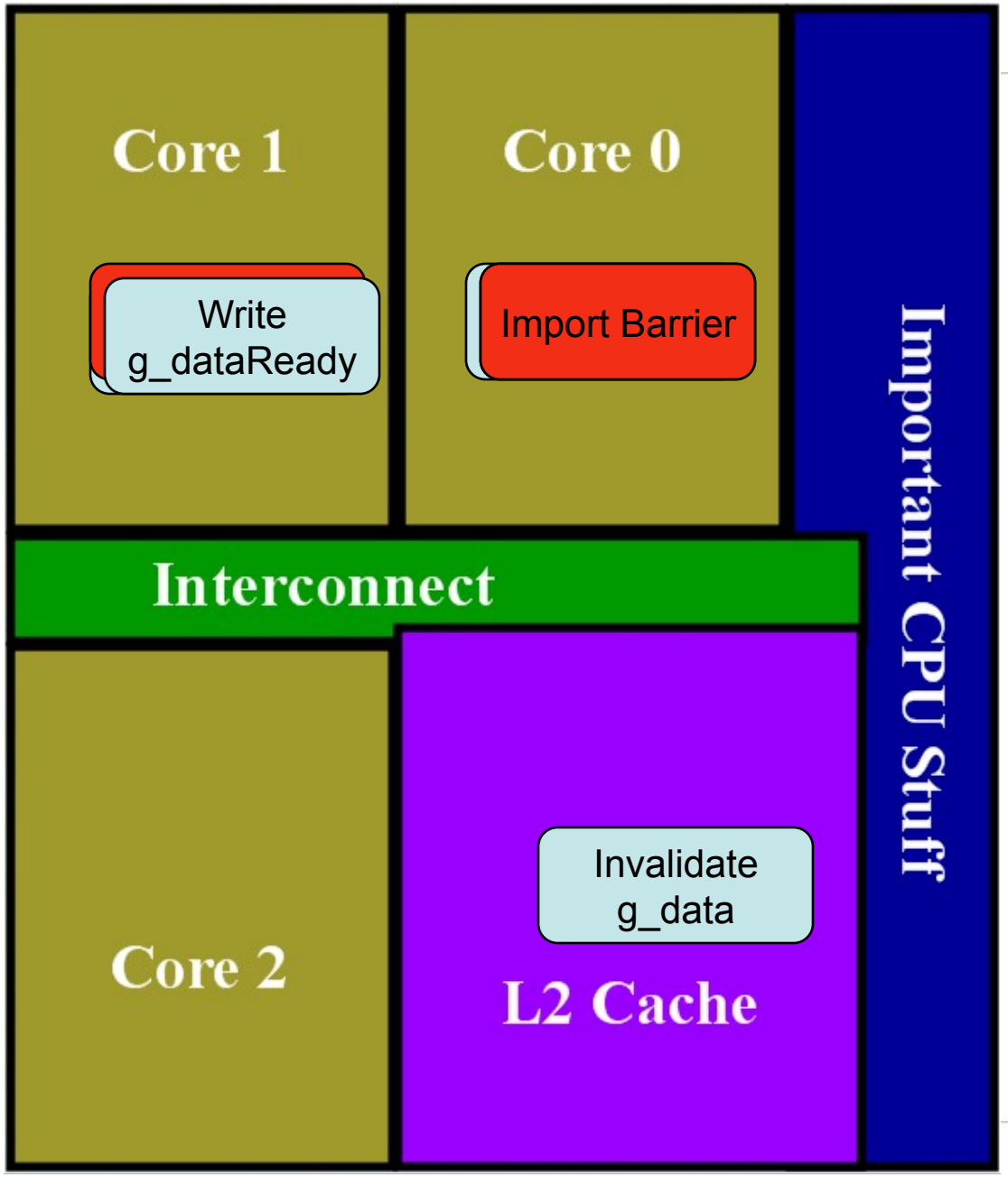
Proc 2:

Read g_dataReady

Read g_data

- Reads may leave L2 out of order – g_data may be stale

Publisher/Subscriber on PowerPC



From B. Dawson, "Lockless Programming in Games,"
http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Proc 1:

Write g_data

MyExportBarrier();

Write g_dataReady

Proc 2:

Read g_dataReady

MyImportBarrier();

Read g_data

- It's all good!

x86/x64 FTW!!!

- Not so fast…
- Compilers can be just as evil as processors
- Compilers will rearrange your code as much as legally possible
  - And compilers assume your code is single threaded
- Compiler and CPU reordering barriers needed

From B. Dawson, “Lockless Programming in Games,” http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Barrier Summary

- MyExportBarrier when publishing data, to prevent write reordering
- MyImportBarrier when acquiring data, to prevent read reordering
- MemoryBarrier to stop all reordering, including reads passing writes
- Identify where you are publishing/ releasing and where you are subscribing/acquiring

From B. Dawson, “Lockless Programming in Games,”
http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

What about “volatile” in C++?

- Standard volatile semantics not designed for multi-threading
  - Compiler can move normal reads/writes past volatile reads/writes
  - Also, doesn’t prevent CPU reordering
- VC++ 2005+ volatile is better…
  - Acts as read-acquire/write-release on x86/x64 and Itanium
  - Doesn’t prevent hardware reordering on Xbox 360
- Watch for atomic<T> in C++0x
  - Sequentially consistent by default but can choose from four memory models

From B. Dawson, “Lockless Programming in Games,”
http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Interlocked.X

- Interlocked.X is a full barrier on Windows for x86, x64, and Itanium
- Not a barrier at all on Xbox 360
  - Oops. Still atomic, just not a barrier

From B. Dawson, “Lockless Programming in Games,” http://www.gdcvault.com/play/1751/Lockless_Programming_in_Games

Dangers in the Xbox 360 CPU (1)

- `Interlocked.X` & `volatile`-type operations are very fast
- Safe on Windows (because of Intel memory model)
- When doing "real X++ XDK" Xbox 360 development, `Interlocked.X` and `volatile` keyword will prevent *compiler* from reordering reads and writes, but not the CPU!

Info from B. Dawson, "Lockless Programming Considerations for Xbox 360 and Microsoft Windows," msdn2.microsoft.com/en-us/library/bb310595.aspx

Dangers in the Xbox 360 CPU (2)

- Xbox 360 CPU may reorder writes to L2 cache
- Writes go to one of 8 store-gather buffers first, not L2 cache
- 64 bytes can be transferred from a buffer to L2 in one op
- Reads are an issue too
- None of this is a problem with single-threaded code, but can be a problem with multithreaded
- Allowed in PowerPC's "relaxed memory consistency" model

Info from B. Dawson, "Lockless Programming Considerations for Xbox 360 and Microsoft Windows," msdn2.microsoft.com/en-us/library/bb310595.aspx

Dangers in the Xbox 360 CPU (3)

- Can still do native lockless programming in on the Xbox 360, but you have to really know what you're doing

Info from B. Dawson, "Lockless Programming Considerations for Xbox 360 and Microsoft Windows," msdn2.microsoft.com/en-us/library/bb310595.aspx

Playing it safe

- Locks and Mutexes provide needed memory barriers
- Makes them easier to use than lockless programming

C#: MemoryBarrier()

- “Synchronizes memory access as follows: The processor executing the current thread cannot reorder instructions in such a way that memory accesses prior to the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier.”

http://msdn.microsoft.com/en-us/library/system.threading.thread.memorybarrier.aspx

Notes on MemoryBarrier()

- "MemoryBarrier is required only on multiprocessor systems with weak memory ordering (for example, a system employing multiple Intel Itanium processors)."

- "For most purposes, the C# lock statement…the Monitor class provide easier ways to synchronize data."

http://msdn.microsoft.com/en-us/library/system.threading.thread.memorybarrier.aspx

Compact Framework to the rescue? (1)

- “Now, we have access to a fair few Interlocked.xxx methods in the framework, which would do fine if I were programming on Windows, however on the 360 I need to be sure that I am not going to be caught out by write-reordering by the CLR or CPU. (i.e the reading thread spins until Interlocked.xxx sees a flag change, but the writing thread's CPU hasn't finished writing out its data to its cache, causing the reading thread to see old data).”

  - CosmicFlux, 7/9/2007

Creator’s Club community forum post, “Lightweight locking on the 360”
http://forums.xna.com/forums/t/3252.aspx

Compact Framework to the rescue? (2)

"From the CF guys who implemented these methods: *The Interlocked functions in NETCF provide a memory barrier on both sides of the interlocked operation. (This is different than native Xbox360 programming.) In addition, we provide the Thread.MemoryBarrier api if the customer needs to place an explicit memory barrier. Also, the Monitor functions are generally a higher performance operation than using a Mutex unless there are many many collisions on the lock.* They were quite impressed that someone actually understood the issues involved :-)"

- Shawn Hargreaves, 7/10/2007

Creator's Club community forum post, "Lightweight locking on the 360"
http://forums.xna.com/forums/t/3252.aspx

Take home message

- Xbox 360 CPU may cause different threads to see writes happening out-of-order (not a problem on Windows)
- Lockless techniques (Interlocked.X, etc.) can give faster performance
  - Only for ninja kung-fu when doing native Xbox 360 development; gains must justify complexity
  - Should be “safer” in XNA (but lockless programming is still tricky)
- Monitor locks and simple polling are probably easiest/safest at this stage in your career

Setting thread priority in C#

```
t.Priority = ThreadPriority.Normal;
```

or

```
Highest, AboveNormal, BelowNormal, Lowest
```

- Defaults to normal
- OS may ignore you
- Be careful about boosting thread priority
  - If the priority is too high, you could cause the system to hang and become unresponsive
  - If the priority is too low, the thread may starve

Final bullet from Bruce Dawson & Chuck Walbourn, Microsoft Game Technology Group, "Coding for Multiple Cores," PowerPoint presentation

Locating your threads on the Xbox 360

```
Thread.CurrentThread.SetProcessorAffinity
   (new int[] {index});
```

- Set thread affinity ***within*** the worker thread immediately after starting it
  - Don't forget to call it, or your worker thread will be running on the same hardware thread as your main thread
- Only available on Xbox 360 XNA

Check to see if you're on an Xbox 360

```
#if XBOX360
  Thread.CurrentThread.SetProcessorAffinity
    (new int[] {index});
#endif
```

- No way I know of in C# to manually set processor affinity in Windows like on the Xbox 360

- Windows decides what threads run where

Xbox 360 hardware threads

Ind	CPU	Thr	Comment
0	1	1	Not available in XNA
1	1	2	Available; main thread; game runs here by default
2	2	1	Not available in XNA
3	2	2	Available; parts of the Guide and Dashboard live here
4	3	1	Available; Xbox Live Marketplace downloads
5	3	2	Available; parts of the Guide and Dashboard live here

Table from Joseph Hall, “XNA Game Studio Express,” p. 608

Xbox 360 specific notes (1)

- “If a program holds a lock for too long—because of poor design or because the thread has been swapped out by a higher priority thread—then other threads may be blocked for a long time.”
- “This risk is particularly great on Xbox 360, because the software threads are assigned a hardware thread by the developer, and the operating system won't move them to another hardware thread, even if one is idle.”

Info from B. Dawson, “Lockless Programming Considerations for Xbox 360 and Microsoft Windows,” msdn2.microsoft.com/en-us/library/bb310595.aspx

Xbox 360 specific notes (2)

- The Xbox 360 also has no protection against **priority inversion**, where a high-priority thread spins in a loop while waiting for a low-priority thread to release a lock

Info from B. Dawson, “Lockless Programming Considerations for Xbox 360 and Microsoft Windows,” msdn2.microsoft.com/en-us/library/bb310595.aspx

Advice

- More than one thread per core isn't bad…
- …but more than one processor-intensive task per core is!
- Put most intensive tasks on separate cores, and some less-demanding tasks on those same cores (threads that work in short bursts, disk I/O, etc.)

Advice from Joseph Hall, "XNA Game Studio Express," p. 610

More advice

- Limit number of synchronization points
- Don't lock resources longer than necessary
- Avoid sharing data when possible
- Profile your code before and after to make sure you're getting the performance benefits you expect
  - Very easy to write multithreaded code that performs worse than single threaded!

Advice from Joseph Hall, "XNA Game Studio Express," p. 611