Lockless Programming Considerations for Xbox 360 and Microsoft Windows

By Bruce Dawson
Software Design Engineer
Advanced Technology Group (ATG)

Published: March 28, 2007
Updated: May 3, 2008

Introduction

Lockless programming is a way to safely share changing data between multiple threads without the cost of acquiring and releasing locks. This sounds like a panacea, but lockless programming is complex and subtle, and sometimes doesn't give the benefits that it promises. Lockless programming is particularly complex on Xbox 360.

Lockless programming is a valid technique for multi-threaded programming, but it should not be used lightly. Before using it you must understand the complexities, and you should measure carefully to make sure that it is actually giving you the gains that you expect. In many cases there are simpler and faster solutions, such as sharing data less frequently, which should be used instead.

Using lockless programming correctly and safely requires significant knowledge of both your hardware and your compiler. This paper gives an overview of some of the issues to consider when trying to use lockless programming techniques.

Programming with Locks

When writing multi-threaded code it is often necessary to share data between threads. If multiple threads are simultaneously reading and writing the shared data structures, then memory corruption can occur. The simplest way of solving this problem is using locks. For instance, if **ManipulateSharedData** should only be executed by one thread at a time, then a **CRITICAL_SECTION** can be used to guarantee this, as in the following code:

```
// Initialize
CRITICAL_SECTION cs;
InitializeCriticalSection(&cs);

// Use
void ManipulateSharedData()
{
    EnterCriticalSection(&cs);
    // Manipulate stuff...
    LeaveCriticalSection(&cs);
}

// Destroy
DeleteCriticalSection(&cs);
```

This code is fairly simple and straightforward, and it is easy to tell that it is correct. However, programming with locks comes with several potential disadvantages. For example, if two threads try to acquire the same two locks but acquire them in a different order, then a deadlock may result. 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. 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. Finally, if a deferred procedure call or interrupt service routine tries to acquire a lock, then a deadlock may occur.

Despite these problems, synchronization primitives, such as critical sections, are generally the best way of coordinating multiple threads. If the synchronization primitives are too slow, then the best solution is usually to use them less frequently. However, for those who can afford the extra complexity, another option is *lockless programming*.

Lockless Programming

Lockless programming, as the name suggests, is a family of techniques for safely manipulating shared data without using locks. There are lockless algorithms available for passing messages, sharing lists and queues of data, and other tasks.

When doing lockless programming, there are two challenges that you must deal with: non-atomic operations and reordering.

Non-Atomic Operations

An *atomic operation* is one that is indivisible — one where other threads are guaranteed to never see the operation when it is half done. Atomic operations are important for lockless programming, because without them, other threads might see half-written values, or otherwise inconsistent state.

On all modern processors, you can assume that reads and writes of naturally aligned native types are atomic. As long as the memory bus is at least as wide as the type being read or written, the CPU reads and writes these types in a single bus transaction, making it impossible for other threads to see them in a half-completed state. On x86 and x64, there is no guarantee that reads and writes larger than eight bytes are atomic. This means that 16-byte reads and writes of streaming SIMD extension (SSE) registers, and string operations, might not be atomic.

Reads and writes of types that are not naturally aligned — for instance, writing **DWORDs** that cross four-byte boundaries — are not guaranteed to be atomic. The CPU may have to do these reads and writes as multiple bus transactions, which could allow another thread to modify or see the data in the middle of the read or write.

Composite operations, such as the read-modify-write sequence that occurs when you increment a shared variable, are not atomic. On Xbox 360, these operations are

implemented as multiple instructions (**lwz**, **addi**, and **stw**), and the thread could be swapped out partway through the sequence. On x86 and x64, there is a single instruction (**inc**) that can be used to increment a variable in memory. If this instruction is used, then incrementing a variable is atomic on single-processor systems, but it is still not atomic on multi-processor systems. Making **inc** atomic on x86- and x64-based multi-processor systems requires using the **lock** prefix, which prevents another processor from doing its own read-modify-write sequence between the read and the write of the **inc** instruction.

The following code shows some examples:

```
// This write is not atomic because it is not natively aligned.
DWORD* pData = (DWORD*)(pChar + 1);
*pData = 0;

// This is not atomic because it is three separate operations.
++g_globalCounter;

// This write is atomic.
g_alignedGlobal = 0;

// This read is atomic.
DWORD local = g_alignedGlobal;
```

Guaranteeing Atomicity

You can be sure you are using atomic operations by a combination of the following:

- naturally atomic operations
- locks to wrap composite operations
- operating system functions that implement atomic versions of popular composite operations

Incrementing a variable is not an atomic operation, and incrementing may lead to data corruption if executed on multiple threads.

```
// This will be atomic.
g_globalCounter = 0;

// This is not atomic and gives undefined behavior
// if executed on multiple threads
++g_globalCounter;
```

Win32 comes with a family of functions that offer atomic read-modify-write versions of several common operations. These are the **Interlocked***Xxx* family of functions. If all modifications of the shared variable use these functions, then the modifications will be thread safe.

```
// Incrementing our variable in a safe lockless way.
InterlockedIncrement(&g_globalCounter);
```

Reordering

A more subtle problem is reordering. Reads and writes do not always happen in the order that you have written them in your code, and this can lead to very confusing problems. In many multi-threaded algorithms, a thread writes some data and then writes to a flag that tells other threads that the data is ready. This is known as a write-release. If the writes are reordered then other threads may see that the flag is set before they can see the written data.

Similarly, in many cases, a thread reads from a flag and then reads some shared data if the flag says that the thread has acquired access to the shared data. This is known as a read-acquire. If reads are reordered, then the data may be read from shared storage before the flag, and the values seen might not be up to date.

Reordering of reads and writes can be done both by the compiler and by the processor. Compilers and processors have done this reordering for years, but on single-processor machines it was less of an issue. This is because CPU rearrangement of reads and writes is invisible on single-processor machines (for code that is not part of a device driver), and compiler rearrangement of reads and writes is less likely to cause problems on single-processor machines.

If the compiler or the CPU rearranges the writes shown in the following code, then another thread may see that the *alive* flag is set while still seeing the old values for *x* or *y*. Similar rearrangement can happen when reading.

In this code, one thread adds a new entry to the sprite array:

```
// Create a new sprite by writing its position into an empty
// entry and then setting the 'alive' flag. If 'alive' is
// written before x or y then errors may occur.
g_sprites[nextSprite].x = x;
g_sprites[nextSprite].y = y;
g_sprites[nextSprite].alive = true;
```

In this next code block, another thread reads from the sprite array:

```
// Draw all sprites. If the reads of x and y are moved ahead of
// the read of 'alive' then errors may occur.
for( int i = 0; i < numSprites; ++i )
{
    if( g_sprites[nextSprite].alive )
    {
        DrawSprite( g_sprites[nextSprite].x,
                g_sprites[nextSprite].y );
    }
}
```

To make this sprite system safe, we need to prevent both compiler and CPU reordering of reads and writes.

Understanding CPU Rearrangement of Writes

Some CPUs rearrange writes so that they are externally visible to other processors or devices in non-program order. This rearranging is never visible to single-threaded non-driver code, but it can cause problems in multi-threaded code.

Xbox 360

While the Xbox 360 CPU does not reorder instructions, it does rearrange write operations, which complete after the instructions themselves. This rearranging of writes is specifically allowed by the PowerPC memory model.

Writes on Xbox 360 do not go directly to the L2 cache. Instead, in order to improve L2 cache write bandwidth, they go through store queues and then to store-gather buffers. The store-gather buffers allow 64-byte blocks to be written to the L2 cache in one operation. There are eight store-gather buffers, which allow efficient writing to several different areas of memory.

The store-gather buffers are normally written to the L2 cache in first-in-first-out (FIFO) order. However if the target cache-line of a write is not in the L2 cache, then that write may be delayed while the cache-line is fetched from memory.

Even when the store-gather buffers are written to the L2 cache in strict FIFO order, this does not guarantee that individual writes are written to the L2 cache in order. For instance, imagine that the CPU writes to location 0x1000, then to location 0x2000, and then to location 0x1004. The first write allocates a store-gather buffer and puts it at the front of the queue. The second write allocates another store-gather buffer and puts it next in the queue. The third write adds its data to the first store-gather buffer, which remains at the front of the queue. Thus, the third write ends up going to the L2 cache before the second write.

Reordering caused by store-gather buffers is fundamentally unpredictable, especially because both threads on a core share the store-gather buffers, making the allocation and emptying of the store-gather buffers highly variable.

This is one example of how writes can be reordered. There may be other possibilities.

x86 and x64

Even though x86 and x64 CPUs do reorder instructions, they generally do not reorder write operations relative to other writes. There are some exceptions for write-combined memory. Additionally, string operations (MOVS and STOS) and 16-byte SSE writes can be internally reordered, but otherwise, writes are not reordered relative to each other.

Understanding CPU Rearrangement of Reads

Some CPUs rearrange reads so that they effectively come from shared storage in non-program order. This rearranging is never visible to single-threaded non-driver code, but can cause problems in multi-threaded code.

Xbox 360

Cache misses can cause some reads to be delayed, which effectively causes reads to come from shared memory out of order, and the timing of these cache misses is fundamentally unpredictable. Prefetching and branch prediction can also cause data to come from shared memory out of order. These are just a few examples of how reads can be reordered. There may be other possibilities. This rearranging of reads is specifically allowed by the PowerPC memory model.

x86 and x64

Even though x86 and x64 CPUs do reorder instructions, they generally do not reorder read operations relative to other reads. String operations (MOVS and STOS) and 16-byte SSE reads can be internally reordered, but otherwise reads are not reordered relative to each other.

Other Reordering

Even though x86 and x64 CPUs do not reorder writes relative to other writes, or reorder reads relative to other reads, they can reorder reads relative to writes. Specifically, if a program writes to one location followed by reading from a different location, then the read data may come from shared memory before the written data makes it there. This reordering can break some algorithms, such as Dekker's mutual exclusion algorithms. In Dekker's algorithm, each thread sets a flag to indicate that it wants to enter the critical region, and then checks the other thread's flag to see if the other thread is in the critical region or trying to enter it. The initial code follows.

```
volatile bool f0 = false;
volatile bool f1 = false;

void P0Acquire()
{
    // Indicate intention to enter critical region
    f0 = true;
    // Check for other thread in or entering critical region
    while (f1)
    {
        // Handle contention.
    }
    // critical region
    ...
}


void P1Acquire()
{
    // Indicate intention to enter critical region
```

```
        f1 = true;
        // Check for other thread in or entering critical region
        while (f0)
        {
            // Handle contention.
        }
        // critical region
        ...
    }
```

The problem is that the read of f1 in **P0Acquire** can read from shared storage before the write to f0 makes it to shared storage. Meanwhile, the read of f0 in **P1Acquire** can read from shared storage before the write to f1 makes it to shared storage. The net effect is that both threads set their flags to TRUE, and both threads see the other thread's flag as being FALSE, so they both enter the critical region. Therefore, while problems with reordering on x86- and x64-based systems are less common than on Xbox 360, they definitely can still happen. Dekker's algorithm will not work without hardware memory barriers on any of these platforms.

x86 and x64 CPUs will not reorder a write ahead of a previous read. x86 and x64 CPUs only reorder reads ahead of previous writes if they target different locations.

PowerPC CPUs can reorder reads ahead of writes and can reorder writes ahead of reads, as long as they are to different addresses.

Reordering Summary

The Xbox 360 CPU reorders memory operations much more aggressively than do x86 and x64 CPUs, as shown in the following table. For more details, consult the processor documentation.

Reordering Activity	x86 and x64	Xbox 360
Reads moving ahead of reads	No	Yes
Writes moving ahead of writes	No	Yes
Writes moving ahead of reads	No	Yes
Reads moving ahead of writes	Yes	Yes

Read-Acquire and Write-Release Barriers

The main constructs used to prevent reordering of reads and writes are called read-acquire and write-release barriers. A read-acquire is a read of a flag or other variable to gain ownership of a resource, coupled with a barrier against reordering. Similarly, a write-release is a write of a flag or other variable to give away ownership of a resource, coupled with a barrier against reordering.

The formal definitions, courtesy of Herb Sutter, are:

> A read-acquire executes before all reads and writes by the same thread that follow it in program order.
>
> A write-release executes after all reads and writes by the same thread that precede it in program order.

When your code acquires ownership of some memory, either by acquiring a lock or by pulling an item off of a shared linked list (without a lock), there is always a read involved — testing a flag or pointer to see if ownership of the memory has been acquired. This read may be part of an **Interlocked***Xxx* operation, in which case it involves both a read and a write, but it is the read that indicates whether ownership has been gained. After ownership of the memory is acquired, values are typically read from or written to that memory, and it is very important that these reads and writes execute *after* acquiring ownership. A read-acquire barrier guarantees this.

When ownership of some memory is released, either by releasing a lock or by pushing an item on to a shared linked list, there is always a write involved which notifies other threads that the memory is now available to them. While your code had ownership of the memory, it probably read from or wrote to it, and it is very important that these reads and writes execute *before* releasing ownership. A write-release barrier guarantees this.

It is simplest to think of read-acquire and write-release barriers as single operations; however, they sometimes have to be constructed from two parts: a read or write and a *barrier* that does not allow reads or writes to move across it. In this case, the placement of the barrier is critical. For a read-acquire barrier, the read of the flag comes first, then the barrier, and then the reads and writes of the shared data. For a write-release barrier the reads and writes of the shared data come first, then the barrier, and then the write of the flag.

```
// Read that acquires the data.
if( g_flag )
{
    // Guarantee that the read of the flag executes before
    // all reads and writes that follow in program order.
    BarrierOfSomeSort();

    // Now we can read and write the shared data.
    int localVariable = sharedData.y;
    sharedData.x = 0;

    // Guarantee that the write to the flag executes after all
    // reads and writes that precede it in program order.
    BarrierOfSomeSort();
    // Write that releases the data.
    g_flag = false;
}
```

The only difference between a read-acquire and a write-release is the location of the memory barrier. A read-acquire has the barrier after the lock operation, and a write-release has the barrier before. In both cases the barrier is in-between the references to the locked memory and the references to the lock.

To understand why barriers are needed both when acquiring and when releasing data, it is best (and most accurate) to think of these barriers as guaranteeing synchronization with shared memory, not with other processors. If one processor uses a write-release to release a data structure to shared memory, and another processor uses a read-acquire to gain access to that data structure from shared memory, then the code will work properly. If either processor doesn't use the appropriate barrier, then the data sharing may fail.

Using the right barrier to prevent compiler and CPU reordering for your platform is critical.

One of the advantages of using the synchronization primitives provided by the operating system is that all of them include the appropriate memory barriers.

Preventing Compiler Reordering

A compiler's job is to aggressively optimize your code in order to improve performance. This includes rearranging instructions wherever it is helpful and wherever it will not change behavior. Because the C++ Standard never mentions multithreading, and because the compiler doesn't know what code needs to be thread-safe, the compiler assumes that your code is single-threaded when deciding what rearrangements it can safely do. Therefore, you need to tell the compiler when it is not allowed to reorder reads and writes.

With Visual C++ you can prevent compiler reordering by using the compiler intrinsic **_ReadWriteBarrier**. Where you insert **_ReadWriteBarrier** into your code, the compiler will not move reads and writes across it.

```
#if _MSC_VER < 1400
    // With VC++ 2003 you need to declare _ReadWriteBarrier
    extern "C" void _ReadWriteBarrier();
#else
    // With VC++ 2005 you can get the declaration from intrin.h
    #include <intrin.h>
#endif
// Tell the compiler that this is an intrinsic, not a function.
#pragma intrinsic(_ReadWriteBarrier)

// Create a new sprite by filling in a previously empty entry.
g_sprites[nextSprite].x = x;
g_sprites[nextSprite].y = y;
// Write-release, barrier followed by write.
// Guarantee that the compiler leaves the write to the flag
// after all reads and writes that precede it in program order.
_ReadWriteBarrier();
g_sprites[nextSprite].alive = true;
```

In the following code, another thread reads from the sprite array:

```
// Draw all sprites.
for( int i = 0; i < numSprites; ++i )
{
    // Read-acquire, read followed by barrier.
    if( g_sprites[nextSprite].alive )
    {
        // Guarantee that the compiler leaves the read of the
flag
        // before all reads and writes that follow in program
order.
        _ReadWriteBarrier();
        DrawSprite( g_sprites[nextSprite].x,
                g_sprites[nextSprite].y );
    }
}
```

It is important to understand that **_ReadWriteBarrier** does not insert any additional instructions, and it does not prevent the *CPU* from rearranging reads and writes—it only prevents the *compiler* from rearranging them. Thus, **_ReadWriteBarrier** is sufficient when implementing a write-release barrier on x86 and x64 (because x86 and x64 do not reorder writes, and a normal write is sufficient for releasing a lock), but in most other cases, it is also necessary to prevent the CPU from reordering reads and writes.

_ReadWriteBarrier can also be used when writing to non-cacheable write-combined memory to prevent reordering of writes. In this case **_ReadWriteBarrier** helps to improve performance, by guaranteeing that the writes happen in the processor's preferred linear order.

It is also possible to use the **_ReadBarrier** and **_WriteBarrier** intrinsics for more precise control of compiler reordering. The compiler will not move reads across a **_ReadBarrier**, and it will not move writes across a **_WriteBarrier**.

Preventing CPU Reordering

CPU reordering is more subtle than compiler reordering. You can't ever see it happen directly, you just see inexplicable bugs. In order to prevent CPU reordering of reads and writes you need to use memory barrier instructions, on some processors. The all-purpose name for a memory barrier instruction, on Xbox 360 and on Windows, is **MemoryBarrier**. This macro is implemented appropriately for each platform.

On Xbox 360 **MemoryBarrier** is defined as **lwsync** (lightweight sync), also available through the **__lwsync** intrinsic, which is defined in ppcintrinsics.h. **__lwsync** also serves as a compiler memory barrier, preventing rearranging of reads and writes by the compiler.

The **lwsync** instruction is a memory barrier on Xbox 360 that synchronizes one processor core with the L2 cache. It guarantees that all writes before **lwsync** make it to the L2 cache before any writes that follow. It also guarantees that any reads that follow **lwsync** don't get older data from L2 than previous reads. The one type of reordering that it does not prevent is a read moving ahead of a write to a different address. Thus, **lwsync** enforces memory ordering that matches the default memory ordering on x86 and x64 processors. To get full memory ordering requires the more expensive **sync** instruction (also known as *heavyweight sync*), but in most cases this is not required. The memory reordering options on Xbox 360 are shown in the following table.

Xbox 360 Reordering	No sync	lwsync	sync
Reads moving ahead of reads	Yes	No	No
Writes moving ahead of writes	Yes	No	No
Writes moving ahead of reads	Yes	No	No
Reads moving ahead of writes	Yes	Yes	No

PowerPC also has the synchronization instructions **isync** and **eieio** (which is used to control reordering to caching-inhibited memory). These synchronization instructions should not be needed for normal synchronization purposes.

On Windows **MemoryBarrier** is defined in Winnt.h and gives you a different memory barrier instruction depending on whether you are compiling for x86 or x64. The memory barrier instruction serves as a full barrier, preventing all reordering of reads and writes across the barrier. Thus, **MemoryBarrier** on Windows gives a stronger reordering guarantee than it does on Xbox 360.

On Xbox 360, and on many other CPUs, there is one additional way that read-reordering by the CPU can be prevented. If you read a pointer and then use that pointer to load other data, then the CPU guarantees that the reads off of the pointer are not older than the read of the pointer. If your lock flag is a pointer and if all reads of shared data are off of the pointer, then the **MemoryBarrier** can be omitted, for a modest performance savings.

```
Data* localPointer = g_sharedPointer;
if( localPointer )
{
    // No import barrier is needed--all reads off of localPointer
    // are guaranteed to not be reordered past the read of
    // localPointer.
    int localVariable = localPointer->y;
    // A memory barrier is needed to stop the read of g_global
    // from being speculatively moved ahead of the read of
    // g_sharedPointer.
    int localVariable2 = g_global;
}
```

The **MemoryBarrier** instruction only prevents reordering of reads and writes to cacheable memory. If you allocate memory as PAGE_NOCACHE or PAGE_WRITECOMBINE, a common technique for device driver authors and for game developers on Xbox 360, then **MemoryBarrier** has no effect on accesses to this memory. Synchronization of non-cacheable memory is not needed by most developers and is beyond the scope of this document.

Interlocked Functions and CPU Reordering

Sometimes the read or write that acquires or releases a resource is done using one of the **Interlocked***Xxx* functions. On Windows this simplifies things, because on Windows, the **Interlocked***Xxx* functions are all full-memory barriers — they effectively have a CPU memory barrier both before and after them, which means that they are a full read-acquire or write-release barrier all by themselves.

On Xbox 360 the **Interlocked***Xxx* functions do *not* contain CPU memory barriers. They prevent compiler reordering of reads and writes but not CPU reordering. Therefore, in most cases when using **Interlocked***Xxx* functions on Xbox 360, you should precede or follow them with an **__lwsync**, to make them a read-acquire or write-release barrier. For convenience and for easier readability there are **Acquire** and **Release** versions of many of the **Interlocked***Xxx* functions. These come with a built-in memory barrier. For instance, **InterlockedIncrementAcquire** does an interlocked increment followed by an **__lwsync** memory barrier to give the full read-acquire functionality.

It is recommended that you use the **Acquire** and **Release** versions of the **Interlocked***Xxx* functions (most of which are available on Windows as well, with no

performance penalty) to make your intent more obvious and to make it easier to get the memory barrier instructions in the correct place. Any use of **Interlocked***Xxx* on Xbox 360 without a memory barrier should be examined very carefully, because it is often a bug.

This sample demonstrates how one thread can pass tasks or other data to another thread using the **Acquire** and **Release** versions of the **Interlocked***Xxx***SList** functions. The **Interlocked***Xxx***SList** functions are a family of functions for maintaining a shared singly linked list without a lock. Note that **Acquire** and **Release** variants of these functions are not available on Windows, but the regular versions of these functions are a full memory barrier on Windows.

```
// Declarations for the Task class go here.

// Add a new task to the list using lockless programming.
void AddTask( DWORD ID, DWORD data )
{
    Task* newItem = new Task( ID, data );
    InterlockedPushEntrySListRelease( g_taskList, newItem );
}

// Remove a task from the list, using lockless programming.
// This will return NULL if there are no items in the list.
Task* GetTask()
{
    Task* result = (Task*)
            InterlockedPopEntrySListAcquire( g_taskList );
    return result;
}
```

Volatile Variables and Reordering

The C++ Standard says that reads of volatile variables cannot be cached, volatile writes cannot be delayed, and volatile reads and writes cannot be moved past each other. This is sufficient for communicating with hardware devices, which is the purpose of the volatile keyword in the C++ Standard.

However, the guarantees of the standard are not sufficient for using volatile for multi-threading. The C++ Standard does not stop the compiler from reordering non-volatile reads and writes relative to volatile reads and writes, and it says nothing about preventing CPU reordering.

Visual C++ 2005 goes beyond standard C++ to define multi-threading-friendly semantics for volatile variable access. Starting with Visual C++ 2005, reads from volatile variables are defined to have read-acquire semantics, and writes to volatile variables are defined to have write-release semantics. This means that the compiler will not rearrange any reads and writes past them, and on Windows it will ensure that the CPU does not do so either.

It is important to understand that these new guarantees only apply to Visual C++ 2005 and future versions of Visual C++. Compilers from other vendors will generally implement different semantics, without the extra guarantees of Visual C++ 2005. Also, on Xbox 360 the compiler does *not* insert any instructions to prevent the CPU from reordering reads and writes.

Example of a Lock-Free Data Pipe

A *pipe* is a construct that lets one or more threads write data that is then read by other threads. A lockless version of a pipe can be an elegant and efficient way to pass work from thread to thread. The DirectX SDK supplies **LockFreePipe**, a single-reader, single-writer lockless pipe that is available in DXUTLockFreePipe.h. The same **LockFreePipe** is available in the Xbox 360 SDK in AtgLockFreePipe.h.

LockFreePipe can be used when two threads have a producer/consumer relationship. The producer thread can write data to the pipe for the consumer thread to process at a later date, without ever blocking. If the pipe fills up, then writes fail, and the producer thread will have to try again later, but this would only happen if the producer thread is ahead. If the pipe empties, then reads fail, and the consumer thread will have to try again later, but this would only happen if there is no work for the consumer thread to do. If the two threads are well-balanced, and the pipe is big enough, then the pipe lets them smoothly pass data along with no delays or blocks.

Xbox 360 Performance

The performance of synchronization instructions and functions on Xbox 360 will vary depending on what other code is running. Acquiring locks will take much longer if another thread currently owns the lock. **InterlockedIncrement** and critical section operations will take much longer if other threads are writing to the same cache line. The contents of the store queues can also affect performance. Therefore, all of these numbers are just approximations, generated from very simple tests:

- **lwsync** was measured as taking 33-48 cycles.
- **InterlockedIncrement** was measured as taking 225-260 cycles.
- Acquiring or releasing a critical section was measured as taking about 345 cycles.
- Acquiring or releasing a mutex was measured as taking about 2350 cycles.

Windows Performance

The performance of synchronization instructions and functions on Windows vary widely depending on the processor type and configuration, and on what other code is running. Multi-core and multi-socket systems often take longer to execute synchronizing instructions, and acquiring locks take much longer if another thread currently owns the lock.

However, even some measurements generated from very simple tests are helpful:

- **MemoryBarrier** was measured as taking 20-90 cycles.
- **InterlockedIncrement** was measured as taking 36-90 cycles.
- Acquiring or releasing a critical section was measured as taking 40-100 cycles.
- Acquiring or releasing a mutex was measured as taking about 750-2500 cycles.

These tests were done on Windows XP on a range of different processors. The short times were on a single-processor machine, and the longer times were on a multi-processor machine.

While acquiring and releasing locks is more expensive than using lockless programming, it is even better to share data less frequently, thus avoiding the cost altogether.

Performance Thoughts

Acquiring or releasing a critical section consists of a memory barrier, an **Interlocked***Xxx* operation, and some extra checking to handle recursion and to fall back to a mutex, if necessary. You should be wary of implementing your own critical section, because spinning in a loop waiting for a lock to be free, without falling back to a mutex, can waste considerable performance. For critical sections that are heavily contended but not held for long, you should consider using **InitializeCriticalSectionAndSpinCount** so that the operating system will spin for a while waiting for the critical section to be available rather than immediately deferring to a mutex if the critical section is owned when you try to acquire it. In order to identify critical sections that can benefit from a spin count, it is necessary to measure the length of the typical wait for a particular lock.

If a shared heap is used for memory allocations — the default behavior — then every memory allocation and free involves acquiring a lock. As the number of threads and the number of allocations increases, performance levels off, and eventually starts to decrease. Using per-thread heaps, or reducing the number of allocations, can avoid this locking bottleneck.

If one thread is generating data and another thread is consuming data, then they may end up sharing data frequently. This can happen if one thread is loading resources and another thread is rendering the scene. If the rendering thread references the shared data on every draw call, then the locking overhead will be high. Much better performance can be realized if each thread has private data structures which are then synchronized once per frame or less.

Lockless algorithms are not guaranteed to be faster than algorithms that use locks. You should check to see if locks are actually causing you problems before trying to avoid them, and you should measure to see if your lockless code actually improves performance.

Platform Differences Summary

- **Interlocked***Xxx* functions prevent CPU read/write reordering on Windows, but not on Xbox 360.
- Reading and writing of volatile variables using Visual Studio C++ 2005 prevents CPU read/write reordering on Windows, but on Xbox 360, it only prevents compiler read/write reordering.
- Writes are reordered on Xbox 360, but not on x86 or x64.
- Reads are reordered on Xbox 360, but on x86 or x64 they are only reordered relative to writes, and only if the reads and writes target different locations.

Recommendations

- Prefer using locks when possible, because they are easier to use correctly.
- Avoid locking too frequently, so that locking costs do not become significant.
- Avoid holding locks for too long, in order to avoid long stalls.
- Use lockless programming when appropriate, but be sure that the gains justify the complexity.
- Use lockless programming or spin locks in situations where other locks are prohibited, such as when sharing data between deferred procedure calls and normal code.
- Only use standard lockless programming algorithms that have been proven to be correct.
- When doing lockless programming, be sure to use volatile flag variables and memory barrier instructions as needed.
- When using **Interlocked***Xxx* on Xbox 360 use the Acquire and Release variants.

References

MSDN Library. "volatile (C++)." *C++ Language Reference*.

http://msdn2.microsoft.com/en-us/library/12a04hfd.aspx

Vance Morrison. "Understand the Impact of Low-Lock Techniques in Multithreaded Apps." *MSDN Magazine*, October 2005.

http://msdn.microsoft.com/msdnmag/issues/05/10/MemoryModels/default.aspx

Lyons, Michael. "PowerPC Storage Model and AIX Programming." IBM developerWorks, 16 Nov 2005.

http://www-128.ibm.com/developerworks/eserver/articles/powerpc.html

McKenney, Paul E. "Memory Ordering in Modern Microprocessors, Part II." *Linux Journal*, September 2005. [This article has some x86 details]

http://www.linuxjournal.com/article/8212

Intel Corporation. "Intel® 64 Architecture Memory Ordering." August 2007. [Applies to both IA-32 and Intel® 64 processors.]

http://developer.intel.com/products/processor/manuals/index.htm

Advanced Micro Devices. "Multiprocessor Memory Access Ordering," section 7.2, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*." September 2007.

http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf

Niebler, Eric. "Trip Report: Ad-Hoc Meeting on Threads in C++." *The C++ Source*, 17 Oct 2006.

http://www.artima.com/cppsource/threads_meeting.html

Hart, Thomas E. 2006. "Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation." *Proceedings of the 2006 International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, April 2006.

http://www.cs.toronto.edu/~tomhart/papers/hart_ipdps06.pdf