# Introduction to Multithreaded Programming

*By Michael Austin*
*Software Design Engineer*
*Advanced Technology Group (ATG)*

*Published: August 6, 2004*

Imagine you are building a house by yourself. You have your toolbox, plenty of lumber, and the blueprints. It'll take you a long time, since you can only do one thing at a time, but eventually you will finish the house.

Now imagine your buddy is along to help. Your house will go up in half the time! Of course, there are problems; you only have one hammer, and your friend happens to be nailing the wall frame when you need to put the plywood down on the staircase. Later, you are laying the floor and he's in the way doing the plumbing. Welcome to the world of multithreading!

A multithreaded program runs several threads (you and your buddy) as a way to utilize multiple processors or run regular periodic tasks (he goes out regularly to get food and nails). Threads sometimes vie for resources (the hammer), or run into problems when simultaneously accessing or writing data (working on the same part of the house). Threads share static storage and heap space, but each has an independent stack and register set.

On single-CPU platforms such as Xbox, additional threads are mainly useful for things like audio and file streaming. For games, the main update loop is tied to 16 or 33 millisecond intervals, which often isn't frequent enough to maintain full resource utilization. A separate thread is an easy way to manage such things.

On a multiprocessor system, using multiple threads is the only way to fully utilize the system. Xbox 360 will have the equivalent of six processors (two simultaneous threads per core, with three cores). Although there are interactions between threads on the same core (for instance, they share L1 cache), for simplicity this white paper assumes the six threads are independent.

## Multithreading Challenges

With this additional power come several challenges. Synchronization control is required to ensure that a thread is not trying to write data to a memory location at the same time as another thread is reading or writing from it. Threads need to be able to communicate and signal each other when they are handing off work. If there is a shared resource like a global variable, the threads must make sure they don't step on each others toes when accessing it.

For example, look at the following code:

```
void Object::AddReference()
{
      ++m_dwRefs;
}
```

This looks pretty simple. If two threads happen to call **::AddReference** at the same time, they both would increment the reference count and everything would be fine, right? Unfortunately, things are not so simple. The assembly for this function will look something like:

```
(1) load m_dwRefs into register 3
(2) increment reg. 3
(3) store reg. 3 back into m_dwRefs
```

Threads can be executing anywhere in their code at any time. Look what happens if thread 2 is one step behind thread 1….

Assume *m_dwRefCount* starts out at 1.

- Thread 1 loads *m_dwRefs* into reg. 3 (thread 1, reg. 3 = 1)
- Thread 1 increments reg. 3 (thread 1, reg. 3 = 2)
- Thread 2 loads *m_dwRefs* into reg. 3 (thread 2, reg. 3 = 1)
- Thread 1 stores reg. 3 into *m_dwRefs* (*m_dwRefs* = 2)
- Thread 2 increments reg. 3 (thread 2, reg. 3 = 2)
- Thread 2 stores reg. 3 into *m_dwRefs* (*m_dwRefs* = 2)

Unfortunately, our reference count is now 2, not the expected 3. This particular problem is called a ”race condition.” Thread 1 is racing to update *m_dwRefs* before any other threads access it.

## Synchronization Primitives

To solve this and other threading problems, there are a variety of synchronization primitives provided by Win32 and available on Xbox 360:

- Critical sections ensure that only a single thread is executing a specific section of code at any one time.
- Mutexes are used to handle cross-process synchronization. On Xbox 360, it’s not worth the additional overhead, since games run in a single process.
- Interlocked intrinsic functions provide efficient atomic operations like incrementing integers.
- Events handle communication, such as signaling between threads when a task is complete.
- Timers allow a thread to sleep until a specific amount of time has passed.
- Wait functions cause the thread to stop consuming CPU cycles until a specific system event or custom event has occurred.

Critical sections and interlocked functions are the most lightweight and efficient synchronization primitives. For more information about threads and synchronization, see the ”Thread Overview” in the Xbox 360 SDK.

One of the essential functions provided by many of these primitives (interlocked functions, critical sections) is that of locking data or resources. When a thread needs exclusive access to an object, it calls a locking function. If the object is already locked, the thread will stop running until the object becomes available. If the object isn’t already locked, the thread will acquire the lock to stop other threads from accessing the object. When the thread is done with the object, it releases the lock and continues.

The code above, for instance, could be fixed using a simple locking mechanism.

```
void Object::AddReference()
{
    EnterCriticalSection( &m_cs );
    ++m_dwRefs;
    LeaveCriticalSection( &m_cs );
}
```

In this case, *cs* is a critical section object owned by the object. Because in this case we’re doing something simple (incrementing), it would be more efficient to use the **InterlockedIncrement** intrinsic:

```
void Object::AddReference()
{
    InterlockedIncrement( &m_dwRefs );
}
```

A more complicated problem that arises from using synchronization primitives is known as a deadlock. A deadlock occurs when two threads are both trying to lock two separate resources at once. For instance, suppose a game needs to access two separate objects, A and B, in order to run.

```
void Thread1Func()
{
    A->Lock();
    B->Lock();

    // do stuff

    B->Unlock();
    A->Unlock();
}
void Thread2Func()
{
    B->Lock();
    A->Lock();

    // do stuff

    A->Unlock();
    B->Unlock();
}
```

Imagine now that thread 1 acquires object A about the same time thread 2 acquires object B. Thread 1 will wait indefinitely for object B so it can continue, and thread 2 will wait indefinitely for object A. This is an example of deadlock.

Deadlocks can be handled in several ways. The easiest is to make sure that all resources are acquired in the same order. This is sometimes hard to enforce though, because, in practice, there can be a lot of code between **A->Lock()** and **B->Lock()**.

Wherever possible, avoid acquiring multiple locks at the same time. If your code locks up, you can check if it's a deadlock by breaking in the debugger and identifying what objects threads are waiting on.

Synchronization problems are hard to detect, a pain to isolate once you know there is a problem, and often require re-architecting to fix. However, by following good design principles, you can minimize the coding impact of multithreading.

- Encapsulate and isolate the threads as much as possible; minimize the amount of shared data and the time spent updating it.
- Ensure that all shared data that can be written to is protected with a synchronization primitive. Shared data includes global variables, class statics, function statics, and any objects that are referenced by multiple threads.
- Make sure each thread has a well-defined way of interfacing with other threads and interacting with shared data.

## Types of Multithreaded Architectures

You may already be familiar with one type of multithreaded architecture from the Xbox. The CPU and GPU operate relatively independently, and use synchronization primitives when they read to or write from shared non-constant data (for example, **VertexBuffer->Lock()** and **Unlock()**)

The CPU and GPU operation on the Xbox is an example of a multithreading architecture called *pipelined* architecture. In this case, you have a set of linearly connected components. Each component interfaces only with the components on either side, typically reading input from the component before it and writing to the component afterwards.

If you've worked on a rendering engine for Xbox, you've probably experienced many of the tricks and caveats of pipelined multithreaded programs; interface structures (vertex buffers, push buffers) need to be locked before being written to or read from. Buffers can be double-buffered to minimize the amount of time spent locked for a write.

Another type of architecture is the *contractor queue*. Imagine you have a for loop, iterating over 1000 independent objects. A primary thread acts as the task master, and creates a queue of 1000 objects. The worker threads then each take a set of iterations and run them in parallel. If a thread finishes early, it can request more work or terminate. At the end of the loop, the threads all combine back into the master thread. The parallelism of the pixel pipelines in the GPU is similar to this

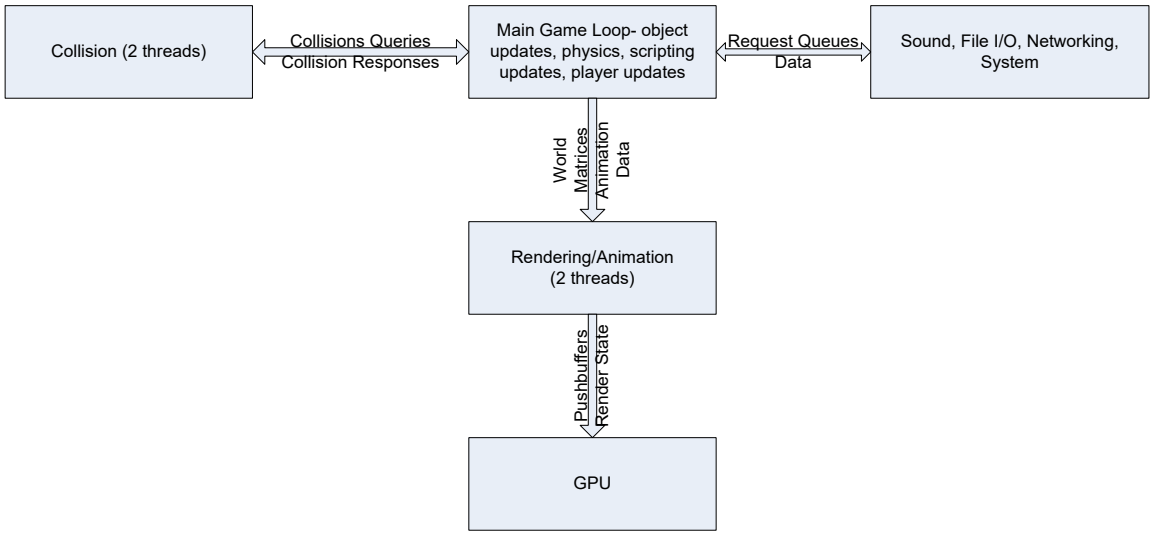
because pixels are generally independent, and as many threads as desired can split up the work.

The third main architecture type is the *work crew*. In this case, you have many threads, each working on independent tasks that all interoperate. An example is having a thread for sound or for file I/O. Each thread provides a service and has a specific detailed way for interacting, and wraps its own implementation of locking and unlocking resources.

A typical multithreaded game will have a mix of these three architectures. Working out thread architecture ahead of time is essential. When planning your game architecture, clearly define what global data each thread accesses, and make plans to synchronize those accesses. You should also build the general system for thread communication.

Here's an example of a complex multithreaded game:

1. A thread for the main loop. Game object updates and physics, AI, player input, and script updates would be run here (pretty much everything not covered by the other threads)
2. Sound, file I/O, and networking threads on a single shared processor. Because each of these subsystems requires a small amount of work on a frequent regular interval, they are good candidates for sharing a processor.
3. Two threads for collision. Because collision queries only read from the game state, the only interface would be the queries and responses. This is also a good place to use parallel threads to work on the queue at the same time.
4. Two threads for rendering and animation. One thread can do animation while the other builds object render state.



## OpenMP

The Xbox 360 SDK supports OpenMP 2.0. OpenMP is an easy way to tell the compiler to set up and optimize multithreaded code for you. The specification for OpenMP 2.0 is located at http://www.openmp.org/specs.

There are two types of OpenMP calls—pragmas and runtime calls. Pragmas give details about code that should be multithreaded. Runtime calls allow you to identify the current details of the multithreading state, with additional support for resource locking.

A few examples are included here so you can get a feel for OpenMP. The basic idea is that a single master thread executes, and then at a `#pragma omp parallel`, a team of worker threads is created that execute a given structured block in parallel. At the end of the structured block, the worker threads terminate and the main thread resumes. The final Xbox 360 implementation will have special extensions for specifying which processors additional threads will be run on.

**Example 1**

Traditional code:

```
for( int i = 0; i < 10000; ++i )
{
    UpdateObject( i );
}
```

OpenMP code:

```
#pragma omp parallel for
for( int i = 0; i < 10000; ++i )
{
    UpdateObject( i );
}
```

**Example 2**

Traditional code:

```
void TestCollisions( PotentialColliders, CollisionResults )
{
    for( ;; )
    {
        NextTest = PotentialColliders.dequeue();
        if (!NextTest )
            break;
        CollisionResult = TestCollide( NextTest );
        CollisionResults.push( CollisionResult )
    }
}
OpenMP code:
void TestCollisions( PotentialColliders, CollisionResults )
{
```

```
  #pragma omp parallel shared( PotentialColliders,
CollisionResults ) \
                      private ( NextTest, CollisionResult ) \
                      num_threads(2)
  {
      for( ;; )
      {
          #pragma omp critical
              NextTest = PotentialColliders.dequeue();
          if ( !NextTest )
              break;
          CollisionResult = TestCollide( NextTest );
          #pragma omp critical
              CollisionResults.push( CollisionResult )
      }
  }
}
```

The `#pragma omp parallel` directive creates the threads, and `#pragma omp critical` prevents multiple threads from executing the same line at once. Compilers that don't support OpenMP will ignore the pragmas. Platforms that don't have multiple cores will execute OpenMP code on a single thread. This allows for cross-platform compatibility as well as being a good fallback position for debugging.

OpenMP does not make your program thread safe outside of the automatic compiler-generated code. It merely creates a convenient abstraction for thread parallelization.

## Summary

- Use the synchronization functions to manage data shared between threads.
- Architect your game thread usage upfront.
- Use OpenMP to take advantage of threads in an already-architected engine.

## More Information

XDK topics: "Multithreading Considerations" and "Synchronization"

Cohen, Aaron, and Mike Woodring. *Win32 Multithreaded Programming*. O'Reilly and Associates, Sebastopol, CA, 1998.

Richter, Jeffrey. *Programming Applications for Windows.* http://www.openmp.org/