Choosing Appropriate Synchronization Primitives to Minimize Overhead

Abstract
When threads wait at a synchronization point, they are not doing useful work. Unfortunately, some degree of synchronization is usually necessary in multithreaded programs, and explicit synchronization is sometimes even preferred over data duplication or complex non-blocking scheduling algorithms, which in turn have their own problems. Currently, there are a number of synchronization mechanisms available, and it is left to the application developer to choose an appropriate one to minimize overall synchronization overhead.

This article is part of the larger series, "The Intel Guide for Developing Multithreaded Applications," which provides guidelines for developing efficient multithreaded applications for Intel® platforms.

Background
By nature, synchronization constructs serialize execution, limiting parallelism and potentially decreasing overall application performance. In reality, however, very few multithreaded programs are entirely synchronization-free. Fortunately, it is possible to mitigate some of the system overhead associated with synchronization by choosing appropriate constructs. This article reviews several available approaches, provides sample code for each, and lists key advantages and disadvantages.

Win32* Synchronization API
The Win32 API provides several mechanisms to guarantee atomicity, three of which are discussed in this section. An increment statement (e.g., \texttt{var++}) illustrates the different constructs. If the variable being updated is shared among threads, the load→write→store instructions must be atomic (i.e., the sequence of instructions must not be preempted before completion). The code below demonstrates how these APIs can be used:

```c
#include <windows.h>

CRITICAL_SECTION cs; /* InitializeCriticalSection called in main() */
HANDLE mtx; /* CreateMutex called in main() */
static LONG counter = 0;

void IncrementCounter ()
{
    // Synchronize with Win32 interlocked function
    InterlockedIncrement (&counter);

    // Synchronize with Win32 critical section
    EnterCriticalSection (&cs);
    counter++;
    LeaveCriticalSection (&cs);

    // Synchronize with Win32 mutex
    WaitForSingleObject (mtx, INFINITE);
```
counter++
    ReleaseMutex (mtx);
}

Compare these three mechanisms to illustrate which one could be more suitable in various synchronization scenarios. The Win32 interlocked functions (InterlockedIncrement, InterlockedDecrement, InterlockedExchange, InterlockedExchangeAdd, InterlockedCompareExchange) are limited to simple operations, but they are faster than critical regions. In addition, fewer function calls are required; to enter and exit a Win32 critical region requires calls to EnterCriticalSection and LeaveCriticalSection or WaitForSingleObject and ReleaseMutex. The interlocked functions are also non-blocking, whereas EnterCriticalSection and WaitForSingleObject (or WaitForMultipleObjects) block threads if the synchronization object is not available.

When a critical region is necessary, synchronizing on a Win32 CRITICAL SECTION requires significantly less system overhead than Win32 mutex, semaphore, and event HANDLEs because the former is a user-space object whereas the latter are kernel-space objects. Although Win32 critical sections are usually faster than Win32 mutexes, they are not as versatile. Mutexes, like other kernel objects, can be used for inter-process synchronization. Timed-waits are also possible with the WaitForSingleObject and WaitForMultipleObjects functions. Rather than wait indefinitely to acquire a mutex, the threads continue after the specified time limit expires. Setting the wait-time to zero allows threads to test whether a mutex is available without blocking. (Note that it is also possible to check the availability of a CRITICAL SECTION without blocking using the TryEnterCriticalSection function.) Finally, if a thread terminates while holding a mutex, the operating system signals the handle to prevent waiting threads from becoming deadlocked. If a thread terminates while holding a CRITICAL SECTION, threads waiting to enter this CRITICAL SECTION are deadlocked.

A Win32 thread immediately relinquishes the CPU to the operating system when it tries to acquire a CRITICAL SECTION or mutex HANDLE that is already held by another thread. In general, this is good behavior. The thread is blocked and the CPU is free to do useful work. However, blocking and unblocking a thread is expensive. Sometimes it is better for the thread to try to acquire the lock again before blocking (e.g., on SMP systems, at small critical sections). Win32 CRITICAL SECTIONS have a user-configurable spin-count to control how long threads should wait before relinquishing the CPU. The InitializeCriticalSectionAndSpinCount and SetCriticalSectionSpinCount functions set the spin-count for threads trying to enter a particular CRITICAL SECTION.

Advice

For simple operations on variables (i.e., increment, decrement, exchange), use fast, low-overhead Win32 interlocked functions.

Use Win32 mutex, semaphore, or event HANDLEs when inter-process synchronization or timed-waits are required. Otherwise, use Win32 CRITICAL SECTIONS, which have lower system overhead.

Control the spin-count of Win32 CRITICAL SECTIONS using the InitializeCriticalSectionAndSpinCount and SetCriticalSectionSpinCount functions. Controlling how long a waiting thread spins before relinquishing the CPU is especially important for small and high-contention critical sections. Spin-count can significantly impact performance on SMP systems and processors with Intel® Hyper-Threading Technology.
Intel® Threading Building Blocks Synchronization API

Intel® Threading Building Blocks (Intel® TBB) provides portable wrappers around atomic operations (template class `atomic<T>`) and mutual exclusion mechanisms of different flavors, including a wrapper around a "native" mutex. Since the previous section already discussed advantages and disadvantages of using atomic operations and operating system-dependent synchronization API, this section skips `tbb::atomic<T>` and `tbb::mutex`, focusing instead on fast user-level synchronization classes, such as `spin_mutex`, `queuing_mutex`, `spin_rw_mutex`, and `queuing_rw_mutex`.

The simplest mutex is `spin_mutex`. A thread trying to acquire a lock on a `spin_mutex` busy waits until it can acquire the lock. A `spin_mutex` is appropriate when the lock is held for only a few instructions. For example, the following code uses a mutex `FreeListMutex` to protect a shared variable `FreeList`:

```cpp
Node* FreeList;
typedef spin_mutex FreeListMutexType;
FreeListMutexType FreeListMutex;

Node* AllocateNode()
{
    Node* n;
    {
        FreeListMutexType::scoped_lock lock(FreeListMutex);
        n = FreeList;
        if( n )
            FreeList = n->next;
    }
    if( !n )
        n = new Node();
    return n;
}
```

The constructor for `scoped_lock` waits until there are no other locks on `FreeListMutex`. The destructor releases the lock. The role of additional braces inside the `AllocateNode` function is to keep the lifetime of the lock as short as possible, so that other waiting threads can get their chance as soon as possible.

Another user-level spinning mutex provided by Intel TBB is `queuing_mutex`. This also is a user-level mutex, but as opposed to `spin_mutex`, `queuing_mutex` is fair. A fair mutex lets threads through in the order they arrived. Fair mutexes avoid starving threads, since each thread gets its turn. Unfair mutexes can be faster than fair ones, because they let threads that are running go through first, instead of the thread that is next in line, which may be sleeping because of an interrupt. Queuing mutex should be used when mutex scalability and fairness is really important.

Not all accesses to shared data need to be mutually exclusive. In most real-world applications, accesses to concurrent data structures are more often read-accesses and only a few of them are write-accesses. For such data structures, protecting one reader from another one is not necessary, and this serialization can be avoided. Intel TBB reader/writer locks allow multiple readers to enter a critical region, and only a writer thread gets an exclusive access. An unfair version of busy-wait reader/writer mutex is `spin_rw_mutex`, and its fair counterpart is `queuing_rw_mutex`. Reader/writer mutexes supply the same `scoped_lock` API as `spin_mutex` and `queuing_mutex`, and in addition provide special functions to allow a reader lock to upgrade to a writer one and a writer lock to downgrade to a reader lock.
Advice

A key to successful choice of appropriate synchronization is knowing your application, including the data being processed and the processing patterns.

If the critical region is only a few instructions long and fairness is not an issue, then spin mutex should be preferred. In situations where the critical region is fairly short, but it’s important to allow threads access to it in the order they arrived to critical region, use queuing_mutex.

If the majority of concurrent data accesses are read-accesses and only a few threads rarely require write access to the data, it’s possible that using reader/writer locks will help avoid unnecessary serialization and will improve overall application performance.

Usage Guidelines

Beware of thread preemption when making successive calls to Win32 interlocked functions. For example, the following code segments will not always yield the same value for localVar when executed with multiple threads:

```c
static LONG N = 0;
LONG localVar;
...
InterlockedIncrement (&N);
InterlockedIncrement (&N);
InterlockedExchange (&localVar, N);

static LONG N = 0;
LONG localVar;
...
EnterCriticalSection (&lock);
localVar = (N += 2);
LeaveCriticalSection (&lock);
```

In the example using interlocked functions, thread preemption between any of the function calls can produce unexpected results. The critical section example is safe because both atomic operations (i.e., the update of global variable N and assignment to localVar) are protected.

For safety, Win32 critical regions, whether built with CRITICAL_SECTION variables or mutex HANDLEs, should have only one point of entry and exit. Jumping into critical sections defeats synchronization. Jumping out of a critical section without calling LeaveCriticalSection or ReleaseMutex will deadlock waiting threads. Single entry and exit points also make for clearer code.

Prevent situations where threads terminate while holding CRITICAL_SECTION variables, because this will deadlock waiting threads.

Additional Resources

Intel® Software Network Parallel Programming Community
