



Developing Platform Consistent Multithreaded Applications: Synchronization

March 2003

Copyright © Intel Corporation 2003

Terms of Use

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document or by the sale of Intel products. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel retains the right to make changes to its test specifications at any time, without notice.

The hardware vendor remains solely responsible for the design, sale and functionality of its product, including any liability arising from product infringement or product warranty.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, call (U.S.) 1-800-628-8686 or 1-916-356-3104.

The Pentium® III Xeon™ processors, Pentium® 4 processors and Itanium® processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel, Itanium, Pentium, VTune, and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Copyright © Intel Corporation 2003

Other names and brands may be claimed as the property of others.

Contents

1 Overview	4
Motivation.....	4
Prerequisites.....	4
Scope.....	4
Organization and Author Attribution.....	4
Series Conventions.....	5
4 Synchronization.....	6
4.1 Managing Lock Contention, Large And Small Critical Sections	7
4.2 Use Synchronization Routines Provided By The Threading API Rather Than Hand-Coded Synchronization.....	12
4.3 Win32 Atomics Versus User-Space Locks Versus Kernel Objects For Synchronization	15
4.4 Use Non-Blocking Locks When Possible.....	19
4.5 Use A Double-Check Pattern To Avoid Lock Acquisition For One-Time Events	23

FIGURES

Figure 1. Fundamental differences between interlocked functions and critical sections	17
---	----

1 Overview

Motivation

The objective of this series, which is comprised of this overview and four parts, is to provide guidelines for developing efficient multithreaded applications across Intel[®] architecture-based symmetric multiprocessors (SMP) and/or systems with Hyper-Threading Technology. An application developer can use the advice contained in this series to improve multithreading performance and to minimize unexpected performance variations on current as well as future SMP architectures built with Intel[®] processors.

This first version of this documentation provides general advice on multithreaded performance. Hardware-specific optimizations have deliberately been kept to a minimum. In future versions, topics covering hardware-specific optimizations will be added for developers who are willing to sacrifice portability for higher performance.

Prerequisites

Readers should have programming experience in a high-level language, preferably C, C++, and/or Fortran, though many of the recommendations in this document also apply to languages such as Java*, C#, and Perl. Readers must also understand basic concurrent programming and be familiar with one or more threading methods, preferably OpenMP*, POSIX threads (also referred to as Pthreads), or the Win32* threading API.

Scope

The main objective of these documents is to provide a quick reference to design and optimization guidelines for multithreaded applications on Intel[®] platforms. They are not intended to serve as a textbook on multithreading, nor do they represent a porting guide to Intel platforms.

Organization and Author Attribution

The “Developing Platform Consistent Threaded Applications” series covers topics ranging from general advice applicable to any multithreading method, to usage guidelines for Intel[®] software products, as well as API-specific issues. While designed as part of a series, each included chapter contains a discrete discussion of an important threading issue and can be read separately.

The included chapters and author attribution for each, are as follows:

Chapter	Scope	Contributing Authors
Chapter 1, Overview	This series overview	Bill Magro
Chapter 2, Intel® Software Development Products	This chapter describes how to use Intel software products to develop, debug, and optimize multithreaded applications	Bruce Greer, Clay Breshears, Judi Goldstein, Martyn Corden, Phil Kerly, and Vasanth Tovinkere
Chapter 3, Application Threading	This chapter covers general topics in parallel performance and occasionally refers to API-specific issues	Aaron Coday, Bill Magro, Clay Breshears, Henry Gabb, Prasad Kakulavarapu, Sanjiv Shah, and Vasanth Tovinkere
Chapter 4, Synchronization	The sections in this chapter discuss techniques to mitigate the negative impact of synchronization on performance	Grant Haab, Henry Gabb, Prasad Kakulavarapu and Vasanth Tovinkere
Chapter 5, Memory Management	Threads add another dimension to memory management that should not be ignored. This chapter covers memory issues that are unique to multithreaded applications	Clay Breshears, Jay Hoeflinger, Paul Petersen, and Phil Kerly

The user is free to download the entire series as a whole, or to download or read each chapter of interest as the need arises. Cross-references to related topics are provided throughout.

Series Conventions

‘This series’ refers to the above five chapters. Topics within chapters are referred to as ‘sections’. Cross-references are in outline notation, combining chapter and section numbers.

4 Synchronization

In order to avoid race conditions during the execution of a threaded application, mutual exclusion to shared resources is required to allow a single thread to access and change the state of shared resources. The shared resource can be a data structure, or memory in the address space. Minimizing synchronization overheads is a critical to application performance. This chapter discusses effective synchronization practices in multithreaded applications.

In multithreaded applications, while a thread is executing a code section that accesses a shared resource (critical section), competing threads are either spinning or waiting in a queue. In order to ensure fairness in scheduling control over the lock among all competing threads, it is important to minimize the time spent by a thread within a critical section. This usually means reducing code within the critical section to the bare minimum to process the state change. The first topic in this chapter addresses design issues for optimally sized critical sections.

The standard threading implementations provide synchronization primitives that are optimized for the architecture, and have been widely tested in varying application scenarios. Typically, these primitives include optimized spin-waits, efficient scheduling algorithms, and as result, minimal synchronization, and scheduling overheads. Further, the synchronization primitives with the standard threading implementations are portable, usually are forward and backward compatible, and can be easily migrated across platforms. The second section in this chapter discusses the benefits of using standard threading APIs in preference to hand-coded synchronization functions.

The Windows* multithreading API provides multiple synchronization primitives – critical section, mutex, semaphore, events, and interlocked operations. All of these primitives implement mutual exclusion but have varying performance benefits and usage models. A comparison of the different synchronization primitives is discussed in the next chapter, “Memory Management.”

Most threading implementations provide non-blocking threading primitives as a cost-effective alternative to their blocking counterparts. The non-blocking threading calls are reviewed next.

The final section of this chapter deals with the merits of using double-check pattern locks to minimize lock-acquisition costs for events that are executed only once such as initialization, file opening/closing, dynamic memory allocation, etc.

4.1 Managing Lock Contention, Large And Small Critical Sections

Category

Synchronization

Scope

General multithreading

Keywords

Lock contention, synchronization, spin-wait, critical section, lock size

Abstract

In multithreaded applications, locks are used to synchronize entry to regions of code that access shared resources. The region of code protected by these locks is called a critical section. While one thread is inside a critical section, no other thread can enter. Therefore, critical sections serialize execution. This topic introduces the concept of critical section size – the length of time a thread spends inside a critical section – and its effect on performance.

Background

Critical sections ensure data integrity when multiple threads attempt to access shared resources. They also serialize the execution of code within the critical section. Threads should spend as little time inside a critical section as possible to reduce the amount of time other threads sit idle waiting to acquire the lock, or lock contention. In other words, it is best to keep critical sections small. On the other hand, using a multitude of small, separate critical sections can quickly introduce system overheads, from acquiring and releasing each separate lock, to such a degree that the performance advantage of multithreading is negated. In this latter case, one larger critical section could be best. Scenarios illustrating when it is best to use large or small critical sections will be explored below.

The thread function in Example Code 1 contains two critical sections. Assume that the critical sections protect different data and that the work in functions `DoFunc1` and `DoFunc2` is independent. We shall also assume that the amount of time to perform either of the update functions is always very small. The critical sections are separated by a call to `DoFunc1`. If the threads only spend a small amount of time in `DoFunc1`, the synchronization overhead of two critical sections may not be justified. In this case, a better scheme might be to merge the two small critical sections into one larger critical section, as in Example Code 2. If the time spent in `DoFunc1` is much higher than the combined time to execute both update routines, this might not be a viable option because the increased size of the critical section increases the likelihood of lock contention, especially as the number of threads increases.

```

Begin Thread Function ()
  Initialize ()

  BEGIN CRITICAL SECTION 1
    UpdateSharedData1 ()
  END CRITICAL SECTION 1

  DoFunc1 ()

  BEGIN CRITICAL SECTION 2
    UpdateSharedData2 ()
  END CRITICAL SECTION 2

  DoFunc2 ()
End Thread Function ()

```

Example Code 1. A threaded function containing two critical sections to protect updates to different shared data

```

Begin Thread Function ()
  Initialize ()

  BEGIN CRITICAL SECTION 1
    UpdateSharedData1 ()
    DoFunc1 ()
    UpdateSharedData2 ()
  END CRITICAL SECTION 1

  DoFunc2 ()
End Thread Function ()

```

Example Code 2. A threaded function containing one critical section that protects updates to all shared data used by the function

Let's consider a variation of the previous example. This time, assume the threads spend a large amount of time in the `UpdateSharedData2` function. Using a single critical section to synchronize access to `UpdateSharedData1` and `UpdateSharedData2`, as in Example Code 2, is no longer a good solution because the likelihood of lock contention is higher. On execution, the thread that gains access to the critical section spends a considerable amount of time in the critical section, while all the remaining threads are blocked. When the thread holding the lock releases it, one of the waiting threads is allowed to enter the critical section and all other waiting threads remain blocked for a long time. Therefore, two critical sections, as in Example Code 1, is a better solution for this case.

It is good programming practice to associate locks with particular shared data. Protecting all accesses of a shared variable with the same lock does not prevent other threads from accessing different shared variables protected by a different lock. Consider a shared data structure. A separate lock could be created for each element of the structure, or a single lock could be created to protect access to the whole structure. Depending on the

computational cost of updating the elements, either of these extremes could be a practical solution. The best lock granularity might lie somewhere in the middle. For example, given a shared array, a pair of locks could be used: one to protect the even numbered elements and the other to protect the odd numbered elements.

In the case where `UpdateSharedData2` requires a substantial amount of time to complete, dividing the work in this routine and creating new critical sections may be a better option. In Example Code 3, the original `UpdateSharedData2` has been broken up into two functions operating on different data. It is hoped that using separate critical sections will reduce lock contention. If the entire execution of `UpdateSharedData2` did not need protection, rather than enclose the function call, critical sections inserted into the function at points where shared data are accessed should be considered.

```
Begin Thread Function ()
  Initialize ()

  BEGIN CRITICAL SECTION 1
    UpdateSharedData1 ()
  END CRITICAL SECTION 1

  DoFunc1 ()

  BEGIN CRITICAL SECTION 2
    UpdateSharedData2 ()
  END CRITICAL SECTION 2

  BEGIN CRITICAL SECTION 3
    UpdateSharedData3 ()
  END CRITICAL SECTION 3

  DoFunc2 ()
End Thread Function ()
```

Example Code 3. Separating one critical section into two can help reduce lock contention

Advice

Balance the size of critical sections against the overhead of acquiring and releasing locks. Consider aggregating small critical sections to amortize locking overhead. Divide large critical sections with significant lock contention into smaller critical sections. Associate locks with particular shared data such that lock contention is minimized. The optimum probably lies somewhere between the extremes of a different lock for every shared datum and a single lock for all shared data.

Remember that synchronization serializes execution. Large critical sections indicate that the algorithm has little natural concurrency or that data partitioning among threads is sub-optimal. Nothing can be done about the former except changing the algorithm. For the latter, try to create local copies of shared data that the threads can access asynchronously.

The previous discussion of critical section size and lock granularity does not take the cost of context switching into account. When a thread blocks at a critical section waiting to acquire the lock, the operating system swaps an active thread for the idle thread. This is known as a context switch. In general, this is the desired behavior because it releases the CPU to do useful work. For a thread waiting to enter a small critical section, however, a spin-wait may be more efficient than a context switch. The waiting thread does not relinquish the CPU when spin-waiting. Therefore, a spin-wait is only recommended when the time spent in a critical section is less than the cost of a context switch.

Example Code 4 shows a useful heuristic to employ when using the Win32* threading API. This example uses the spin-wait option on Win32 `CRITICAL_SECTION` objects. A thread that is unable to enter a critical section will spin rather than relinquish the CPU. If the `CRITICAL_SECTION` becomes available during the spin-wait, a context switch is avoided. The spin-count parameter determines how many times the thread will spin before blocking. On uniprocessor systems the spin-count parameter is ignored. Code Sample 4 uses a spin-count of 1000 for each thread in the application but a maximum spin-count of 8000.

```

int gNumThreads;
CRITICAL_SECTION gCs;

int main ()
{
    int spinCount = 0;
    ...
    spinCount = gNumThreads * 1000;
    if (spinCount > 8000) spinCount = 8000;
    InitializeCriticalSectionAndSpinCount (&gCs, spinCount);
    ...
}

DWORD WINAPI ThreadFunc (void *data)
{
    ...
    EnterCriticalSection (&gCs)
    ...
    LeaveCriticalSection (&gCs);
}

```

Example Code 4. Heuristic to control the behavior of waiting threads**Usage Guidelines**

The spin-count parameter used in Example Code 4 should be tuned differently on Intel® processors with Hyper-Threading Technology. In general, spin-waits are detrimental to Hyper-Threading Technology performance. Unlike true symmetric multiprocessors (SMP), which have multiple physical CPUs, Hyper-Threading Technology creates two logical processors on the same CPU core. Spinning threads and threads doing useful work must compete for logical processors. Thus, spinning threads can impact the performance of multithreaded applications to a greater extent on Hyper-Threaded systems compared to SMP systems. The spin-count for the heuristic in Example Code 4 should be lower or not used at all.

References

In this series, see also:

Intel® Software Development Tools, 2.4: Find Multithreading Errors With The Intel Thread Checker

Intel® Software Development Tools, 2.5: Using Thread Profiler To Evaluate OpenMP Performance

Memory Management, 5.2: Use Thread-Local Storage To Reduce Synchronization

4.2 Use Synchronization Routines Provided By The Threading API Rather Than Hand-Coded Synchronization

Category

Synchronization

Scope

General multithreading

Keywords

Synchronization, spin-wait, Hyper-Threading, Win32 threads, OpenMP, Pthreads

Abstract

Application programmers sometimes write hand-coded synchronization routines rather than using constructs provided by a threading API in order to reduce synchronization overhead or provide different functionality than existing constructs offer. Unfortunately, using hand-coded synchronization routines may have a negative impact on performance, performance tuning, or debugging of multi-threaded applications.

Background

It is often tempting to write hand-coded synchronization to avoid the overhead sometimes associated with the synchronization routines from the threading API. Another reason programmers write their own synchronization routines is that those provided by the threading API do not exactly match the desired functionality. Unfortunately, there are serious disadvantages to hand-coded synchronization compared to using the threading API routines.

One disadvantage of writing hand-coded synchronization is that it is difficult to guarantee good performance across different hardware architectures and operating systems. The following example is a hand-coded spin lock written in C that will help illustrate these problems:

```
#include <ia64intrin.h>
void acquire_lock( int *lock )
{
    while
        ( __InterlockedCompareExchange ( lock, TRUE, FALSE ) == TRUE );
}

void release_lock (int *lock)
{
    *lock = FALSE;
}
```

The `__InterlockedCompareExchange` compiler intrinsic is an interlocked memory operation which guarantees that no other thread can modify the specified memory location during its execution. It first compares the memory contents of the address in the first argument with the value in the third argument, and if a match occurs, stores the value in the second argument to the memory address specified in the first argument. The original value found in the memory contents of the specified address is returned by the intrinsic. In this example, the `acquire_lock` routine spins until the contents of the

memory location `lock` are in the unlocked state (`FALSE`) at which time the lock is acquired (by setting the contents of `lock` to `TRUE`) and the routine returns. The `release_lock` routine sets the contents of the memory location `lock` back to `FALSE` to release the lock.

Although this lock implementation may appear simple and reasonably efficient at first glance, it has several problems. First, if many threads are spinning on the same memory location, cache invalidations and memory traffic can become excessive at the point when the lock is released, resulting in poor scalability as the number of threads increases. Second, this code uses an atomic memory primitive which may not be available on all processor architectures, limiting portability. Third, the tight spin loop may result in poor performance for certain processor architecture features, such as Hyper-Threading Technology. Fourth, the `while` loop appears to the operating system to be doing useful computation, which could negatively impact the fairness of operating system scheduling. Although techniques exist for solving all these problems, they often complicate the code enormously, making it difficult to verify correctness. Tuning the code while maintaining portability is also difficult. These problems are better left to the authors of the threading API who have more time to spend verifying and tuning the synchronization constructs to be portable and scalable.

Another serious disadvantage of hand-coded synchronization is that it often decreases the accuracy of programming tools for threaded environments. For example, the [Intel® Threading Tools](#) need to be able to identify synchronization constructs in order to provide accurate information about performance (see Intel® Software Development Tools, 2.5: Using Thread Profiler To Evaluate OpenMP Performance) and correctness (see Intel Software Development Tools, 2.4: Find Multithreading Errors With The Intel Thread Checker) of the threaded application program. Threading tools are often designed to identify and characterize the functionality of the synchronization constructs provided by the supported threading API(s). Synchronization is difficult for the tools to identify and understand if standard synchronization API's are not used to implement it, which is the case in the example above. Sometimes tools support hints from the programmer in the form of tool-specific directives, pragmas, or API calls to identify and characterize hand-coded synchronization. Such hints, even if they are supported by a particular tool, may result in less accurate analysis of the application program than if threading API synchronization were used: the reasons for performance problems may be difficult to detect or threading correctness tools may report spurious race conditions or missing synchronization.

Advice

Avoid the use of hand-coded synchronization if possible. Instead, use the routines provided by your preferred threading API, such as `omp_set_lock/omp_unset_lock` or `critical/end critical` directives for OpenMP, `pthread_mutex_lock/pthread_mutex_unlock` for Pthreads, and `EnterCriticalSection/LeaveCriticalSection` or `WaitForSingleObject` or `WaitForMultipleObjects` and `ReleaseMutex` for the Win32 API. Study the threading API synchronization routines and constructs to find one that is appropriate for your application.

If a synchronization construct is not available that provides the needed functionality in the threading API, consider using a different algorithm for the program that requires less or different synchronization. Furthermore, expert programmers could build a custom synchronization construct from simpler synchronization API constructs instead of starting from scratch. If hand-coded synchronization must be used for performance reasons, consider using pre-processing directives to enable easy replacement of the hand-coded synchronization with a functionally equivalent synchronization from the threading API; thus increasing the accuracy of the threading tools.

Usage Guidelines

Programmers who build custom synchronization constructs from simpler synchronization API constructs should avoid using spin loops on shared locations to avoid non-scalable performance. If the code must also be portable, avoiding the use of atomic memory primitives is also advisable. The accuracy of threading performance and correctness tools may suffer because the tools may not be able to deduce the functionality of the custom synchronization construct, even though the simpler synchronization constructs from which it is built may be correctly identified.

References

In this series, see also:

Intel Software Development Tools, 2.4: Find Multithreading Errors With The Intel Thread Checker

Intel Software Development Tools, 2.5: Using Thread Profiler To Evaluate OpenMP Performance

This chapter, 4.3: Win32 Atomics Versus User-Space Locks Versus Kernel Objects For Synchronization

This chapter, 4.4: Use Non-Blocking Locks When Possible

See also:

John Mellor-Crummey, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,” *ACM Transactions on Computer Systems*, 9, 21-65, 1991.

Intel Pentium® 4 and Intel® Xeon™ Processor Optimization Reference Manual, Chapter 7: “Multiprocessor and Hyper-Threading Technology,” [Intel Developer Services](#).

4.3 Win32 Atomics Versus User-Space Locks Versus Kernel Objects For Synchronization

Category

Synchronization

Scope

Win32 multithreading

Keywords

Synchronization, lock contention, system overhead, mutual exclusion, Win32 threads

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. The Win32 API provides several synchronization mechanisms with varying utility and system overhead.

Background

Synchronization constructs, by their very nature, serialize execution. 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. An increment statement (e.g., `var++`) will be used to illustrate 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 Win32 API provides several mechanisms to guarantee atomicity, three of which are shown below:

```
#include <windows.h>

CRITICAL_SECTION cs; /* Initialized 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);
}
```

The advantages and disadvantages of each construct will now be discussed.

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. Though 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. Blocking and unblocking a thread is expensive, however. 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 the 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 CPUs with Hyper-Threading Technology.

Usage Guidelines

Beware of thread preemption when making successive calls to Win32 interlocked functions. For example, the two code segments in Figure 1 will not always yield the same value for `localVar` when executed with multiple threads. 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.

<pre> static LONG N = 0; LONG localVar; InterlockedIncrement (&N); InterlockedIncrement (&N); InterlockedExchange (&localVar, N); </pre>	<pre> static LONG N = 0; LONG localVar; EnterCriticalSection (&lock); localVar = (N += 2); LeaveCriticalSection (&lock); </pre>
---	--

Figure 1. Fundamental differences between interlocked functions and critical sections

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.

References

In this series, see also:

Intel Software Development Products, 2.3: Avoiding And Identifying False Sharing Among Threads With The VTune™ Performance Analyzer

This chapter, 4.2: Use Synchronization Routines Provided By The Threading API Rather Than Hand-Coded Synchronization

This chapter, 4.4: Use Non-Blocking Locks When Possible

See also:

Johnson M. Hart, *Win32 System Programming* (2nd Edition), Addison-Wesley, 2001

Jim Beveridge and Robert Wiener, *Multithreading Applications in Win32*, Addison-Wesley, 1997.

4.4 Use Non-Blocking Locks When Possible

Category

Synchronization

Scope

Windows threads, Pthreads, IA-32, Itanium[®] processor

Keywords

Non-blocking lock, synchronization, critical section, context switch, spin-wait

Abstract

Threads synchronize on shared resources by executing synchronization primitives offered by the supporting threading implementation. These primitives (such as mutex, semaphore, etc.) allow a single thread to own the lock, while the other threads either spin or block depending on their timeout mechanism. Blocking results in costly context-switch, whereas spinning results in wasteful use of CPU execution resources (unless used for very short duration). Non-blocking system calls, on the other hand, allow the competing thread to return on an unsuccessful attempt to the lock, and allow useful work to be done and thereby avoiding wasteful utilization of execution resources at the same time.

Background

Most threading implementations, including the Win32 and POSIX threads APIs provide both blocking and non-blocking thread synchronization primitives. Often the blocking primitives are used as default. When the lock attempt is successful, the thread gains control of the lock, and executes the code in the critical section. In the case of an unsuccessful attempt, however, a context-switch occurs and the thread is placed in a queue of waiting threads. A context-switch is costly and is avoidable for the following reasons:

- Context-switch overheads are considerable, especially if the threads implementation is based on kernel threads.
- Any useful work in the application following the synchronization call needs to wait execution until the thread gains control of the lock.

Using non-blocking system calls can alleviate the performance penalties. In this case, the application thread resumes execution following an unsuccessful attempt to lock the critical section. This avoids context-switch overheads and avoidable spinning on the lock. Instead, the thread performs useful work before a next attempt to gain control of the lock.

Advice

Use non-blocking synchronization functions to avoid context-switch overheads. Non-blocking synchronization functions usually start with ‘try.’ For instance, the Win32 API provides blocking and non-blocking critical sections:

```
void EnterCriticalSection (LPCRITICAL_SECTION cs);
bool TryEnterCriticalSection (LPCRITICAL_SECTION cs);
```

If the attempt to gain ownership of the critical section is successful, `TryEnterCriticalSection` returns the Boolean value `TRUE`. Otherwise, it returns `FALSE` and the thread can continue.

The following example shows a typical use of non-blocking synchronization:

```
CRITICAL_SECTION cs;

void threadfoo()
{
    while (TryEnterCriticalSection (&cs) == FALSE)
    {
        // Useful work
    }
    //
    // Code requiring protection by critical section
    //
    LeaveCriticalSection (&cs);
}
```

Similarly, Pthreads provides non-blocking versions of its mutex functions:

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

It is also possible to specify timeouts for Win32 synchronization primitives. The Win32 API provides the `WaitForSingleObject` and `WaitForMultipleObjects` functions to synchronize on kernel objects (i.e., `HANDLE`), e.g.:

```
DWORD WaitForSingleObject (HANDLE hHandle, DWORD dwMilliseconds);
```

where `hHandle` is the handle to the kernel object, and `dwMilliseconds` is the timeout interval after which the function returns if the kernel object is not signaled. A value of `INFINITE` indicates that the thread waits indefinitely. The thread waits until the relevant kernel object is signaled or a user-specified time interval has passed. Once the time interval elapses, the thread can resume execution. The following example demonstrates the use of `WaitForSingleObject` for non-blocking synchronization:

```
void threadfoo ()
{
    DWORD ret_value;
    HANDLE hHandle;
```

```
ret_value = WaitForSingleObject (hHandle, 0);
if (ret_value == WAIT_TIMEOUT)
{
    // Thread could not acquire lock within the time interval
    //
    // Other useful work
    //
}
else if (ret_value == WAIT_OBJECT_0)
{
    // Thread acquired lock within the time interval
    //
    // Code requiring protection by critical section
    //
}
}
```

Similarly, `WaitForMultipleObjects` allows the thread to wait on the signal status of multiple kernel objects.

Usage Guidelines

When using non-blocking synchronization, for instance `TryEnterCriticalSection`, verify the return value to see if the request is successful before releasing the shared object.

References

In this series, see also:

Intel Software Development Products, 2.3: Avoiding And Identifying False Sharing Among Threads With The VTune™ Performance Analyzer

This chapter, 4.2: Use Synchronization Routines Provided By The Threading API Rather Than Hand-Coded Synchronization

This chapter, 4.3: Win32 Atomics Versus User-Space Locks Versus Kernel Objects For Synchronization

This chapter, 4.4: Use Non-Blocking Locks When Possible

This chapter, 4.5: Use A Double-Check Pattern To Avoid Lock Acquisition For One-Time Events

See also:

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

Jim Beveridge and Robert Wiener, *Multithreading Applications in Win32 – the Complete Guide to Threads*, Addison Wesley, 1997.

Bil Lewis and Daniel J Berg, *Multithreaded Programming with Pthreads*, Sun Microsystems Press, 1998.

4.5 Use A Double-Check Pattern To Avoid Lock Acquisition For One-Time Events

Category

Synchronization

Scope

General multithreading

Keywords

Lock contention, synchronization, mutual exclusion, Win32 threads, Pthreads

Abstract

Acquiring locks, like synchronization, is an expensive operation. For one-time events (e.g., initialization, file opening/closing, dynamic memory allocation), it is often possible to use double-check locking (DCL) to avoid unnecessary lock acquisition.

Background

Synchronization, in this case lock acquisition, requires two interactions (i.e., locking and unlocking) with the operating system – an expensive overhead. When initializing a global, read-only table, for example, it is not necessary for every thread to perform the operation but every thread must check that the initialization occurred. For operations that are only executed once (e.g., initialization, file opening/closing, dynamic memory allocation), it is often possible to use DCL to avoid unnecessary lock acquisition. In DCL, if-tests are used to avoid locking after the first initialization, as the following pseudo-code illustrates:

```

Boolean initialized = FALSE

function InitOnce
{
    if not initialized
    {
        acquire lock
        if not initialized ← Double-check!
        {
            perform initialization
            initialized = TRUE
        }
        release lock
    }
}

```

There are several interesting points about this pseudo-code. First, multiple threads can evaluate the first if-test as true. However, only the first thread to acquire the lock may perform the initialization and set the Boolean variable to true. When the lock is released, subsequent threads re-check the Boolean variable. Failure to double-check the Boolean control variable can result in re-initialization, possibly with different data, which could lead to unexpected results. Second, threads that call the function after initialization has occurred do not acquire the lock. The first if-test evaluates to false. Third, no thread can

return unless the initialization is complete. Finally, a data race exists for the Boolean variable. Specifically, a thread can read its value while another thread is modifying its value. This data race is benign because only the thread holding the lock can modify the variable. However, the [Intel Thread Checker](#) will still report storage conflicts on the Boolean variable (see 2.4: Find Multithreading Errors With The Intel Thread Checker).

Advice

Use DCL to avoid repeated lock acquisition when performing one-time operations. DCL is especially useful when threads repeatedly check whether the operation is complete. The following source code shows one way to implement DCL using C and the Win32 API:

```
#include <windows.h>

CRITICAL_SECTION lock; /* Initialized elsewhere */
static volatile int initialized = 0;

void init_once ()
{
    if (!initialized)
    {
        EnterCriticalSection (&lock);
        if (!initialized)
        {
            /* Perform initialization */
            initialized = 1;
        }
        LeaveCriticalSection (&lock);
    }
}
```

The following source code shows how to implement DCL using C and Pthreads:

```
#include <pthread.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
static volatile int initialized = 0;

void init_once ()
{
    if (!initialized)
    {
        pthread_mutex_lock (&lock);
        if (!initialized)
        {
            /* Perform initialization */
            initialized = 1;
        }
        pthread_mutex_unlock (&lock);
    }
}
```

Usage Guidelines

When initializing shared, read-only data, it is tempting to let multiple threads perform the initialization asynchronously. The initialization will be correct provided the threads are all writing the same values to the global data. However, asynchronous initialization could incur a serious performance penalty as multiple threads invalidate each other's cache lines.

The `pthread_once` function can be used in the same situations as DCL, but it has greater system overhead.

DCL should be used with caution in Java because some Java Virtual Machines implement the Java Memory Model incorrectly.

References

In this series, see also:

Intel Software Development Products, 2.4: Find Multithreading Errors With The Intel Thread Checker

This chapter, 4.3: Win32 Atomics Versus User-Space Locks Versus Kernel Objects For Synchronization

This chapter, 4.4: Use Non-Blocking Locks When Possible

See also:

Douglas C. Schmidt and Tim Harrison, “Double-Checked Locking”, *Pattern Languages of Program Design 3* (Eds: Robert Martin, Frank Buschmann, and Dirke Riehle), Addison-Wesley, 1997.

Brian Goetz, “Double-check locking: Clever, but broken” JavaWorld, February 2001.

