Use Non-blocking Locks When Possible

<keywords: threading, 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, thereby avoiding wasteful utilization of execution resources at the same time.

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

Most threading implementations, including the Windows* and POSIX* threads APIs, provide both blocking and non-blocking thread synchronization primitives. The blocking primitives are often used as default. When the lock attempt is successful, the thread gains control of the lock and executes the code in the critical section. However, in the case of an unsuccessful attempt, a context-switch occurs and the thread is placed in a queue of waiting threads. A context-switch is costly and should be avoided 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 for 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, as well as avoidable spinning on the lock. Instead, the thread performs useful work before the next attempt to gain control of the lock.

Advice

Use non-blocking threading calls to avoid context-switch overheads. The non-blocking synchronization calls usually start with the `try` keyword. For instance, the blocking and non-blocking versions of the critical section synchronization primitive offered by the Windows threading implementation are as follows:

If the lock attempt to gain ownership of the critical section is successful, the `TryEnterCriticalSection` call returns the Boolean value of `True`. Otherwise, it returns `False`, and the thread can continue execution of application code.

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

Typical use of the non-blocking system call is as follows:

```
CRITICAL_SECTION cs;
void threadfoo()
{
  while(TryEnterCriticalSection(&cs) == FALSE)
  {
  // some useful work
  }
    // Critical Section of Code
    LeaveCriticalSection (&cs);
  }
  // other work
}
```

Similarly, the POSIX threads provide non-blocking versions of the `mutex`, `semaphore`, and `condition` variable synchronization primitives. For instance, the blocking and non-blocking versions of the `mutex` synchronization primitive are as follows:

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

It is also possible to specify timeouts for thread locking primitives in the Windows* threads implementation. The Win32* API provides the `WaitForSingleObject` and `WaitForMultipleObjects` system calls to synchronize on kernel objects. The thread executing these calls waits until the relevant kernel object is signaled or a user specified time interval has passed. Once the timeout interval elapses, the thread can resume executing useful work.

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

In the code above, `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. A code snippet demonstrating the use of this API call is included below.

```
void threadfoo ()
{
  DWORD ret_value;
  HANDLE hHandle;
    // Some work
  ret_value = WaitForSingleObject (hHandle,0);

  if (ret_value == WAIT_TIME_OUT)
{
    // Thread could not gain ownership of the kernel
    // object within the time interval;
// Some useful work
    }
    else if (ret_value == WAIT_OBJECT_0)
{
    // Critical Section of Code
    }
    else { // Handle Wait Failure}
    // Some work
    }
```

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

When using a non-blocking synchronization call, for instance, `TryEnterCriticalSection`, verify the return value of the synchronization call see if the request has been successful before releasing the shared object.

Usage Guidelines

Intel® Software Network Parallel Programming Community

Win32 Multithreaded Programming, Aaron Cohen and Mike Woodring.

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

Multithreaded Programming with Pthreads, Bil Lewis and Daniel J Berg.