Multi-threading for Experts: Inside a Parallel Application

By Sergey N. Zheltov and Stanislav V. Bratanov
Introduction

Anyone designing multi-threaded applications will agree: the problem of efficient synchronization is among the most difficult tasks of parallel programming. Modern operating systems provide a multitude of synchronization objects, which determine a variety of synchronization methods and schemes.

This article addresses some of the issues of parallel-program synchronization (and tries to clarify at least a few of them) including several synchronization schemes, synchronization object behavior and internal system implementation of user-visible synchronization objects or functions. Hidden timings and latencies are provided, execution thread layout is shown over time and non-obvious thread time shifts are pointed out.

You will also find code examples illustrating important points of the synchronization problem.

All discussions of synchronization objects and algorithms appearing in this article pertain to Microsoft Windows implementations of such objects and algorithms.

Internal Implementation

This article addresses wake-up and signaling schemes that are used to directly control the execution of threads. Schemes of mutual exclusion, which are mostly employed to synchronize resource usage (granting access to a resource to only one thread), are beyond the scope of this article.

Also, in this article we assume that all necessary resources have been already divided between threads; the task is to wake up the thread and load each of them with an appropriate amount of work.

As various mutual-exclusion constructions are not within the scope of this article, only two sorts of synchronization objects typically used for signaling are considered: events and semaphores.

From an operating-system kernel point of view, all objects involved in synchronization (those whose handles may be specified as arguments of WaitForSingleObject / WaitForMultipleObjects functions) are derived from one basic structure DISPATCHER_HEADER declared in Microsoft Driver Development Kit in the following form:

```c
typedef struct _DISPATCHER_HEADER {
    union {
        struct {
            UCHAR Type;
            UCHAR Absolute;
            UCHAR Size;
            union {
                UCHAR Inserted;
                BOOLEAN DebugActive;
            };
        };
        volatile LONG Lock;
    };
    LONG SignalState;
    LIST_ENTRY WaitListHead;
} DISPATCHER_HEADER;
```

The most important fields are Type, SignalState and WaitListHead that let the system differentiate between synchronization objects and organize a list of waiting threads which are to be woken up when the SignalState becomes Signaled.

So, event, semaphore, mutex, timer and thread objects look similar to the kernel.
The operating-system kernel supports two types of events corresponding to the user-level auto-reset and manual-reset events. Namely: synchronization events and notification events.

The difference between the two types is similar to the user-level events: signaling a synchronization event would cause only one waiting thread to wake up, while setting a notification event to a signaled state would wake as many threads as there might be waiting.

The functionality of a semaphore object is close to that of a notification event: when a semaphore is released, the operating system tries to satisfy as many waits (that is, to mark as many threads ready for execution) as possible, though staying within the limit specified upon creation of the semaphore object. The above mentioned limit is the only difference between semaphores and notification events.

**Thread Wake Algorithms**

The two basic methods of triggering the execution of a thread include:

- Waking a thread by signaling the synchronization object it waits on
- Queueing an APC to the context of an alertable thread

An attentive reader may think we forgot about `Sleep` function as well as about `SuspendThread` and `ResumeThread`. But the former function is equivalent to waiting for a timer and thus does not contradict our classification.

The latter suspension/resumption case is more complex, though still within the above classification: suspending a thread is implemented as a wait for an internal semaphore associated with each thread within the system. When `SuspendThread` function is called, the system queues an APC object, which in its turn enters a wait state (on the aforementioned semaphore). `ResumeThread` function releases the semaphore and the execution of the thread resumes.

The pseudo-code below exemplifies the operating-system kernel algorithms of waking up a waiting thread.

```
Event- or semaphore-driven algorithm from a general point of view looks as follows:

OnSetEvent(object)
{
    switch(object->type)
    {
        case SynchronizationEvent:
            new_thread = object->WaitList[0];
            processor = SelectIdleProcessor();
            SendIPI(processor, new_thread);
            break;

        case NotificationEvent:
            for(i = 0; i < object->WaitList.size; i++)
            {
                new_thread = object->WaitList[i];
                processor = SelectIdleProcessor();
                SendIPI(processor, new_thread);
            }
            break;

        case Semaphore:
            for(i = 0; i < min(object->WaitList.size, object->limit); i++)
            {
                new_thread = object->WaitList[i];
                processor = SelectIdleProcessor();
                SendIPI(processor, new_thread);
            }
    }
}
```

The APC-based thread-waking algorithm, as opposed to the event-driven scheme, comprises two stages.

First, an APC has to be queued to the context of a thread of interest:

```
QueueAPC(thread, function, type)
{
    If(type == kernel_mode_APC)
    {
        InsertKernelQueueAPC(thread, function);
    }
    else
    {
        InsertNormalQueueAPC(thread, function);
    }
    if(thread.state == waiting_in_user_mode_and_alertable)
    {
        processor = SelectIdleProcessor();
        SendIPI(processor, thread);
    }
}
```
Then, when a new thread is activated, all APC functions associated with the thread need to be executed:

```c
OnThreadSwitch(new_thread)
{
    if(new_thread->APC_queue.size != 0)
    {
        for(i = 0; i < new_thread->APC_queue.size; i++)
        {
            CallAPCFunction(new_thread->APC_queue.function[i])
        }
    }
}
```

You may observe that the execution of the destination thread's ‘real’ code is delayed until all the queued functions are executed. This, in our opinion, may impose additional overhead in case such a scheme is used for thread synchronization. On the other hand, when employed to notify a running or suspended thread on the completion of an input/output operation or expiration of a preset time interval, the APC approach seems to be one of the fastest possible solutions.

**Useful Timings**

Let’s prove or rebut the above considerations on the synchronization methods and their usability with the actual timing information. A sample OpenMP application built using different compilers (in order to reproduce at least two different synchronization models) and a manual thread control example were investigated. The results are shown below.

Figure 1 gives an overview of the sample application. The three regions separated by dashed lines correspond to APC-, event- and semaphore-triggered parts of code. Each part includes the main thread that controls the operation of three additional threads. The application is executed by a dual 2.5-GHz Intel® Xeon™ processor with Hyper-Threading Technology enabled. Each thread is represented by a horizontal color bar. Small strokes above the upper-right corner of some bars denote the fact that a corresponding thread was stopped due to a call to a synchronization function. (The absence of a stroke means that the thread was suspended due to expiration of its time quantum.) Similar small strokes below the bars correspond to special marks inserted into the code of the application by a programmer. Thin lines underscoring the bars show the duration of the synchronization functions (for example, `Sleep`, `WaitForSingleObject`, `QueueUserAPC`, `SetEvent` and `ReleaseSemaphore`).

Figure 2: APC Timings

Now we’ll proceed with the timings.

**APC Invocation Delay**

Figure 2 provides an example of an APC-based thread control. The main thread inserts three APC objects into the queues of each controlled thread. In the figure, the beginning of APC insertion and the start of the last thread are marked with dashed lines. The marked interval corresponds to 47 microseconds (as typed in the bottom-right corner of the figure), which is the APC invocation delay.
Event Triggering Delay

Figure 3 presents similar information for the event-based synchronization method. The execution region ranging from triggering the first event to the activation of the last thread covers 33 microseconds, which is faster than the time the APC version takes to execute.

Semaphore Release Delay

Figure 4 exemplifies the case when all the three threads are controlled with one semaphore. The marked region, again, covers the time from calling the `ReleaseSemaphore` function to the resumption of the last thread. The marked time interval corresponds to 32 microseconds (the difference of 1 microsecond between the semaphore- and event-triggering delays may be attributed to the error of measurements).

This timing information proves our assumptions on the internal algorithms involved in synchronization. As it was predicted, the event- and semaphore-based methods yield similar performance, and the double-staged APC queuing introduces additional overhead compared with events and semaphores.

Parallel Execution of Threads

Consider Figure 5. We marked with yellow the regions of code where the actual computation is performed. You will notice that no part of the computational code is executed in parallel. These results are slightly disappointing, because there is no guarantee that a properly threaded application outperforms its sequential analog.

All things considered, to ensure that at least part of an application runs in parallel, the following condition should hold true: the Sequential Time of an application divided by the Number of Processors should be greater than 50 microseconds for the worst case (see Figure 2).

Now the question: is there a method to trigger multiple threads simultaneously?
A closer look at Figure 4 would reveal a discrepancy between the expected behavior of threads (as the type of a user-level synchronization object used would suggest) and the actual operation. Indeed, if three threads are waiting for the same synchronization object, one expects them to wake up synchronously as that object acquires a signaled state. But the figure looks exactly as if each thread was triggered independently.

This only confirms the above hypothetical algorithms of operating-system kernel synchronization. As one can see, when releasing a semaphore, the operating system picks a new thread from a wait list (associated with the semaphore), selects an idle processor and then signals to the processor (by means of inter-processor interrupt) to start executing the new thread. This procedure is repeated until the wait list is empty or the semaphore limit is reached. As the wait list is analyzed on one processor and other processors are invoked sequentially, the delay between threads becomes greater or equal to the time of wait-list processing plus the time of sending and receiving an inter-processor interrupt.

Currently, although there is no means to enable synchronous thread start (at least in the operating systems considered in this article), it may be easily amended:

```c
OnSetEvent(object)
{
  ...
  if(object->type == Semaphore)
  {
    min_limit = min(object->WaitList.size,
                    object->limit, NumberOfProcessors - 1);
    SendIPI(All_Excluding_Self, object->WaitList, min_limit);
    for(i = min_limit; i < min(object->WaitList.size, object->limit); i++)
    {
      new_thread = object->WaitList[i];
      new_thread->state = Ready;
    }
  }
}
```

Here we suggest using advanced inter-processor interrupt (IPI) control features of modern processors which allow shorthand to be specified as an IPI destination. For example, specifying the ‘All_Excluding_Self’ shorthand would cause an IPI to be sent to all active processors but the current one. This will enable the simultaneous start of all waiting threads and, thus, reduce the sequential part of an application. While this suggestion will work for operating-system writers, other programmers should keep in mind that the biggest component of thread synchronization overhead is not the execution time of system functions, but the time lag between threads being synchronized. And the time lag is proportional to the number of threads or processors (as a computational thread is supposed to occupy a separate processor).
**Conclusion**

A variety of synchronization objects and schemes exist, rooted in the basic support of the operating-system kernel. The actual efficiency of different user-level synchronization schemes may appear to be the same, as such schemes may serve as mere synonyms of one general kernel mode construction.

On the contrary, some synchronization schemes may support a very specific case; and their performance in other cases of synchronization may degrade.

That is why it may be useful, while writing a well-balanced multi-threaded application, to keep in mind the above relationships between user- and kernel-based objects, and to only apply synchronization schemes that have been specifically designed for a particular purpose for the user.

This article provided examples of both the non-standard usage of a synchronization scheme, and the usage of different schemes having similar performance due to the same underlying system-kernel algorithm.

Non-evident timings of synchronization calls and time shifts between threads are also provided. Such timing information is extremely necessary for planning a well-threaded application, since not taking into account the thread-switching overhead and thread-scheduling order may adversely affect the scalability of the application, and even decrease its performance as compared with a sequential version.

A point that lacks operating-system support was identified: relatively large time shifts between threads that were triggered by a common semaphore. The solution with broadcast IPIs was proposed.

**In summary:**

- Use synchronization objects and schemes specifically designed by operating-system manufacturers to suit your task (it is very likely that such schemes are already optimized for solving your type of synchronization problems).
- Avoid redundant API calls (in a synchronization loop).
- Pay attention to the amount of work you are going to load your threads with (it may not be worth it, considering the overhead).
- Try to compensate for the time shifts by delegating less work to lagging threads (especially for short threads).

**References**

http://msdn.microsoft.com/library/
http://www.orgon.com/w2k_internals/
http://www.sysinternals.com/
Microsoft Windows 2003 Driver Development Kit
Sven Schreiber. *Undocumented Windows 2000 Secrets: A Programmer’s Cookbook*
About the Authors

Sergey Zheltov is a project manager in Microprocessor Research, Intel Labs. His research interests include media compression and processing, software and platforms architecture, signal processing and high-order spectra. He received a Diploma in radio-physical engineering and MS degree in theoretical and mathematical physics from Nizhny Novgorod State University.

Stanislav Bratanov is a software engineer in Microprocessor Research, Intel Labs. His research interests include multi-processor software platforms, operating-system environments and platform-dependent media data coding. He graduated from Nizhny Novgorod State University, Russia.

Experience 64-bit computing on Intel® Architecture.

*Other names and brands may be claimed as the property of others.
This document and the information described in it are furnished for informational use only and subject to change without notice. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

THIS DOCUMENT, RELATED MATERIALS AND INFORMATION DESCRIBED HERIN ARE PROVIDED "AS IS" WITH NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION, OR SAMPLE. INTEL ASSUMES NO RESPONSIBILITY FOR ANY ERRORS CONTAINED IN THIS DOCUMENT AND HAS NO LIABILITIES OR OBLIGATIONS FOR ANY DAMAGES ARISING FROM OR IN CONNECTION WITH THE USE OF THIS DOCUMENT OR THE INFORMATION PROVIDED HERIN.

Intel may make changes to specifications, product descriptions and features, and plans at any time, without notice.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
Copyright © 2005, Intel Corporation. All rights reserved. Please Recycle 306773-001US