

Multithreading in Windows NT



CALIFORNIA SOFTWARE LABS

REALIZE YOUR IDEAS

California Software Labs

6800 Koll Center Parkway,
Suite 100 Pleasanton
CA 94566, USA.

Phone (925) 249 3000

Fax (925) 426 2556

info@cswl.com

<http://www.cswl.com>



Multithreading in Windows NT

A Technical Report

Technical Expertise Level : Intermediate
Requires knowledge of : Windows NT Architecture

INDEX

INTRODUCTION.....	3
PREEMPTIVE VS. NONPREEMPTIVE MULTITASKING	3
WHY MULTITHREAD? WHY NOT?	4
WIN32 THREADING & COM THREADING	5
APARTMENT THREADING MODEL (SINGLE THREADED APARTMENTS) .	6
FREE THREADING MODEL (MULTI THREADED APARTMENTS)	6
BOTH APARTMENT AND FREE THREADED MODEL	7
RENTAL THREADED APARTMENT MODEL	7
THE BIRTH OF A THREAD: CREATETHREAD.....	8
THREAD LOCAL STORAGE (TLS).....	8
UNSYNCHRONIZED MULTITHREADED APPLICATIONS.....	9
THREAD COMMUNICATION & SYNCHRONIZATION	11
THREAD COMMUNICATION & SYNCHRONIZATION.....	14
DATA CORRUPTION AND DEADLOCK.....	14

One of the major functional enhancements of the Win32® application programming interface (API) over the 16-bit Microsoft® Windows™ API is the introduction of threads. This article introduces multi-threading in Windows NT, with focus on how and when to use this feature. Although multithreading as such is more general and is applicable to future implementations of the Win32 API as well, this discussion focuses primarily on Windows NT. This article series attempts to give an understanding of many aspects of multithreading.

INTRODUCTION

Threads can be defined as "code sequences that run multitasked on individual stacks," "something that behaves absolutely unpredictably and differently each time I try to debug it," "a programming tool that models concurrency," or "a concept that helps designing modular and interleaved applications".

What this tells us is that there are many different aspects to multitasking. Microsoft® Windows™ NT™ responds totally differently to user interaction than Windows version 3.1—for example, while one application starts up, you can switch to another application and work with it (because the two applications execute different threads). This is because threads within the foreground application might utilize the machine so heavily that other processes or threads in the application are locked out. By assigning the task manager a higher priority than most applications,



Preemptive vs. Nonpreemptive Multitasking

This strategy to share the CPU is called preemptive multitasking and is different from the multitasking that Windows 3.1 performs between applications. The latter variation is called nonpreemptive multitasking and relies on an application voluntarily relinquishing control to the operating system before letting another

application execute. In a nonpreemptive multitasking scheme, the amount of time a task is allowed to run is determined by the task, whereas in a preemptive scheme, the time is determined by the operating system.

Note that the difference between those two flavors of multitasking can be a very big one—for example, under Windows 3.1, you can safely assume that no other application executes while a particular application processes one message. Under the multithreaded execution scheme of Windows NT, this is not true because an application may lose its timeslice while it is in the middle of processing a message. Thus, if your application relies on the assumption that things do not change in the middle of processing a message, it might break under Windows NT (for example, if it calls **FindWindow** and expects the returned window handle to be valid, even while processing the same message).

The other important difference between the way Windows NT and Windows 3.1 multitask is that under Windows 3.1 the smallest schedulable unit is an application instance (also known as a task), whereas under Windows NT, you can run multiple threads within the same process. One consequence of this is that multiple threads in the same application have access to the same address space and thus can share memory. This is both a blessing and a curse: a blessing because that makes it fairly straightforward for multiple threads to share data and communicate with each other; a curse because the task of synchronizing access to the data can be extremely



Why Multithread? Why Not?

When do we multithreading in our application—whenever there is a true case of background processing (that is, a sequence of code that does not require user interaction and can run independent of whatever happens in the foreground), multithreading is very likely to help your application respond and perform better. Also, any asynchronous work that needs to be done (such as polling on a serial

port) probably works much better in a dedicated thread than competing with the foreground task in the same thread of execution.

Note that in most cases it does not make sense to distribute to separate threads the input from and output to the user because, by definition, the user feeds data sequentially into the application and also receives output sequentially. Thus, the areas in which you are most likely to benefit from multithreading are kernel and general data manipulation rather than USER and GDI, although some elements of graphical processing (such as calculating coordinates for outputting objects) may



Win32 Threading & COM Threading

Win32 defines two types of threads. User-Interface threads and Worker threads. Each process in Win32 can have one or more User Interface and/or multiple Worker threads. User Interface threads have message-loops that receive events targetted to that window and are hence associated with one or more windows. Worker threads are used for background processing and are not associated with any window. User Interface threads always own one or more windows. Whenever there is an event for a particular window, the UI thread takes care of calling the appropriate method. Since the message loop is executed on the same UI thread regardless of the thread that sent the message, synchronization is guaranteed by Windows. You as the programmer need not write any special code for thread synchronization. However, if you are developing worker threads, it is your responsibility to handle thread synchronization and prevent deadlocks or racing conditions.

COM terminology to threads is slightly different. There are three types of threading models in COM. They are apartment threading, free threading and rental threading (introduced for MTS) The closest analogy to Win32 is that UI threads are

analogically similar to the **Apartment** threading model and worker threads are



Apartment Threading Model (single threaded apartments)

This model was introduced in the first version of COM with Windows NT3.51 and later Windows 95. The apartment model consists of a multithreaded process that contains only one COM object per thread. Single Threaded Apartments (STA)- This also means that each thread can be called an apartment and each apartment is single threaded. All calls are passed through the Win32 message processing system. COM ensures that these calls are synchronized. Each thread has its own apartment or execution context and at any point in time, only one thread can access this apartment. Each thread in an apartment can receive direct calls only from a thread that belongs to that apartment. The call parameters have to be marshalled between apartments. COM handles marshalling between apartments through the Windows



Free Threading Model (multi threaded apartments)

This model was introduced with Windows NT 4.0 and Windows95 with DCOM. The free threaded model allows multiple threads to access a single COM object. Free threaded COM objects have to ensure thread synchronization and they must implement message handlers which are thread aware and thread safe. Calls may not be passed through the Win32 messaging system nor does COM synchronize the calls, since the same method may be called from different processes simultaneously. Free threaded objects should be able to handle calls to their methods from other threads at any time and to handle calls from multiple threads simultaneously. Parameters are passed directly to any thread since all free threads reside in the same apartment. These are also called Multi-Threaded Apartments (MTA)



Both Apartment and Free Threaded Model

It is possible for a process to have both the apartment and free threaded model. The only restriction is that you can have only one free threaded apartment but you can have multiple single threaded apartments. Interface pointers and data have to be marshalled between apartments. Calls to objects within the STAs will be



Rental Threaded Apartment Model

Components that use the Rental Threaded model (RTA), mark themselves as Free Threaded or Both. Each instance of a COM class can run on a different thread each time a method is called. When a thread is executing a method in a COM object, and that method creates a new object, MTS will suspend the current thread and create a new thread to handle the new object. Like the MTA, RTAs allow more than one thread to enter an apartment. However, once a thread has entered an apartment, it obtains an apartment-wide lock and no other thread can enter that apartment until it exits. This model was introduced into MTS to ensure that context switches are faster. The MTS COM object is however oblivious to the fact about multithreading and assumes that it is single threaded.

To provide programmers with more thread options and a more responsive and dynamic multithreading environment, the Win32 API supports four different kinds of thread categories or classes. There are normal threads for daily programming needs; idle-time threads, which are run only when the operating system has nothing else to do; the high-priority class, which is for important tasks like client-server functionality, and receives CPU time, even though it doesn't interact with the user; and time-critical threads, which are of the highest priority and guaranteed to receive CPU attention, regardless of how busy the system is. Time-critical threads should



The Birth of a Thread: CreateThread

An inevitable part of a thread is some code to execute. Under Windows NT, the address of a function is passed as a parameter to **CreateThread** to execute in the thread. The thread executes as long as it does not return from it. The thread can be terminated using **TerminateThread**. Each thread runs on a separate stack. To be more precise, each thread runs on either of two dedicated stacks—the kernel stack or the application stack—depending on whether system or application code executes in it, respectively, but the kernel stack is nothing that is ever visible in any form. Windows NT will dynamically grow the stack if necessary, but it will never grow it past 1 MB. This is to prevent infinitely recursive function calls from blowing up your process.

The `_beginthread` function is similar to the `Createthread` function in the Win32 API but has these differences:

- The `_beginthread` function lets you pass multiple arguments to the thread.
- The `_beginthread` function initializes certain C run-time library variables. This is important only if you use the C run-time library in your threads.
- `CreateThread` provides control over security attributes. You can use this function to start a thread in a suspended state

On the other hand, `_beginthread` is the preferred choice if you wish to work with variables and functions specific to the C run-time library, such as **errno** and



Thread Local Storage (TLS)

Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process may allocate locations in which to store thread-specific data.

Dynamically bound (run-time) thread-specific data is supported by way of the TLS API. Win32 and the Visual C++ compiler now support statically bound (load-time) per-thread data in addition to the existing API implementation. To support TLS, a new attribute, `thread`, has been added to the C and C++ languages and is supported by the Visual C++ compiler. This attribute is an extended storage class modifier, as described in the previous section. Use the `__declspec` keyword to declare a thread variable. For example, the following code declares an integer thread local variable and initializes it with a value:

```
__declspec( thread ) int tls_i = 1;
```



Unsynchronized Multithreaded Applications

Unsynchronized is the easiest form of multithreading. Let us look at a very small console application that does some data processing—it could be the front end of a database application. The application lets the user input 100 numeric values (for example, sales figures) and saves the values into a file. It then uses those values to compute data members of another file—let's say, to update a revenue spreadsheet. Once the update is done, the application goes back to asking for data from the user, who could be a data entry person.

The code for such an application typically looks something like this:

```
#include < stdio.h >
#include < windows.h > /* For the HANDLE type declaration and file API */
void main(void)
{ int iCount, iDataValue, iBytesWritten, iTemp;

    HANDLE hFreshFile, hOldFile;
    /* Step 1: Let the user input some data. */

    FreshFile = CreateFile("datafile",...)
```

```
for (iCount = 0; iCount<100; iCount++)
{ printf("Please enter next data item: ");
scanf("%d",&iDataValue);
WriteFile(hFreshFile,&iDataValue,sizeof(int),&iBytesWritten,NULL);
}
CloseHandle(hFreshFile);

/* Step 2: Process the data. */

FreshFile = CreateFile("datafile",...);
hOldFile = CreateFile("revenues.dat",...);

/* Let the following function do all of the data manipulation. */

UpdateRevenueFile(hOldFile,hFreshFile);

/* Step 3: Let the user enter more data. */
...
}
```

What happens here is that users will experience quite a delay between the time they type the first 100 values and the time they can go on typing. This distraction is not only unintuitive, but also fairly inefficient.

This is a primo case for introducing multithreading into the application. Analyzing the execution flow in this application, we will find that there are three sequential steps involved:

1. Entering the first 100 values.
2. Processing the values (must happen after Step 1).
3. Entering more values.

In this case, there is no reason why Steps 2 and 3 should not be executed at the same time: The program logic that updates the revenue file does not rely on the new data to be entered at all, and the program part that lets the user enter the subsequent data does not need to wait for the update routine to finish either. Chances are that under 16-bit Windows you have launched a background process (that is, a second task) to do the update while the main application goes back to asking the user for input. Windows, knowing how to execute two or more tasks in an interleaved fashion, divides up the CPU between the processes such that both



Thread Communication & Synchronization

The most elementary level of synchronization that two or more threads can have is waiting for each other to terminate. This happens frequently in practice. The Win32 API provides a set of wait functions to allow a thread to block its own execution.

There are three types of wait functions:

- single-object
 - SignalObjectAndWait
 - WaitForSingleObject
 - WaitForSingleObjectEx
- multiple-object
 - WaitForMultipleObjects
 - WaitForMultipleObjectsEx
 - MsgWaitForMultipleObjects
 - MsgWaitForMultipleObjectsEx
- alertable
 - MsgWaitForMultipleObjectsEx
 - SignalObjectAndWait
 - WaitForMultipleObjectsEx
 - WaitForSingleObjectEx



The wait functions do not return until the specified criteria have been met. The type of wait function determines the set of criteria used. When a wait function is called, it checks whether the wait criteria have been met. If the criteria have not been met, the calling thread enters an efficient wait state, consuming very little processor time while waiting for the criteria to be met.

Before returning, a wait function can modify the states of some types of synchronization objects. A synchronization object is an object whose handle can be specified in one of the wait functions to coordinate the execution of multiple threads.

Type	Description
Event	Notifies one or more waiting threads that an event has occurred.
Mutex	Can be owned by only one thread at a time, enabling threads to coordinate mutually exclusive access to a shared resource.
Semaphore	Maintains a count between zero and some maximum value, limiting the number of threads that are simultaneously accessing a shared resource.
Timer	Notifies one or more waiting threads that a specified time has arrived.
Critical Section	Similar to mutexes except that the objects can be used only by the threads of a single process.

Though available for other uses, the following objects can also be used for synchronization.

Object	Description
Change notification	Created by the FindFirstChangeNotification function, its state is set to signaled when a specified type of change occurs within a specified directory or directory tree.



Console input	Created when a console is created. The handle to console input is returned by the CreateFile function when CONIN\$ is specified, or by the GetStdHandle function. Its state is set to signaled when there is unread input in the console's input buffer, and nonsignaled when the input buffer is empty.
Process	Created when a new process is created by calling the CreateProcess function. Its state is set to nonsignaled while the process is running, and signaled when the process terminates.

The important aspect here is that we allow the threads to run interleaved and that the files have different sizes. The threads that work on the short files will terminate earlier than the ones that work on longer files, and as soon as the short threads terminate, the main thread can pick up the result and write it into the target file. Thus, obtaining the results of all the threads' computations can be interleaved with processing them.

If the application were set up such that we needed to wait for all threads to finish before processing the results, the objection would be valid, and we would not gain anything from multithreading. But because we can use the results from the individual threads once they come in, the main thread benefits from the interleaved execution and makes its progress along with the individual threads. Also, the individual threads might have side effects that aid application responsiveness. Although this approach imposes some additional problems to the application designer, the user will definitely benefit from multithreading because he or she sees



Thread Communication & Synchronization

Data corruption and Deadlock

The bulk of work in designing and implementing a multithreaded application lies in designing it such that concurrent, asynchronous access of multiple threads to shared data does not affect the desired functionality of the application. A big chunk of that task is theoretical work—namely, analyzing the behavior of the application without actually running it. This consists of two components:

- 1) Determining whether data can be corrupted by concurrent access from more than one thread, and if so, designing mechanisms to prevent it from happening (safety analysis); and
- 2) Determining whether the application will be alive (liveness analysis).

In order to prevent data corruption we need to identify all the variables that more than one thread can look at, and put together a matrix where the rows are labeled with the variables and the columns with the threads. The whole idea behind this matrix is to determine which thread can reach which variable. Because a variable can only be accessed from code that in C++ must reside inside a function, the first step is to list the functions there are in the entire application and which threads will call which functions. Because each thread starts its life span when its thread function is invoked, we can use that thread function as an anchor function (that is, a function to start scanning the code from) for each thread. The anchor on which the primary application thread rests is the WinMain function.

Now what the chart tells us is which threads will ever access which variables, and this information is useful for excluding some variables as candidates for access conflicts right away. Three intuitive rules can help us here:

1. Any variable that is only read from and never written to is safe.



2. Any variable that is only written to and never read from is safe.
3. Any variable that is accessed from one thread only is safe because the thread in itself runs sequentially and, presumably, knows how to make good use of its data.

For each of the remaining variables, we must analyze when, where, and how it gets accessed and whether that can cause us trouble or not and appropriately take care that using some suspend-and-wait mechanism.

Almost all mechanisms that can be established to prevent data corruption from happening work through a suspend-and-wait mechanism—that is, one thread that is about to work on some shared resource is suspended until another thread has finished using the resource in some manner. This scheme can lead to some undesired effects, though. It may happen that the two threads end up waiting for each other until doomsday (a so-called deadlock) or that two threads keep waking each other up in such a way that a third thread will never run (a so-called lockout). A multithreaded application is alive if no deadlocks or lockouts occur, but it needs to be proven that neither happens.

Next to data corruption a deadlock is the worst problem that can occur in a multithreaded application. A deadlock, very simply, is a condition in which two or more threads wait for each other to release a shared resource before resuming their execution. Most difficult part is, deadlocks are not always that easy to see. Using what is called a resource allocation graph (RAG) for any given state, it can be determined whether that state constitutes a deadlock or not. It is decidable whether a scenario that is identified as a deadlock can be reached given a certain initial state of the application.

To analyze and study concurrent systems there is an excellent framework called Petri Nets. Petri nets are very general and can be employed to model much more than multithreaded applications; for example, computer hardware or industrial



processes may be depicted as Petri nets. A Petri net is a directed graph which can be used to simulate the dynamic behavior of an application. Discussing more about



Copyright Notice:

2002 California Software Labs. All rights Reserved. The contents on the document are not to be reproduced or duplicated in any form or kind, either in part or full, without the written permission of California Software labs. Product and company names mentioned here in are the trademarks of their respective companies.