# Multithreading for Rookies

Ruediger R. Asche
Microsoft Developer Network Technology Group

Created: September 24, 1993

## Abstract

One of the major functional enhancements of the Win32® application programming interface (API) over the 16-bit Microsoft® Windows™ API is the introduction of multiple threads. Whereas Helen Custer's *Inside Windows NT* (Microsoft Press, 1992) provides an excellent overview of the implementation of threads as part of the system architecture of Windows NT™, this article focuses on some practical applications of multithreading. At the time the article was written, the only implementation of the Win32 API that fully supported the thread functionality was provided by Windows NT; thus, this discussion focuses primarily on Windows NT, although multithreading as such is more general and is applicable to future implementations of the Win32 API as well.

This article is the first in a series that describes the Win32 API's approach to multithreading. It covers the two simplest levels of multithreading: unsynchronized multithreading and termination waiting. Future articles will deal with synchronization mechanisms, multithreading in the Win32 graphical subsystem, and practical applications of multithreading.

This article is fairly light reading. I suggest reading it before going to bed, while flossing your teeth, or during the commercials in "Saturday Night Live." It contains a number of code fragments that are meant to be conceptual rather than segments to be cut and pasted into an existing application. I deliberately did not implement full-fledged samples because I wanted to keep everything as short and readable as possible. In certain parts I refer to existing samples that you can use to study the real-life behavior of multithreaded applications.

Also, I left out the error checking on most function calls in the code fragments included here. In a real-life application, this is a cardinal sin because an error condition that is not caught may seriously affect the offending application or, in pathological cases, the entire system. You should double-check that your application handles all possible error and failure conditions. Although this seems like pretty trivial advice, keep in mind that multiple threads can cause errors to occur intermittently (that is, a problem may occur only once in possibly many invocations of a multithreaded application), or it might manifest itself in several ways, depending on the interleaved sequence of statements executed in the multiple threads. Thus, you benefit from any additional degree of stability you build into your application.

## Introduction

So what exactly is a thread? There are about 27,000 different answers to this question, and everybody who writes another article (or book) about threads rejects all of them, coming up with yet another definition of a thread, which is probably as good as any of the other ones. Depending on whether you like to see threads from the implementation level, from the perspective of an application's programmer, from a conceptual angle, or from a software designer's view, you can define threads 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," respectively.

What this tells us is that there are many different aspects to multitasking. The dark side of this profound statement is that it is almost impossible to develop multithreaded applications without understanding all of these aspects. For example, if you are a program manager, in order to decide where your application can benefit from multithreading (if at all), you need to be familiar mainly with concurrent program design schemes, but you also need to know how threads are scheduled, what scope a thread runs under, and what data it has access to (which is mainly an implementation issue). On the other hand, if you are a programmer, to implement a solid multithreaded application you need to be familiar with synchronization mechanisms and correctness analysis theory, as well as debugging strategies and implementation details.

As a user of a multithreaded application, you might not be aware of its "multithreadedness," but you will already have noticed that 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), or even when the system seems to be stalled altogether, you can always bring up the task manager. 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, Windows NT ensures that it gets to preempt applications' threads if necessary.

This article series attempts to give you an understanding of all the aspects of multithreading, while still being useful and practical—so let us dive into the subject matter right away.

# 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. */

  hFreshFile = 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. */

hFreshFile = 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 will make some progress over time without explicitly calling each other.

Multithreading is a similar concept. By spawning a new thread, Windows NT can be asked to do "something else" at the same time your application goes on doing whatever it wants to do.

What does "at the same time" mean? That actually depends on the hardware you run on. If your Windows NT machine has more than one processor, it may indeed mean that a second processor picks up executing the background thread and running at the very same time the first processor goes on executing the application. If your machine has only one main processor (as the majority of computers currently on the market do), Windows NT will "timeslice" the threads—that is, give one thread a little time of the CPU, then switch to another thread, let it execute for a while, and so on, until the first thread has its turn again and does some computation, and so on.

In other words, the threads are "chopped up" and served to the CPU in an interleaved fashion. The trick here is to make each of the threads believe that it runs exclusively, while in reality it only runs for a little while, freezes, and thaws later on when it runs for another little while. This task is being taken care of by the operating system, which keeps what is called a *context record* for each thread. A context record is a collection of the data that a thread must preserve in order to later pick up execution in the same state it was in before losing control over the CPU. The maintenance of context records is done by the operating system in such a manner that the thread will never know about it.

## 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. I like

to formulate the difference as follows: 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 difficult.

## Why Multithread? Why Not?

The work you need to put into spawning a new thread is next to nothing. The hard part is to make sure that multiple threads do not interfere with each other in an undesired way. That task can easily end up taking up more than 90 percent of the time you have for designing and debugging a multithreaded application. While fixing problems that arise due to synchronization problems, you may, in the worst case, need to implement mutual exclusion mechanisms that undo all of the efficiency and speed gains that arose from multithreading in the first place.

Then why would you want to put up with the hassle of introducing multiple threads to your application in the first place? The first example has already shown us a few cases in which an application can benefit from multiple threads —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. We will see another meaningful example of multithreading later on in this article, while the article "Using C++ and Multithreading to Generate Live Objects" shows yet another possibility for making interesting use of multiple threads.

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 be optimized with the help of multithreading.

> **Hint**
>
> If you have programmed system-level software under Windows 3.1, there is a good chance that you have established a system-level timer to perform asynchronous I/O in the background—for example, to poll the status of an I/O device periodically or to implement timeout retries on network cards. Because Windows NT does not allow you to interact directly with hardware from an application, you will need to write a device driver to address the hardware directly and communicate between the device driver and the application via I/O device control calls. In most cases, a dedicated thread or, depending on the capabilities of the driver, an asynchronous I/O call can substitute for reliance on a timer. For details on asynchronous status reporting, please refer to the documentation for **ReadFileEx** and **WriteFileEx**.

The simplest variation of multithreading—unsynchronized multithreading—is very straightforward and simple. Here is a multithreaded version of the data processing application described above:

```
#include <stdio.h>
#include <windows.h> /* For the CreateThread prototype */

long WINAPI Validate(long); /* Function prototype */

HANDLE hFreshFile;
int iBytesWritten, iCount, iDataValue;

void main(void)
{ int iID;
   HANDLE hThread;

/* Step 1: Let the user input some data. */

  hFreshFile = 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);

/* Dispatch a thread that does Step 2 (Validate the data) for you. */

hThread = CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Validate,NULL,0,&iID);

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

long WINAPI Validate(long lParam)
{ HANDLE hOldFile;
  int iTemp;
hFreshFile = CreateFile("datafile",...)
hOldFile = CreateFile("revenues.dat",...);

< Munge the data here. >
}
```

This program will dispatch a background thread for you that does the validation while the user can keep entering data. The important call here is the **CreateThread** function that, when successful, will generate a second thread that executes the function **Validate** concurrently with the first thread.

The first thread? Which one is that? You never created a first thread in the first place, did you?

Yes, you did, even though you did not know until now. Any Windows NT process must have at least one thread to run, and the loader will automatically create the first thread for you. The first thread is the thread in which **main** (or

**WinMain** for GUI applications) executes.

## The Birth of a Thread: CreateThread

The previous paragraph already tells us something about threads: An inevitable part of a thread is some code to execute. Under Windows NT, you must pass the address of a function to execute in the thread as the third parameter to **CreateThread**. **CreateThread** expects this to be a function that is declared as **WINAPI** and takes and returns one parameter of type **long** each. It is entirely up to you what to pass to the routine and what to return from it. Windows NT will not look at the return value, but after termination, you can retrieve the value if needed. You pass the argument to the thread function as the fourth parameter to **CreateThread**. This may be anything you want; it is up to the spawner and the newly created thread to interpret it. Later on in this and in future articles, we will see some typical values for this parameter.

The second parameter to **CreateThread** is a stack size. This leads us to another important implementation detail of threads: 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 to you in any form.

The size of the application stack that you pass as the second parameter determines how big you want the thread's application stack to be; in our example, we pass in 0 to indicate to Windows NT that we want the stack to have the same size as the application's primary thread. Note that 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.

As a side note to OS/2® programmers: You will notice that Windows NT does not require that you allocate and deallocate the stack for the memory yourself as OS/2 does. Windows NT will allocate the memory for you in the virtual address space of the application that contains the thread.

This leaves us with three remaining parameters to the **CreateThead** call: the first, fifth, and sixth. The first parameter is a pointer to a **SECURITY_ATTRIBUTES** structure. All native objects under Windows are securable, and this parameter is the key to allowing other processes to access the thread. Security will be the focus of a future article, so for the time being, we will be content with leaving this parameter at NULL, which means that we do not wish to secure the thread specifically.

The fifth parameter to **CreateThread** is an integer that can take up either of two values: CREATE_SUSPENDED or 0. If it is CREATE_SUSPENDED, the thread will be created, but will not run unless it is explicitly resumed via the **ResumeThread** function. If it is 0, the thread will run as soon as it is created.

The sixth and last parameter, eventually, is the address of an integer variable that will receive the thread ID, a unique value that can be used to identify the thread. Please do not make any assumptions about this identifier; all that is guaranteed is that while the thread is running, no other thread will ever have the same ID. The ID may be recycled later on after the thread has terminated. Please refer to the article "Give Me a Handle, And I'll Show You an Object" for details.

Note that there is another method for creating a thread for a Win32-based application under Windows NT: The C run-time library function **_beginthread** will also create a thread for you. **_beginthread** is actually a fairly thin wrapper around **CreateThread** such that a thread created either way will be "just one more thread" as far as the Windows NT kernel is concerned. You might prefer **CreateThread** over **_beginthread** if you need the higher degree of control that **CreateThread** provides (for example, if you need to associate the thread with a security descriptor, need to create the thread suspended, need to resume and suspend the thread dynamically, need the returned handle to pass to other Win32 functions such as **DuplicateHandle**, or anything along those lines). 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 **_signal**.

This concludes our discussion of "threads for novices." I deliberately left out some of the issues that come to mind right away, such as "What happens if more than two threads try to read from the same input device?" Questions such as this will be addressed soon enough—don't worry.

Before we go on, however, I have to confess something to you. The application I just sketched out has a bug—once the secondary thread has terminated, we can't just forget about it, but instead must close the handle to the thread. Why this happens and how to handle it is part of the next chapter.

# Casual Communication: The Basics of Thread 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. Let us assume that there are 100 database files of different sizes. The main application's thread dispatches one thread for each file. Each of those 100 threads will search for all records that contain a specific string in its file and return the number of records that match the search string. The main thread will then write the respective search string, along with the name of the scanned file and the number of occurrences of the search string, as one record into a new database file.

You might ask why we would bother to create separate threads in this example in the first place. After all, given the market share of multiprocessor computers, the application is most likely to run on a single-processor machine, so instead of scanning the 100 files sequentially, the CPU is constantly switched between 101 threads, such that the total time to process all files does not change, right? Even worse, the process of chopping up the threads and switching the CPU between them introduces even more overhead, so what do we gain by multithreading here?

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—for example, in the application mentioned before, each thread might want to display every hit on the screen while scanning the file. Although this approach imposes some additional problems to the application designer (which we will elaborate on later), the user will definitely benefit from multithreading because he or she sees the hits as the threads encounter them.

By the way, the above discussion applies to files as well as memory blocks. In fact, you will probably want to map the files to memory using the file-mapping API and scan the memory instead of using the file API to read in small chunks of the file little by little.

## Go On, I'm Waiting...

How can we wait for threads, then? The solution is a set of Windows functions called **WaitForSingleObject** and **WaitForMultipleObjects**. Threads are *native objects* under Windows NT—that is, objects that are maintained by the Windows NT kernel in a uniform manner. One characteristic common to all native objects is that they inform the system of state changes. All native objects can be in either a "signaled" or "unsignaled" state. What this means exactly depends on the object types; for thread objects, the signaled state means that the thread has terminated (which happens explicitly when the **ExitThread** or **TerminateThread** function is called, or happens implicitly as soon as the thread function returns or the process that owns the thread terminates). When a thread is created, it is set to the unsignaled state.

Another characteristic of native objects is that they are referenced by handles; as soon as a process creates a new object or requests access to an existing object, it opens a handle to it. Opening a handle informs the Windows NT kernel that there are references to the object, and the object will not be removed from memory as long as there are pending references to it. Thus, a process must eventually release all its references to the objects it used using the **CloseHandle** function. Windows NT will close all handles to a process implicitly as soon as the process terminates, but you are well advised to close all handles that you opened as soon as you do not need the objects anymore. As soon as the last handle is closed, the object is destroyed.

Note that there is one peculiarity with threads in the handle/object model: You can close the last handle to a thread while the thread is running, and the thread will still be in the system! That is because the Windows NT kernel, whose responsibility it is to schedule threads for execution, runs independently from the object manager that assigns and maintains user-visible handles.

Thus, the following sequence consists of valid, working NT code that will also work on other platforms that support the Win32 API:

```
HANDLE hThread;
.
.
.
hThread=CreateThread(...);
if (!hThread)
      <Process error here>;
   else
   CloseHandle (hThread);
.
.
.
```

This code will spawn a secondary thread that will implicitly be removed from the system as soon as its thread function terminates. You may want to do this if you never need the handle again—that is, you never need to suspend or resume the thread, synchronize with it, secure it, or duplicate the handle for another process to access. In the rest of the article, we will assume, however, that you indeed need to keep the handle around.

The **WaitForSingleObject** function takes an object handle as the first parameter and will not return before the object that is referenced by the handle attains the signaled state or the timeout value that is specified as the second parameter elapses.

The **WaitForMultipleObjects** call is similar, except that its main parameter (the second) is an array of Windows NT object handles. The first parameter passed to it specifies the number of handles in this array. The third parameter is FALSE if waiting for *any* object is desired and TRUE if the function is to return only when *all* objects in the array have signaled.

The timeout parameter that both **WaitForSingleObject** and **WaitForMultipleObjects** are being passed should be used whenever there is a chance that a thread for which the functions wait will never terminate. For example, a thread might constantly listen to a serial communications line, try to connect to a named pipe, or try to find an remote procedure call (RPC) server on the net. In this case, a timeout parameter should be specified in case the serial communications line, the remote server, or remote process does not respond or times out due to hardware

conditions. A timeout handler in one of those conditions typically informs the user of a timeout condition and prompts for termination or retry of the connection.

Note that the handle array passed to **WaitForMultipleObjects** can be composed of an arbitrary mix of Windows NT objects. Thus, other than thread handles, the array could also contain handles to processes, mutexes, semaphores, events, change notifications, and console input objects. (We have not discussed these and will not do so in this article, which focuses on threads rather than objects in general.) There are a few catches with the **WaitForMultipleObjects** function that we will discuss when we look at the next code snippet, which is basically a modification of the thread dispatch-and-gather routine to be found in the GUIGREP application (see the MSDN Library), a sample distributed with the Windows NT SDK CD.

## Introducing GUIGREP

Analogous to the character-based GREP or FINDSTR utilities, GUIGREP searches a number of files for occurrences of a particular string, but it collects the hits in a list box for easy reference. GUIGREP is a fairly elaborate example for a multithreaded application; some of the pitfalls to be found there will be explained in this article, whereas other ones will be discussed in other articles in this series. The major difference between the shipping code for GUIGREP and this code fragment is that the shipping code does not process the results of the threads that process the file interleaved with the threads themselves.

To make the code a bit more readable, I have separated it into several sections that are consecutively numbered and labeled. I also simplified some function calls.

You may want to incorporate this code into GUIGREP to study the behavior of **WaitForMultipleObjects**:

```
long Poll_On_Threads(LONG lParam)
{

/* Section 0: local variables */

  int iLoop, iEndLoop,iThread,iArrayIndex;
  static HANDLE aThreads[MAX_CONCURRENT_THREADS];

/* Section 1: Consecutively dispatch all threads. */


/* iEndLoop is the number of files to process. We got that one from */
/* File Manager. */

  for (iLoop = 0; iLoop < iEndLoop; iLoop++)
   {HANDLE hNewThread;

      if (iLoop < MAX_CONCURRENT_THREADS)
                iArrayIndex = iLoop;
      else
         {
            iArrayIndex = WaitForMultipleObjects(
                     MAX_CONCURRENT_THREADS,aThreads,FALSE,INFINITE);
/* In the shipping version of GUIGREP, the thread function does not
   return a value but lives by its side effects. In this modified
   version, we assume that the thread function returns the number of
```

```
      hits so that we can further process the return value stored into
      iTemp here.    */
               GetExitCodeThread(aThreads[iArrayIndex],&iTemp);
               /* Now use iTemp to further process the result */
               CloseHandle(aThreads[iArrayIndex]);
            };
   /* ProcessFileCommonCode is a function that processes a file by
      index. In this code fragment we merely pass the index to the
      thread; in GUIGREP, ProcessFileCommonCode expects a pointer to
      a structure that contains the index as well as additional data.
      The File Manager knows how to convert the index into a file name. */
         hNewThread = CreateThread(NULL,0,
                         (LPTHREAD_START_ROUTINE) ProcessFileCommonCode,
                         iLoop,0,(LPDWORD)&iThread);
         aThreads[iArrayIndex] = hNewThread;

      };

   /* Section 2: Clean up all remaining threads. */

      iEndLoop = min(iEndLoop,MAX_CONCURRENT_THREADS);
      while (iEndLoop > 0)
      {
         iArrayIndex = WaitForMultipleObjects(iEndLoop, aThreads,FALSE,INFINITE);
         GetExitCodeThread(aThreads[iArrayIndex],&iTemp);
   /* Now use iTemp to further process the result. */
          CloseHandle(aThreads[iArrayIndex]);
          if (iArrayIndex < iEndLoop-1)
             aThreads[iArrayIndex] = aThreads[iEndLoop-1];
        iEndLoop--;
      };

   /* We are done! Do some cleaning up here... */
   /*    ...                                   */
      return(0);
   }
```

The array of threads for which **WaitForMultipleObjects** will wait is **aThreads**. Because it is fairly hard to maintain a dynamically sized array and the responsiveness of the system deteriorates rapidly if very many threads execute concurrently, the manifest constant MAX_CONCURRENT_THREADS limits the number of threads running at any time. Also, there is a system-provided limit on handles for which **WaitForMultipleObjects** can wait (the constant MAXIMUM_WAIT_OBJECTS contains this value). MAX_CONCURRENT_THREADS cannot be greater than MAXIMUM_WAIT_OBJECTS, or the application will not work correctly. Note that if an application wanted to exploit the underlying hardware more efficiently, it might want to query the number of processors on the machine using the **GetSystemInfo** API and configure MAX_CONCURRENT_THREADS as a function of the number of processors installed.

This code basically consists of two parts. In the first part, all threads are created and dispatched using the **CreateThread** function. The tricky part here is to determine what the array index of the newly created thread in **aThreads** is to be. For the first MAX_CONCURRENT_THREADS, this is easy; we just allocate the next free slot in the array for the thread handle. If the number of files selected by the user is smaller than MAX_CONCURRENT_THREADS,

the array never gets filled up, and execution proceeds with Section 2.

If there are more files to process than there are slots in **aThreads**, we need to wait for one thread to terminate. This happens in the **else** branch of the **if** statement in Section 2: The call to **WaitForMultipleObjects** will return the array index of a handle in the array that signaled, because FALSE was passed as the third parameter, indicating that any signaling handle will cause the function to return. We retrieve the return value from that thread using the **GetExitCodeThread** function, process it, then destroy the thread object using **CloseHandle**, and recycle the array entry for the next thread.

This way, the processing of the individual files is interleaved with the processing of their return values, as discussed before. Before we go on to discuss what happens after all threads have been dispatched, we need to clarify one point. In the last paragraph, I said that **WaitForMultipleObjects** will return the array index of "a" handle in the array that signaled. What handle is that?

The documentation states that the handles are prioritized from the first to the last entry. This is a rather fancy way to paraphrase that Windows NT internally executes a **for** loop through that array that runs from the first to the last entry, and as soon as it has retrieved one handle that has signaled, it will return that handle. In other words, if thread handle A has a lower index in the array than thread handle B, A will be returned from **WaitForMultipleObjects** before B if both objects signaled before the call to **WaitForMultipleObjects**, even if B's thread signaled earlier.

## Who Says That Waiting Is Easy?

The second phase of the execution (Section 2) is entered as soon as all threads are dispatched. In theory, what needs to be done is exactly the same as in Section 1—that is, wait for a thread to terminate, retrieve its return value, and process that value. Unfortunately, the difference between Section 1 and Section 2 is significant enough to make the control flow quite different.

The problem is that there is no new thread to fill in for the one that just terminated in Section 2. We cannot leave the handle of the object that just signaled in the array because then it will be returned again the next time **WaitForMultipleObjects** is called. A thread that has signaled once is always signaled, and due to the priority scheme mentioned above, no other handle would ever be returned again if we left a signaled handle in the array.

What happens if we close the handle and leave it in its array position? Even worse: **WaitForMultipleObjects** will return right away with the error code ERROR_INVALID_HANDLE. So how about taking the handle of any object that has not signaled yet and fill it in for the handle we just closed? Well, we could do that, but there is another problem with **WaitForMultipleObjects**: The call will fail right on the spot if any element occurs more than once in the array—even if there are two different handles to the same object. Thus, we would need to find a unique, nonsignaled object handle for every slot in the array that is freed. This may very well be impractical because we would have to create a potentially rather large number of bogus objects that exist only to be placeholders, and afterwards, we would have to delete all of them, too.

The solution here is to substitute each handle that has signaled with the last element in the array and tell **WaitForMultipleObjects** that the array has shrunk the next time we call it. We basically compress the array upon each iteration, throwing out the handle that has signaled, and terminate as soon as there is no handle left to signal anymore.

By the way, there are a few other ways to accomplish what we wanted to do in the first place. For example, instead of having the main thread gather the return values from all threads, each file-processing thread could communicate its return value to yet another thread as soon as it has the result. This other thread could be dedicated to collecting the results. This is basically the approach that the shipping version of GUIGREP takes, and it has the advantage that the degree of concurrency and interleaved instruction is slightly higher than in the "scatter-gather" version that we just introduced.

The disadvantage is that, depending on the form of communication, we might very well run into circular waiting situations that eventually lead to deadlock; also, in order to avoid synchronization problems that may arise if several threads try to access the same data at the same time, most probably some serialization mechanism needs to be employed.

## More Waiting...

A discussion of waiting would not be complete without mentioning the remaining members of the **WaitFor*xxx*** family, namely, **WaitForInputIdle**, **SleepEx**, **WaitForSingleObjectEx**, and **WaitForMultipleObjectsEx**. Although I will postpone the discussion of **WaitForInputIdle** for now, the other three functions direct our attention to asynchronous I/O processing, which is new for Windows NT. (By the way, asynchronous I/O processing will *not* be part of other implementations of the Win32 API because it requires a particular I/O model that at this point is implemented only under Windows NT.)

Asynchronous I/O is a way the operating system provides for letting I/O instructions execute in the background, the idea being roughly that a call to **ReadFile** or **WriteFile** will return immediately to the caller and overlap the I/O operation itself with the computation of the thread that called it in the first place. This technique only works for I/O devices whose drivers support asynchronous I/O and manifests itself to the programmer through the flag FILE_IO_OVERLAPPED that can be passed to the **CreateFile** call. I/O performed on such a file object will behave quite differently from synchronous I/O.

Working with asynchronous I/O is tremendously rich and far from being trivial, which is why I will not discuss in detail here the techniques that an application must use to work with it. I would like to describe, though, how asynchronous I/O can be synchronized with the thread that requests it.

First, under Windows NT, a file is an abstraction of an I/O device; as I'm sure you know, almost any I/O device can be opened via a call to **CreateFile** and accessed via the **ReadFile** and **WriteFile** calls. Thus, a file, like a thread, a process, a mutex, or a semaphore, is a "native Windows NT object," which means (among other things) that it can signal and thereby wake up a thread that is waiting for it. Conveniently, in the case of asynchronous I/O, the file object enters the unsignaled state when an I/O request is submitted and signals when the operation finishes. Thus, the following sequence can be used by a thread to submit an I/O request and then synchronize with its termination:

```
hFile = CreateFile(....,FILE_IO_OVERLAPPED,...);
ReadFile(hFile,...)
< Do some computation.>
WaitForSingleObject(hFile,INFINITE);
```

Thus, the time between the calls to **ReadFile** and **WaitForSingleObject** can be used to do something useful. If the line labeled <Do some computation.> is empty, the effect is the same as if the thread had submitted a synchronous I/O request (which, by definition, does not return before the I/O operation is completed) in a secondary thread.

At this point, it is appropriate to issue a note of caution: Although asynchronous I/O is a really powerful mechanism for overlapping thread work with I/O operations, it can easily be overused—I have a hard time coming up with a scenario in which the same effect as asynchronous I/O in one thread cannot be achieved by doing synchronous I/O in a background thread (except, maybe, for the requirement for truly random file access or an application that must log file accesses). Synchronous I/O is easier to code, but a little less flexible. You decide.

After this message from our sponsors, we can proceed directly to the second technique by which asynchronous I/O

can be synchronized with its requesting thread, *alertable waiting*. I like to nickname it "Ex" because all of the Win32 API functions that provide access to alertable waiting end in the suffix "Ex"—**ReadFileEx**, **WriteFileEx**, **WaitForSingleObjectEx**, **WaitForMultipleObjectsEx**, and **SleepEx**. None of these functions does any significant work on the Win32 subsystem level—they all are very thin layers that basically pass the call right on to the Windows NT executive. (Note that there are other functions in the Win32 subsystem that end in "Ex," but those are different meanings of "Ex.")

What happens here is that when submitting an asynchronous I/O request, the application may pass the address of a callback function to **ReadFileEx** or **WriteFileEx**, respectively; the callback (hereafter also referred to as a "completion routine") may be invoked after the request has finished.

Before I explain why the last sentence is so vague (Why "may be invoked?" Why "after the request has finished" instead of "when it has finished"?), let me make clear right away that alertable waiting is very much a byproduct of the way Windows NT implements its I/O system and its interrupt structure, which are very nicely explained in Chapters 7 and 8, respectively, of Helen Custer's *Inside Windows NT*. I strongly recommend reading these chapters to get a better understanding of these concepts.

The callback *may* be invoked because it will not be called unless the asynchronous I/O operation has completed *and* the requesting thread explicitly enters an "alertable wait state"; that is, calls one of the functions **SleepEx**, **WaitForSingleObjectEx**, or **WaitForMultipleObjectsEx** with the *fAlertable* parameter set to TRUE. Note that this requirement in a way synchronizes the asynchronous I/O, because the application decides when it wants to receive the asynchronous I/O notification.

**SleepEx**, **WaitForSingleObjectEx**, and **WaitForMultipleObjectsEx** have semantics similar to their nonalerting counterparts **Sleep**, **WaitForSingleObject**, and **WaitForMultipleObjects**, except that they will return not only when those counterparts would return, but also when an asynchronous I/O request has called the application-provided callback. If this happens, the return value of one of those functions will be the constant WAIT_IO_COMPLETION to distinguish the return from one that results from a signaling object. Thus, by executing the sequence

```
while (WaitForSingleObject(hObject,TRUE)==WAIT_IO_COMPLETION);
```

the calling thread, while waiting for *hObject* to signal, can process all outstanding completion requests and still perform its original wait—this loop will be left only when *hObject* has signaled, but in between, all pending asynchronous I/O requests that have finished will be completed. The effect of the completion will only be visible in the invocations of the completion routines.

This entire situation seems somewhat strange, doesn't it? Why is the callback function only callable when the requesting thread enters the alertable state? Why can't the alertable wait functions return the return value of the callback function? Why do we need this mechanism in the first place? Is Pat Riley a man or a woman?

To answer those questions (well, almost all of them), we need to look at the way those calls are implemented on the system level of Windows NT. Remember, though, before all of this discussion totally escapes you, that this API set exists because the operating system architecture that it is built on top of existed first, rather than because it was desperately needed. As stated before, in most cases you may find another, possibly easier way to implement what those functions accomplish.

The I/O manager in the kernel of Windows NT allows the time-critical part of an I/O operation to be uncoupled from the rest; that is, a hardware interrupt handler for an I/O device may decide to process only that which is absolutely

necessary in response to the interrupt and defer the rest for later at the processor's earliest convenience. This process separates an I/O operation into two components, the second of which is called the "completion" phase and is synchronized in that it will be executed only when Windows NT has determined that processing of the completion will not interfere with an executing thread.

Part of the completion is the optional invocation of what is called an *asynchronous procedure call* (APC), which is a process-supplied callback routine that is executed only in the context of the thread that submitted the I/O request. An APC is, in fact, invoked in response to a software interrupt of the lowest priority that Windows NT knows. Generally, a processor's interrupt priority is too high to allow an APC to be processed, but after the I/O routine completes, the I/O manager temporarily lowers the interrupt level of the processor that executes the requesting thread to process an APC.

This APC—which in the case of alerting asynchronous I/O is a static routine provided by the Win32 subsystem—invokes the callback supplied by the user. Regardless of whether the thread is in an alertable wait state or not, the software interrupt that corresponds to the APC is executed. If the thread is in the alertable wait state, the APC gets called as soon as the thread gets a timeslice; if not, the I/O manager queues the APC for the requesting thread. As soon as the thread enters the alertable wait state, it checks its APC queue and, if that queue is not empty, calls all outstanding APCs in the thread's queue and returns from the alertable wait state. If the queue is empty, however, a flag is set in the thread indicating that it is waiting to be alerted, and the thread itself is suspended.

This is why the callback is only invoked "after" the asynchronous I/O request has been completed; no assumptions can be made about how much time passes in between the completion of the request and the invocation of the APC.

There are a few nonobvious consequences of this implementation. First, the thread entering the alertable wait state "flushes" its APC queue—that is, any outstanding APCs for the thread will be processed. This is why **SleepEx**, **WaitForSingleObjectEx**, and **WaitForMultipleObjectsEx** cannot return a return value of the completion callback. When these functions return, more than one asynchronous request might have been processed, so whose return value should be returned? If the callback function needs to distinguish between different completed requests, it needs to do so by looking at the **OVERLAPPED** structure that is passed in to **ReadFileEx** or **WriteFileEx** and will be passed to the callback. If very many pending APCs queue up and the thread never enters an alertable wait state, the data structures that represent the APCs may drain system resources; thus, alertable I/O requests must be complemented by calls to enter an alertable wait state.

Second, you should not "chain" asynchronous I/O requests—that is, submit as asynchronous I/O request in the completion routine of another such request—because the completion routine will not return until the completion routine of its own asynchronous I/O request has been called. In other words, its call frame will linger on the thread's stack until both have returned. Chained asynchronous I/O requests, therefore, look very much like recursive function invocations because if too many of them occur, the thread's stack may blow up.

# Conclusion

In this article, I have discussed the minimal set of functions that is necessary to work with threads: **CreateThread** and the **WaitFor*xxx*Object** function set. The reason I harped on waiting functions is because they are about the only mechanism you cannot do without, regardless of how simple your multithreaded application is and how much interaction there is between threads. Your application must explicitly close the handle to each object it creates, otherwise it uses up system resources unnecessarily; and the only way to determine when a handle to a thread can be safely closed is to wait for the thread to terminate.

As indicated earlier, the bulk of any discussion about multithreading concerns synchronization. I recommend that you read Jeffrey Richter's article "Synchronizing Win32 Threads Using Critical Sections, Semaphores, and Mutexes," which can be found in the August 1993 issue of the *Microsoft Systems Journal* on this CD (Books and Periodicals, MS

Systems Journal). The article gives a fairly comprehensive overview of the synchronization mechanisms that Windows NT offers. After that, you should be ready for the other articles in this series on multithreading, which deal with the *really* intricate issues, such as using threads in the Win32 subsystem, synchronizing threads in practice, and combining threads with C++ objects.

# Bibliography

Custer, Helen. *Inside Windows NT*. Redmond: Microsoft Press, 1993.

Richter, Jeffrey. "Creating, Managing, and Destroying Processes and Threads under Windows NT." *Microsoft Systems Journal* (July 1993): 55-76.

Richter, Jeffrey. "Synchronizing Win32 Threads Using Critical Sections, Semaphores, and Mutexes." *Microsoft Systems Journal* (August 1993): 27–44.