Windows Multithreading for Application Developers

Ginny Caughey

CA-World 2000
eBusiness Solutions in Internet Time
JP095SNA/B

Introduction

Even casual users of 32-bit Windows know that their computers seem to be able to do more than one thing at a time. Your customers may not know anything about the underlying technology that makes this possible, but they do know that doing more than one thing at a time mirrors the way they themselves work. What's more, they may expect the software you develop for them to do the same. This paper provides you with the basics you need to fulfill this customer requirement. The example code uses CA-Visual Objects 2.5, which is my preferred language, but the concepts apply to multithreading in Windows in general.

Multitasking vs. Multithreading

An application that you write executes under Windows as a *process*. This is what used to be called a *task* in 16-bit Windows, and *multitasking* is the ability of Windows to run (or appear to run) more than one application at a time. Every process points to a private 4GB virtual address space allocated to it (2 GB reserved for the system's use and 2 GB for your program), and the code and data are loaded into this space by Windows. Finally, execution of the code begins at the standard entry point.

Each process, in turn, can run multiple threads. A *thread* is like a process within a process, with one important exception: threads do not have a private memory space allocated by the operating system. All threads run in the memory space already allocated to the process that owns them. Every process has one thread when it begins execution. This is the *primary thread*, and it begins automatically when the application starts execution. So even if you have not used multithreading in your applications, you have still been using threads, one per application. Additional *secondary threads*, if used, are created under your programmatic control.

When an application only has one thread, life is pretty simple. In some ways it's similar to writing an application where all the files are opened for exclusive use. You don't have to worry that someone else might try to update the record you are working on. Multithreading, however, introduces many of the issues you are familiar with for shared file access plus some brand new ones. For example, when you use multithreading in your applications, Windows doesn't give you any "lock required" message to warn you if one thread updates a global variable while another thread is also updating it. Therefore, it is your responsibility to recognize which resources in your application are global (and not all of them are global memory variables!), and provide a mechanism similar to record locking to ensure that each thread has exclusive access while these resources are updated. When the primary thread ends, any secondary threads your application has spawned that are still active will end too. So you may need to think about ways to keep your main application thread alive long enough for any secondary threads to complete their work or clean up after themselves. Thread synchronization techniques are discussed later in this paper.

Performance is an issue with both multitasking and multithreading. Since most of us are still using computers that only have a single processor, the simultaneous execution that we appear to see with multitasking and multithreading is an illusion. When a thread or process's time slice expires, all the information about the current state of that thread or process is saved so the thread or process can start right up where it left off the next time it gets a time slice. And before the next thread or process can resume operation, that thread's state must be restored. This operation is called a *context switch* and the data that is saved is called the *context record*. Saving and restoring all this information takes time, perhaps thousands of CPU cycles in some cases. You can control the priority of threads so Windows will allocate more CPU resources to one thread than to others, and this is also discussed later in this paper, but no matter what you do, there is some total performance overhead associated with multithreading. (This changes with computers with multiple processors, however. Windows is designed to allocate different threads to different processors in a multi-processor computer, and you don't even have to change your multithreaded code to take advantage of multiple processors if they are installed.)

In addition to the performance issue for context switches, another issue concerns exactly *where* in the thread the context switch occurs. Perhaps you have some code like this:

```
SomeMemvar := SomeValue
```

Unless you have also been writing assembly language code recently, it might seem at first glance that any context switch could only occur either before or after this line of code executes. But one line of CA-Visual Objects source code can represent many machine level instructions, and a context switch could occur anywhere in the middle of that code. This isn't a problem with multitasking, because the different tasks aren't sharing the same memory space. When the context is restored, the application continues with no problem because the data in the context record is still valid. But with multithreading, it is possible for the values saved in the context record for one thread to become invalid when another thread in the same process receives a time slice if both threads are accessing a shared resource. If this happens in an application you are developing, it may only occur sporadically, so debugging can be difficult. The best approach is to be aware of these potential problems and try to prevent them as you design the application.

When to Consider Multitasking

It should be clear from the preceding discussion that multithreading introduces some hassles you don't have to deal with when you solve a business problem using several separate applications. Perhaps you have already used WinExec() to spawn off another application to accomplish some job while your main application keeps on running and responding to end user keystrokes and mouse clicks. Consider these situations addressed by spawning separate applications:

- Running a Windows utility.
- Running an application that you didn't write and can't change.
- Using separate applications to customize the code for different customers (for example, different versions of PrintStatements.exe for different customers).
- Using separate applications to take advantage of alternative GUI classes (for example, using the ClassMate GUI Classes in some applications of a system and the CA-Visual Objects GUI Classes in others).
- Running a task manager system that takes tasks queued for background execution from a file-based queue by one or several applications and runs them separately. (George Smith did a case study presentation at Technicon several years ago using CA-Visual Objects applications at Pratt & Whitney to run night updates across all the idle CPUs on a network, replacing a mainframe-based system for a fraction of the cost.)
- Using OLE automation.

Spawning separate applications is generally easy to develop and debug, and with a Spawn class you can even have your main application wait until the spawned application has finished. The main disadvantage of this approach is that the separate applications can't easily share a lot of data, but passing data to the spawned program as parameters or in a temp file created for this purpose works well in many situations. With OLE automation, data could be passed to the spawned process in the OLE server's properties.

When to Consider Multithreading

The primary disadvantage of using separate processes becomes the main advantage of multithreading. Because threads do share the same memory pool, it's easier to share information between threads than it is between applications. Threads can also control other threads in the same application using a variety of techniques. Here are some ideas for things that could work well as separate threads:

- Polling a group of serial ports for input. (You can download a Serial class written in CA-Visual Objects from www.knowvo.com that uses multithreading to watch serial ports.)
- Sending off faxes or email. (In other words, a fax server or email server.)
- Writing a server of any sort. (See the WWW Server sample application that ships with CA-Visual Objects 2.5 for an example of a multithreaded Web server written in CA-Visual Objects.)

The main thing to notice about jobs that are designed to run in separate threads is that, as in multitasking, the jobs execute asynchronously. *If the rest of the application must wait for the thread to complete before it can continue its work, then there is usually no advantage in having separate threads.* (The exception that proves the rule: sometimes using threads can simplify the programming logic, in some situations making the overhead of threading a good tradeoff compared with programming complexity, but don't count on this simplification effect when you're just starting out with multiple threads!) And because of the overhead usually introduced with multithreading in terms of development complexity, debugging difficulty and runtime performance on single-processor computers, the development time as well as the total time to execute some job could all be longer with multiple threads than without.

When to Just Say No

Multithreading is definitely not the solution for every design problem, and timing is often the issue. Just because you have the technical ability to run some operation "in the background" doesn't mean that is always the best way to do it. Here are some examples where multithreading wouldn't help and could actually make matters worse:

- Reindexing DBF files. If you can't do anything until the indexing process is complete, you don't need a secondary thread. (On the other hand, you could create indexes for different DBF files in different threads with your main indexing function waiting for all the indexing threads to complete, but only on a multiprocessor computer *might* there be any advantage to this approach.)

- Performing some operation that will require locking files that would be needed by the main application. (This is a variation of the example above, since other work would have to wait until the files are unlocked.)

- Updating a screen display with the current time. Why not just use a Windows timer instead? Multiple threads can be notoriously tricky to debug since logic errors might appear only occasionally. Don't use them unless you *need* them. (Even if Windows timers use multithreading internally, *you* don't have to debug that code.)

- Using multithreading for the sake of using multithreading. This should be obvious, but sometimes programmers somehow get the idea that they should use some feature of a language (or in the case of multithreading, the operating system) because it's there. Multithreading handles some specialized situations well, but first make sure you *must* use multithreading to solve the problem. Then proceed.

How to Write Multithreaded Applications

The first step in writing a multithreaded application is justifying the design choice for introducing the complexity and overhead of multithreading. (Admittedly, most of the simple examples presented in this paper probably wouldn't pass this first test.) And the second step is to design the application before you begin writing code. This is good software development practice anyway, but it becomes even more important in multithreaded applications since they can be trickier to debug later than single-threaded applications, so it's especially important to get the design right before you start coding.

Getting Started with Multithreading in a Terminal Application

Writing the code for a multithreaded application involves two parts. The first part is writing the function that will execute as a separate thread. This function is the entry point for the secondary thread just as WinMain (inside the CA-Visual Objects runtime) is the entry point for your application's primary thread. Here is a simple example:

```
FUNCTION Count(dwParam AS DWORD) AS LONG PASCAL
	LOCAL i AS LONG

	FOR i := 1 TO 10000000
	NEXT
	RETURN 0
```

Notice that the function that executes as a secondary thread is strong typed. It may only take one argument, a DWORD, and it returns a LONG. The calling convention may be either STRICT or PASCAL. In this example, the function merely counts from one to ten million and returns. When the function returns, the thread ends.

The second part of writing a multithreaded application is to actually spawn off the thread. The Win32 API function for creating a new thread is called CreateThread. In CA-Visual Objects programming, you use CreateVOThread instead, which initializes some things in the CA-Visual Objects runtime and then calls CreateThread internally. Here is an example that times how long it takes to spawn off the Count function as a secondary thread and wait for that function to end:

```
FUNCTION Start()
	LOCAL hThread AS PTR
	LOCAL dwId AS DWORD
	LOCAL dwBegin AS DWORD

	dwBegin := GetTickCountLow()

	hThread := CreateVOThread(null, 0, @Count(), null_ptr, 0, @dwId)
	WaitForSingleObject(hThread, INFINITE)
	CloseHandle(hThread)

	? "Elapsed ",GetTickCountLow() - dwBegin
	wait
	RETURN 1
```

CreateVOThread, like the Win32 API function CreateThread, returns a handle to the new thread. All handles in CA-Visual Objects are of the PTR data type. The important parameters passed to CreateVOThread in this example are the address of the function that will execute as a separate thread, **`@Count()`**, and the address of a DWORD where Windows will store the new thread's thread ID, **`@dwId`**. (Search in the Win32 API Help file for CreateThread for a description of all the parameters.)

WaitForSingleObject is used in this example to keep the primary thread from ending before the Count function running as a separate thread has finished. If that line of the code were omitted, the program would end almost as soon as it began. WaitForSingleObject is a Win32 API function that takes the synchronization object to wait for as the first argument and the timeout value in milliseconds to wait as the second argument. This useful function can wait for a number of types of objects including events, timers, processes, and of course threads. After the thread has ended, the thread handle is closed using the Win32 API function CloseHandle. When I ran this code on my computer from the CA-Visual Objects IDE, the time to run the test was 2711 tenth milliseconds.

The next example uses the same Count function but instead of spawning one thread, it kicks off two threads. Here's the new Start function:

```
FUNCTION Start()
	LOCAL DIM hThreads[2] AS PTR
	LOCAL DIM dwIDs[2] AS DWORD
	LOCAL dwBegin AS DWORD

	dwBegin := GetTickCountLow()

	hThreads[1] := CreateVOThread(null, 0, @Count(), null_ptr, 0, @dwIDs[1])
	hThreads[2] := CreateVOThread(null, 0, @Count(), null_ptr, 0, @dwIDs[2])
	WaitForMultipleObjects(2, @hThreads[1], TRUE, INFINITE)
	CloseHandle(hThreads[1])
	CloseHandle(hThreads[2])

	? "Elapsed ",GetTickCountLow() - dwBegin
	wait
	RETURN 1
```

This example stores both the thread handles and the thread IDs in DIM arrays. This is convenient, because you now need the application to wait for two threads to complete, not just one. So WaitForMultipleObjects is used instead of WaitForSingleObject. WaitForMultipleObjects takes the address of an array of synchronization objects as the second argument, and the first argument is the number of objects that will be in that array. The third argument specifies whether to wait for all the objects (TRUE) or only the first one (FALSE), and the last argument is the timeout value in milliseconds.

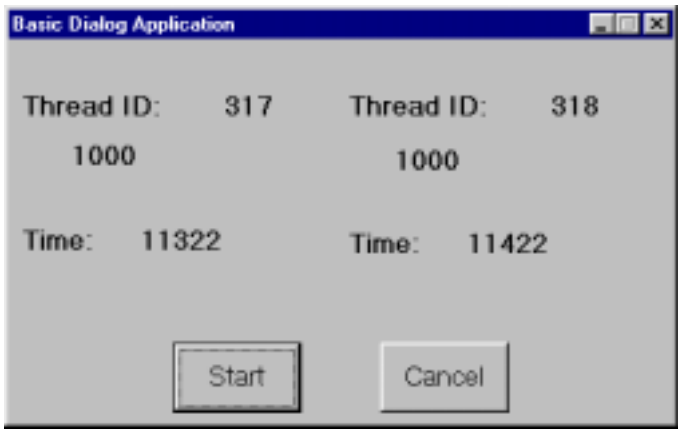
This code completed in 5411 tenth milliseconds on my test machine. That's about twice as long as the first example, so you can easily see that the application isn't really accomplishing the work any faster with multithreading on a computer with only one processor. You could design an application that spawns off some task and then returns control to the end user so the end user could continue with other work, and this would give the *impression* that the application is faster with multithreading, but it is only the user responsiveness that is faster. If you have any doubt, here's a third example using the same Count function:

```
FUNCTION Start()
	LOCAL dwBegin AS DWORD

	dwBegin := GetTickCountLow()
	Count(0)
	Count(0)

	? "Elapsed ",GetTickCountLow() - dwBegin
	wait
	RETURN 1
```

This example uses no multithreading at all and also completed in 5411 tenth milliseconds on my test machine. If there were many threads and therefore many context switches, this example with no secondary threads would theoretically be measurably faster than the version using multithreading to accomplish the same total task on a single-processor computer.

A Simple GUI Example of Multithreading

Multithreading may not save total CPU cycles to complete tasks, but the impression the end user gets of faster responsiveness can be important nonetheless. In fact, improving user responsiveness is one of the most widely used reason for multithreading. (Writing server applications is the other main reason to use multithreading.) Next look at an example of a simple GUI application that uses multithreading to display the number on the screen as the two threads count to 1000.



In this example, each thread counts to 1000 and then reports the time in tenth milliseconds elapsed for that thread to complete. (Because this is a graphical application that updates the screen for each number reached between 1 and 1000 for each thread, it naturally runs much slower than the terminal examples presented earlier in this paper.)

Several new techniques are introduced. Look at the class definition for the dialog window:

```
CLASS MainDialog INHERIT DIALOGWINDOW

       PROTECT oCCStartButton AS PUSHBUTTON
       PROTECT oCCCancelButton AS PUSHBUTTON
       PROTECT oDCID1 AS FIXEDTEXT
       PROTECT oDCID2 AS FIXEDTEXT
       PROTECT oDCCount1 AS FIXEDTEXT
       PROTECT oDCCount2 AS FIXEDTEXT
       PROTECT oDCElapsed1 AS FIXEDTEXT
       PROTECT oDCElapsed2 AS FIXEDTEXT

  //{{%UC%}} USER CODE STARTS HERE (do NOT remove this line)
  PROTECT s1, s2 AS Params
```

The dialog has two push buttons, a Start button and a Cancel button. And there are six fixed text controls, three for each thread. The first two are used to display the thread ID, the second pair are used to display the current count for each thread, and the third pair are used to display the time elapsed for each thread to count to 1000.

There are also two pointers to a Params structure in the class definition. Remember that worker thread functions can only take one argument, which must be a DWORD, but that function will need to have access to both the Count and Elapsed time fixed text controls so it can update the display. A structure is used to hold pointers to these controls so they can both be passed to the thread function. Here's the Params structure:

```
STRUCTURE Params
       MEMBER CountControl AS PTR
       MEMBER TimeControl AS PTR
```

The secondary threads are spawned when the Start button is pressed. Here's the code:

```
METHOD StartButton( ) CLASS MainDialog
       LOCAL DIM hThreads[2] AS PTR
       LOCAL DIM dwIDs[2] AS DWORD

       IF s1 = NULL_PTR
              s1 := MemAlloc(_sizeof(Params))
              RegisterKid(s1, 2, FALSE)
       ENDIF
       IF s2 = NULL_PTR
              s2 := MemAlloc(_sizeof(Params))
              RegisterKid(s2, 2, FALSE)
       ENDIF

       s1.CountControl := PTR(_CAST, SELF:oDCcount1)
       s1.TimeControl := PTR(_CAST, SELF:oDCElapsed1)
       s2.CountControl := PTR(_CAST, SELF:oDCcount2)
       s2.TimeControl := PTR(_CAST, SELF:oDCElapsed2)

       hThreads[1] := CreateVOThread(null, 0, @Count(), s1, 0, @dwIDs[1])
       hThreads[2] := CreateVOThread(null, 0, @Count(), s2, 0, @dwIDs[2])
       CloseHandle(hThreads[1])
       CloseHandle(hThreads[2])
       oDCId1:Caption := "Thread ID: "+AsString(dwIds[1])
       oDCId2:Caption := "Thread ID: "+AsString(dwIds[2])
```

Although there are similarities between this code and the Start functions used in the Terminal examples, there are also a number of interesting differences. Perhaps the most important is the omission of the WaitForMultipleObjects call. Since the window used here is a modal dialog window, the application will keep running until the end user explicitly closes it by pressing the Cancel button. So WaitForMultipleObjects is not needed to keep the primary thread active until any secondary threads have ended. And, in fact, if you do insert a call to WaitForMultipleObjects here in the StartButton method, there will be problems. The primary thread will wait all right, but while it is waiting it will not process any normal Windows messages. In other words, it will not respond to the user or to any attempts by the secondary threads to update the captions on the fixed text controls. This is not what you want!

The second thing to notice is that the thread handles can actually be closed while the thread is still running. They are not needed after the call to CreateVOThread, so they are closed immediately afterwards. In fact, this is usually the best way to clean up thread handles—just close them right after the threads are created if you won't be needing them for WaitForSingleObject or WaitForMultipleObjects.

The third major difference involves working with the two structures used to hold parameters for the Count function. The structures are declared in the class definition, and storage is allocated for the structures s1 and s2 when the Start button is pressed. Because the user could press the Start button more than once during the execution of the application, storage is allocated only if s1 and s2 are null pointers. And because the pointers contained in the s1 and s2 structures are pointers into dynamic memory, RegisterKid is used to indicate to the garbage collector that the pointers contained in those structures should be updated whenever the dynamic variables they point to are moved in memory. The Kids are unregistered and memory allocated to s1 and s2 is freed in the QueryClose method, which is called as the window is closed.

```
METHOD QueryClose(oEvent) CLASS MainDialog
	LOCAL lAllowClose AS LOGIC
	lAllowClose := SUPER:QueryClose(oEvent)
	//Put your changes here
	IF s1 != null_ptr
		UnRegisterKid(s1)
		MemFree(s1)
	ENDIF
	IF s2 != null_ptr
		UnRegisterKid(s2)
		MemFree(s2)
	ENDIF
	RETURN lAllowClose
```

Here's the new Count function that updates fixed text controls on the dialog window with the current count and elapsed time:

```
FUNCTION Count(dwParam AS DWORD) AS INT PASCAL
	LOCAL i AS LONG
	LOCAL oText1, oText2 AS FixedText
	LOCAL s AS Params
	LOCAL dwBegin AS DWORD

	dwBegin := GetTickCountLow()
	s := dwParam

	oText1 := OBJECT(_CAST, s.CountControl)
	oText2 := OBJECT(_CAST, s.TimeControl)
	FOR i := 1 TO 1000
		oText1:Caption := AsString(i)
	NEXT
	oText2:Caption := "Time: "+AsString(GetTickCountLow()-dwBegin)
	RETURN 0
```

If you run this code, you will see that the two text controls that display the current value of i update at approximately the same speed and reach 1000 at about the same time. And when both threads have finished, the elapsed time for each thread is about the same. (It was 11322 tenth milliseconds for the first thread and 11422 tenth milliseconds for the second thread when I saved the screenshot for this paper.)

Setting Thread Priorities

Windows has two mechanisms that determine the priority for threads. The first value that affects the priority for a thread is the priority of the class for the process that owns it. The priority class for a process can be changed programmatically using the Win32 API functions GetPriorityClass and SetPriorityClass, but generally you don't need to change the priority class for your applications. If you make your applications run faster, the rest of the system will become less responsive, so you won't be improving overall performance in most cases. Or to put it differently, your end user won't be happier although he might not know that it was your application that was to blame.

In addition to the priority class for the process as a whole, the thread can have its own priority relative to the priority of the process. The default thread priority is THREAD_PRIORITY_NORMAL. For a background task, however, you might want to decrease the priority of the thread to make the rest of the application relatively more responsive. The thread priority values are (from lowest to highest): THREAD_PRIORITY_IDLE, THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_NORMAL, THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST, and THREAD_PRIORITY_TIME_CRITICAL. You can find complete information about the thread priority options in the Win32 API Help file by searching on SetThreadPriority, but in general you will only consider using the values between LOWEST and HIGHEST.

To illustrate how changing thread priority can affect an application, make some modifications to the previous GUI example. First, change the structure aParams so it can hold three members. The third structure member will indicate whether the thread priority should be changed or not.

```
STRUCTURE Params
       MEMBER CountControl AS PTR
       MEMBER TimeControl AS PTR
       MEMBER ChangePriority AS LOGIC   // Add member for priority change
```

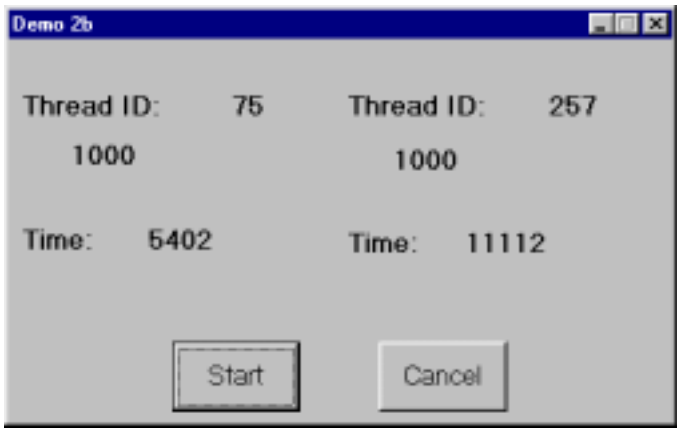
Next, change the StartButton method so the first worker thread does not have its priority changed while the second worker thread does have its priority changed:

```
METHOD OKButton( ) CLASS MainDialog
       LOCAL DIM hThreads[2] AS PTR
       LOCAL DIM dwIDs[2] AS DWORD
       LOCAL dwRet AS DWORD

       IF s1 = NULL_PTR
              s1 := MemAlloc(_sizeof(Params))
              RegisterKid(s1, 2, FALSE)
       ENDIF
       IF s2 = NULL_PTR
              s2 := MemAlloc(_sizeof(Params))
              RegisterKid(s2, 2, FALSE)
       ENDIF

       s1.CountControl := PTR(_CAST, SELF:oDCcount1)
       s1.TimeControl := PTR(_CAST, SELF:oDCElapsed1)

       // Don't change priority of first thread
       s1.ChangePriority := FALSE

       s2.CountControl := PTR(_CAST, SELF:oDCcount2)
       s2.TimeControl := PTR(_CAST, SELF:oDCElapsed2)

       // But change priority of second thread
       s2.ChangePriority := TRUE

       hThreads[1] := CreateVOThread(null, 0, @Count(), s1, 0, @dwIDs[1])
       hThreads[2] := CreateVOThread(null, 0, @Count(), s2, 0, @dwIDs[2])
       CloseHandle(hThreads[1])
       CloseHandle(hThreads[2])
       oDCId1:Caption := "Thread ID: "+AsString(dwIds[1])
       oDCId2:Caption := "Thread ID: "+AsString(dwIds[2])
```

Finally, change the Count function to either change the thread priority to a lower value or not depending on the value of the third structure member:

```
FUNCTION Count(dwParam AS DWORD) AS INT PASCAL
	LOCAL i AS LONG
	LOCAL oText1, oText2 AS FixedText
	LOCAL s AS Params
	LOCAL dwBegin AS DWORD
	LOCAL iPriority AS LONG
	LOCAL hThread AS PTR
	LOCAL lChangePriority AS LOGIC


	dwBegin := GetTickCountLow()
	s := dwParam

	oText1 := OBJECT(_CAST, s.CountControl)
	oText2 := OBJECT(_CAST, s.TimeControl)
	lChangePriority := s.ChangePriority

	IF lChangePriority
		// Get the handle to this thread
		hThread := GetCurrentThread()
		// Save the current thread priority
		iPriority := GetThreadPriority(hThread)
		// Decrease the thread priority
		SetThreadPriority(hThread, THREAD_PRIORITY_LOWEST)
	ENDIF

	// Count to 1000 as before
	FOR i := 1 TO 1000
		oText1:Caption := AsString(i)
	NEXT

	IF lChangePriority
		// Restore the previous thread priority
		SetThreadPriority(hThread, iPriority)
		// Clean up by releasing the thread handle
		CloseHandle(hThread)
	ENDIF

	oText2:Caption := "Time: "+AsString(GetTickCountLow()-dwBegin)
	RETURN 0
```

This time when the code executes, you can see that the two worker threads are not running at the same speed:



When I ran this test, the second thread counted up to about 6, then it seemed to pause until the first thread finished, then it continued counting up to 1000. As you can see from the screenshot, the second thread took about twice as long at THREAD_PRIORITY_LOWEST as the first thread running at THREAD_PRIORITY_NORMAL.

Notice when you are changing the priority of a thread that there are five steps:

1. Get the handle to the current thread by calling GetCurrentThread.

2. Get the current thread's priority by calling GetThreadPriority and save that value so you can reset it later.

3. Call SetThreadPriority to whatever new value you want. Do NOT use THREAD_PRIORITY_TIME_CRITICAL.

4. When you are finished, set the thread's priority back to the original priority by calling SetThreadPriority again.

5. Finally, call CloseHandle to free the resources used by the thread handle you got in step 1.

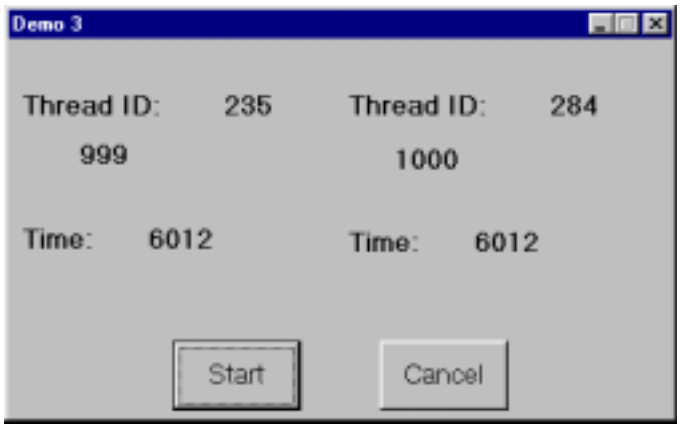
You can get as fancy as you want with thread priorities, but most of the time you won't need to do anything with the default thread priority. And in those situations where you do want to change the priority, most of the time it will be to decrease the priority of a worker thread rather than to increase it.

Sharing Memory

One of the biggest advantages of multithreading is that threads can all share the same memory. But if you don't plan carefully, this can sometimes seem like a big disadvantage instead. Go back to the original GUI sample and make one change to the Count function: change the local variable that functions as the loop counter to a static variable:

```
FUNCTION Count(dwParam AS DWORD) AS INT PASCAL
       STATIC LOCAL i AS LONG
       LOCAL oText1, oText2 AS FixedText
       LOCAL s AS Params
       LOCAL dwBegin AS DWORD

       dwBegin := GetTickCountLow()
       s := dwParam

       oText1 := OBJECT(_CAST, s.CountControl)
       oText2 := OBJECT(_CAST, s.TimeControl)
       FOR i := 1 TO 1000
             oText1:Caption := AsString(i)
       NEXT
       oText2:Caption := "Time: "+AsString(GetTickCountLow()-dwBegin)
       RETURN 0
```

When the test is run, both worker threads update the variable i so the test runs about twice as fast. Here's the screenshot:



You can see that the first worker thread never displayed 1000 because it was the second worker thread that actually incremented the static variable i to 1000. When the time slice returned to the first worker thread, 1000 had already been reached, so all that was left to do was display the elapsed time. (You can press the Start button repeatedly to run the test over and over, and sometimes the first worker thread will reach 1000 and the second worker thread will stop at 999 or 998 or 997, and sometimes it will be the other way around.)

Similarly, you could remove the static variable i, and change i to a global variable. The result is the same: both threads update the global variable, the test runs in about half the time, and at the end of the test only one thread reaches 1000.

Critical Sections

In the preceding examples, the application really has no control over how the variable i gets updated. This is not the safest way to update a shared resource and would be like a highway intersection with no traffic light. Two trucks (thread #1 and thread #2) come racing toward the intersection on intersecting roads. Will they crash? You have no way of knowing because it's a matter of timing and luck. It's much safer to install a traffic light so only one truck can enter the intersection at a time.

The multithreading equivalent of a traffic light is a *critical section*. Or more accurately, a critical section is a structure that acts like a green light. Only one thread "owns" a critical section at a time, and only the thread that owns the critical section can enter sections of code managed by the critical section.

Using critical sections involves several steps:

1. Declare a critical section structure and allocate the memory for it. A critical section object is a structure of type _WINRTL_CRITICAL_SECTION.

2. Initialize the critical section using InitializeCriticalSection.

3. Use EnterCriticalSection to mark the beginning of a block of code that you want to protect with a critical section. Only the thread that can get ownership of the critical section object is allowed to execute the next instruction in this block of code. If a thread can't get ownership of the critical section object, it is blocked (i.e., goes to sleep) until the critical section object becomes available. Then it can execute code inside the block of code managed by the critical section.

4. Use LeaveCriticalSection to mark the end of the block of code protected by a critical section.

5. Release the resources associated with the critical section when you're finished with it by calling DeleteCriticalSection.

You can easily change the GUI example which uses a global variable i for counting the loops to use a critical section to ensure that only one thread at a time is able to update the value of i. First, declare a global critical section structure:

```
GLOBAL CS IS _WINRTL_CRITICAL_SECTION
```

By using the IS keyword to declare the structure, the memory for that structure is automatically allocated. Next, fill in the code to initialize and delete the critical section object in the App:Start method:

```
METHOD Start() CLASS App
       LOCAL oDlg AS MainDialog

       oDlg := MainDialog{}
       InitializeCriticalSection(@CS)
       oDlg:Show()
       DeleteCriticalSection(@CS)
```

All that remains is to rewrite the Count function so every place where the global variable i can be updated is protected within a critical section:

```
FUNCTION Count(dwParam AS DWORD) AS INT PASCAL
	LOCAL oText1, oText2 AS FixedText
	LOCAL s AS Params
	LOCAL dwBegin AS DWORD

	dwBegin := GetTickCountLow()
	s := dwParam

	oText1 := OBJECT(_CAST, s.CountControl)
	oText2 := OBJECT(_CAST, s.TimeControl)

	// Initialize i to zero
	EnterCriticalSection(@CS)
	i := 0
	LeaveCriticalSection(@CS)

	WHILE TRUE
		IF i > 999
			EXIT
		ENDIF

		// Increment i
		EnterCriticalSection(@CS)
		++i
		LeaveCriticalSection(@CS)

		oText1:Caption := AsString(i)
	END
	oText2:Caption := "Time: "+AsString(GetTickCountLow()-dwBegin)
	RETURN 0
```

When this code is run, it behaves similarly to the preceding example, but this is safer code. Why? Because there is always the possibility that more than one thread could try to update i at the same time, but the context switch could occur somewhere in the middle of the operation updating i instead of always before or after that operation.

To show that critical sections really do work, modify the code so the entire count from 1 to 1000 is managed by a critical section:

```
FUNCTION Count(dwParam AS DWORD) AS INT PASCAL
	LOCAL oText1, oText2 AS FixedText
	LOCAL s AS Params
	LOCAL dwBegin AS DWORD

	dwBegin := GetTickCountLow()
	s := dwParam

	oText1 := OBJECT(_CAST, s.CountControl)
	oText2 := OBJECT(_CAST, s.TimeControl)

	// Initialize i to zero
	EnterCriticalSection(@CS)
	i := 0

	WHILE TRUE
		IF i > 999
			EXIT
		ENDIF

		++i

		oText1:Caption := AsString(i)
	END
```

```
LeaveCriticalSection(@CS)
oText2:Caption := "Time: "+AsString(GetTickCountLow()-dwBegin)
RETURN 0
```

With this change, one thread runs to completion, then the second thread runs to completion, just as you'd expect.

Which Resources Might Be Shared?

So far you have seen global memory variables and static locals within threaded functions as examples of resources that are shared by multiple threads. But you need to be aware of other resources that might be global in this sense.

Operating system handles are global.

For example, file handles are shared as are window handles. In fact, all Windows handles are global, but the Windows functions that allow you to change the values associated with these handles manage the traffic light functionality internally. And when a Windows message is received by a control or window, Windows makes sure that the thread that created that window or control owns the time slice when that message is delivered. So another way the sample above might have been constructed would have been to pass handles to the controls to be updated from the worker threads. Then the Dispatch method for those controls could update the values in the primary thread.

CA-Visual Objects Specifics

CA-Visual Objects workareas are global.

But what about CA-Visual Objects workareas? Even if you no longer think in terms of workareas in your applications because you use DBServer objects, the workareas that DBServer uses internally *are* global and visible throughout your application to all threads in your application. You have to be aware of this situation if you use DBServer objects in secondary threads, because passing DBServer objects to a threaded function can cause unexpected problems.

Consider this pseudocode for a function that executes as a worker thread:

```
FUNCTION WorkerThread(dwParam AS DWORD) AS LONG PASCAL
       // Some code here
       Lock Database
       Update Field
       Commit
       UnLock Database
       // Some other code here
       RETURN 0
```

Suppose worker thread #1 loses its time slice just after the Lock Database call. Thread #2 gets a time slice and can also get a legal lock since it is in the same process asking for the lock, and therefore it can continue on to update the database, commit and unlock. When thread #1 wakes up and continues where it left off, it then overwrites any changes made by the second worker thread.

This is similar to the situation with the global memory variable, but the solution can actually be more complex and can present some surprises. An obvious approach would be to use critical sections to guard parts of the code, but which parts? At first glance it might seem that only the Lock operation needs to be protected with a critical section, like this:

```
FUNCTION WorkerThread(dwParam AS DWORD) AS LONG PASCAL
	// Some code here

	// Guard the Lock step with a critical section
	EnterCriticalSection
	Lock Database
	LeaveCriticalSection
	Update Field
	Commit
	UnLock Database
	// Some other code here
	RETURN 0
```

The problem with this solution is that again worker thread #1 could acquire the lock, leave the critical section, then time out. Thread #2 could then also acquire the lock and update the file. Then when the first worker thread receives the time slice, the data actually in the file no longer matches what that thread expects, even though worker thread #1 still thinks the record is locked and indeed it may still be locked. Is this what you would have expected?

Record locks work across processes but they don't work within the same process within the same workarea. The following code illustrates why that is so:

```
FUNCTION Start
	SetDefault("c:\cavo25\samples\gstutor")

	use customer new shared alias one
	? one->(DBRLOCK())  // Returns TRUE
	? one->(DBRLOCK())  // Returns TRUE
	wait
```

It has always been possible in the Clipper language to lock the same record in a workarea repeatedly without unlocking it. Although this example doesn't show the code running in different threads, it makes no difference to the CA-Visual Objects RDD system, and two threads attempting locks on the same record would give the same results–only worse. The "worse" part with the critical section only guarding the lock operation is that, depending on exactly where the context switch occurs, (1) you might lose an update entirely, (2) you might not, or (3) the second thread might attempt to write to a record that is no longer locked and crash with an error because the first thread has unlocked the record. Programming can be difficult enough without this sort of uncertainty! And unless you have thought this process through while you're designing your application, the situation might not even occur at all during your own testing, leaving you trying to figure out why data is unreliable at a customer's site. You could blame Microsoft and network drivers, etc., and not realize what the real problem actually is.

Since using a critical section to guard only the Lock operation doesn't work, next consider this pseudocode:

```
FUNCTION WorkerThread(dwParam AS DWORD) AS LONG PASCAL
	// Some code here

	// Guard the entire transaction with a critical section
	EnterCriticalSection
	Lock Database
	Update Field
	Commit
	UnLock Database
	LeaveCriticalSection
	// Some other code here
	RETURN 0
```

This solution solves the problem of possible lost updates and locking errors, but there are still some issues to be aware of. Suppose there is actually a lot of code involved in the update step and multiple files and/or records must be updated. The process might not get a record lock on the first try and might have to retry. You might even need to do some printing or send off a fax or email as part of your business update process. All of these things take time. And during all this time, any other threads attempting to do the same task are blocked by the critical section. The performance might become unacceptable, and if you aren't very careful with multiple file and record updates, you could code your threads into a deadlock scenario.

Also consider this code:

```
FUNCTION Start
       SetDefault("c:\cavo25\samples\gstutor")

       use customer new shared alias one
       one->(DBSkip(1))
       ? one->(Recno())    // Returns 2
       one->(DBSkip(1))
       ? one->(Recno())    // Returns 3
       wait
```

If you move the record pointer in a workarea in a worker thread, some other worker thread may find that following a context switch it is no longer working on the same record as before the context switch. That situation could corrupt data and be really interesting to debug at a customer site!

You probably now realize that passing a DBServer object to a worker thread might not be a good idea. But what if the worker thread opened its own copy of the file instead? Would that be any improvement? The following test hints at the answer:

```
FUNCTION Start
       SetDefault("c:\cavo25\samples\gstutor")

       use customer new shared alias one
       use customer new shared alias two
       ? one->(DBRLOCK())  // Returns TRUE
       ? two->(DBRLOCK())  // Returns FALSE
       wait
```

Fortunately, record locks are respected between *different* workareas even within the same process. And of course different workareas each have their own record pointers. Now we're getting closer to the behavior of local memory variables in worker functions and less like the behavior of globals or statics. But this doesn't mean that the following pseudo code is safe:

```
FUNCTION WorkerThread(dwParam AS DWORD) AS LONG PASCAL
       LOCAL oDB AS DBServer

       // Some code here
       oDB := DBServer("customer")
       IF oDB:RLock()
              Update Field
              Commit
              UnLock Database
       ENDIF
       oDB:Close()

       RETURN 0
```

There is still the possibility that two threads, each trying to instantiate DBServer objects at the same time, could step on each other inside the DBServer:Init method. Depending on where the context switch occurs, you could end up with two threads trying to open files with the same alias, as just one example. But in this case, a critical section should be able to handle the situation:

```
FUNCTION WorkerThread(dwParam AS DWORD) AS LONG PASCAL
	LOCAL oDB AS DBServer

	// Some code here
	EnterCriticalSection(@CS)
	oDB := DBServer("customer")
	LeaveCriticalSection(@CS)
	IF oDB:RLock()
		Update Field
		Commit
		UnLock Database
	ENDIF
	oDB:Close()

	RETURN 0
```

With a critical section guarding the assignment of the DBServer object to a local memory variable, each worker thread now has its own unique DBServer object with its own workarea.

Not Every Part of CA-Visual Objects 2.5 is Thread Safe.

CA-Visual Objects 2.5 greatly improves multithreading over previous versions, but not every part of CA-Visual Objects 2.5 is intended for use in multithreaded applications. Mostly the restrictions follow the lines of common sense, but you still have to think about them while you are designing your multithreaded application.

For example, what happens in worker threads when there is a BEGIN SEQUENCE construct? As it turns out, the only sensible way for Computer Associates to deal with this situation was to have separate BEGIN SEQUENCE stacks for each thread in an application. After all, you wouldn't want a BREAK occurring in one worker thread that was caused by an error condition in a different worker thread. This makes sense, but make sure you don't expect error conditions in a worker thread to cause a BREAK in the primary thread. Each thread must be responsible for gracefully handling its own errors and informing the calling thread about them if that is appropriate.

Terminal Lite may also not behave as you expect in a multithreaded application. Refer back to the earliest examples of multithreading in this paper that use the Terminal window to display debugging information and you'll see that only the primary thread actually writes to the Terminal. This is the only safe way to use Terminal– restrict its use to the primary thread. If you need debugging information during testing as worker threads run, use the low-level file functions to write to a trace file instead. And remember to either use separate files for each thread or use critical sections to manage access to a shared file, and be sure to also write out the thread ID for each action so you can see which thread did what!

And finally, the CA-Visual Objects GUI classes are not thread safe. So even though the examples in this paper showed sharing GUI controls between threads, this is really not recommended. It would be better to use Windows messages to communicate with the controls in the primary thread rather than update the caption of those controls directly from worker threads.

Getting Threads to Talk to One Another

In the previous examples, the primary and worker threads really just did their own thing. There was no attempt made to communicate directly from one thread to another. But in a multithreaded application, such inter-thread communication is usual. So how is it done?

The easiest approach is to use global variables visible to all threads. In some circumstances, this approach works fine, and that is what I used in my Serial class to communicate between the primary and worker threads to indicate a need to shut down the operation.

But sometimes a more precise means of communication is necessary. I wrote a commercial application that monitors the activity on up to four truck scales. (The actual number of scales is user configurable.) The first version of this application only handled one scale, so increasing the number of scales looked like a good application for multithreading, one scale monitored on each thread. Each thread opens its own copy of the files needed to record the activity on the scale, and as each thread is launched, its thread handle is added to an array of worker threads dwIDs and the count of workers launched in nThreads is incremented. When the user presses the Cancel button on the form running on the primary thread, the Windows API function PostThreadMessage is used to tell all the worker threads, however many there might be, to shut down. The code looks like this:

```
// tell all threads to shut down now
FOR i := 1 TO nThreads
      PostThreadMessage(dwIDs[i], WM_QUIT, 0, 0)
NEXT

// wait for all threads to shut down so data gets correctly written
WaitForMultipleObjects(nThreads, @hThreads, TRUE, INFINITE)
```

PostThreadMessage is used to send a Windows message to a thread handle, just as PostMessage is used to send a Windows message to a window handle. And of course you are not limited to messages aready defined like WM_QUIT; you can easily define your own custom messages.

Finally the main thread waits until all the workers have shut down before ending the application. This is important, because if the primary thread didn't wait for all the workers to finish writing their data to the file, data could get lost. And in my production example, serial ports also need to be shut down in an orderly fashion.

Posting a message to a thread is very easy—the tricky part is getting the thread to respond to the message. If you have created any windows at the API level, you know that you must write a message pump to look for and handle Windows messages. (And even if you don't write this code yourself, your development tool generates code that does this for you.) When you use PostThreadMessage for inter-thread communication, you must also provide a message pump for the thread that will be receiving the message.

In my application, each thread does basic initialization and then goes into a polling loop. This polling loop provides the place for a message pump by just adding a little code:

```
WHILE TRUE
      // Check thread's message queue for messages/
      // PeekMessage returns TRUE if there are messages waiting.
      IF PeekMessage(@uMsg, 0, 0, 0, PM_REMOVE)
            IF uMsg.message == WM_QUIT
                  EXIT
            ENDIF
      ENDIF
      // do some real work here
END
// do clean up here
RETURN 0
```

Once out of the loop, shutdown operations are handled and the thread ends by RETURNing. Because the amount of work done during the loop is non-trivial in my production application, there are actually several other places in my code where PeekMessage is checked so the application will be as responsive as possible.

Multithreading and Components

All the discussion so far in this paper has been directed to multithreading in a one-tier application. This was intentional, because it is important to understand the concepts of multithreading in a simple application before worrying about how threading might come into play in an n-tier setting. And, as you have seen, multithreading can have benefits even in a one-tier application. But where multithreading can be extremely important (and also very complex) is in server-type applications that execute as COM components.

The history of Microsoft Windows is also the history of COM. NT 3.1 had threads but didn't have COM. NT 3.5 allowed one thread per process, followed in NT 3.51 with single-threaded apartments. Finally, according to David Platt (see bibliography), "Windows NT 4.0 allowed tough objects to hold wild parties in the multithreaded apartment." Finally Windows 2000 introduces COM+ and a new type of apartment, the neutral apartment. So the key to understanding threading in components is in understanding the various types of apartments and how they behave.

An *apartment* is a logical grouping within a process for objects that share the same thread. Every object can live in only one apartment, and the kind of apartment the object lives in determines the thread or threads on which it receives calls. When a thread wants to use COM (or COM+), it calls CoInitializeEx, and one of the parameters of CoInitializeEx indicates the threading model to use. The thread remains in that apartment until it ends or until CoUnInitialize is called.

The simplest apartment model is the *single-threaded apartment*, which is created when CoInitializeEx is called with the COINIT_APARTMENT-THREADED flag. All objects created in this apartment run on the same thread—the thread that created the apartment. Of course, a process can have many different threads and many different single-threaded apartments. But each object in a single-threaded apartment receives calls on the same thread. Most development tools that automate the creation of COM components use the single-threaded apartment model, and components that use a GUI must use this model. These components are also the easiest to write.

A more complex option exists, however. CoInitializeEx can also be called with the COINIT_MULTITHREADED flag. This creates a *multithreaded apartment*. Unlike single-threaded apartments, a process can have only one multithreaded apartment, and every thread within the process that calls CoInitializeEx with the COINIT_MULTITHREADED flag shares the same multithreaded apartment with all the other threads. This means that an object created on one thread can receive calls on a different thread. This can be useful for worker threads that are GUI-less, but because an object can receive a call on any thread at any time, it is up to the developer to ensure that concurrent accesses from different threads do not do any harm. This code can be very difficult to write correctly.

COM+ adds the third threading model: *thread neutral apartment*. The neutral apartment contains no threads at all, just objects. And the objects in a neutral apartment always receive their calls on the same thread as the caller (either single-threaded apartment or multithreaded apartment). Because objects are created on the same thread as the caller, there is no context switch on the server when objects are created, so this can be a high performance solution for components that share a thread pool and that don't use the Windows GUI. But synchronization issues can be very complex, especially where multiple computers are involved! COM+ provides *activity-based synchronization*, so you can write code as if you were working on a single logical thread that may span multiple processes and even multiple machines. COM+ provides this synchronization via locks, but there is the possibility for deadlocks if clients share objects and especially if clients share objects across process boundaries.

Conclusion

Multithreading can be very useful in some situations, particularly for creating servers of any sort, but you have to know what you're doing. CA-Visual Objects 2.5 brings enhancements that make multithreading safer and easier to use. Unlike previous versions, the Garbage Collector is now thread safe, so collectible data types such as strings, floats, Clipper arrays, and objects can be used inside worker thread functions. You can also use BEGIN SEQUENCE constructs within worker threads, which was not possible with previous versions of CA-Visual Objects. And the CA-Visual Objects RDDs have been improved to make them more thread friendly internally. But writing multithreaded code is still difficult to write properly and to debug and requires careful planning. In particular, you should avoid sharing resources or global memory if at all possible and carefully plan your critical sections where you cannot avoid sharing a global resource. If you are writing a component, you will probably be using the single-threaded apartment model unless you have a very good reason not to. This paper's purpose is to help you begin the journey, but it's up to you to plan your trip carefully and watch your step along the way!

Bibliography

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

Petzold, Charles. *Programming Windows Fifth Edition.* Redmond, Washington: Microsoft Press, 1999. Chapter 20 covers multitasking and multithreading.

Platt, David S. *Understanding COM+*. Redmond, Washington: Microsoft Press, 1999. Chapter 2 covers threading apartments.

Ginny Caughey is Vice President of Carolina Software, Inc., a Wilmington, NC based company specializing in applications for the solid waste industry. Her WasteWORKS® family of products written in CA-Visual Objects are in use throughout the U.S. and Canada. (See http://www.wasteworks.com for more information about WasteWORKS.) Ginny was also lead author of Special Edition Using Visual Objects *published by Que and has been a featured speaker at developer conferences in the U.S., Europe and Australia. She can be reached at gcaughey@compuserve.com.*