By Douglas A. Hamilton

How Threads Stitch It All Together

One of the most powerful aspects of Windows NT is its support for concurrent threads of execution. However, if you've never used threads or semaphores before, it can also be pretty challenging getting started with these things.

SCOTT BALDWIN

With more than 4,000 people signed up for the NT Developers Conference in San Francisco (dubbed "Winstock") and thousands more received the NT SDK on CD-ROM over the summer, it struck me as a good time to talk about what threads are, how they work and how a developer might take advantage of them under Microsoft's New Technology.

The easiest way to think about threads is in simple producer/consumer relationships. A producer/consumer relationship is any situation where the output of one operation is a more or less continuous stream of raw materials consumed by another operation. Often, it's a two-way relationship. Imagine trying to read a file from *stdin* and copy it to *stdout*. The read operation consumes empty buffers and produces buffers full of the data it has read; the write operation consumes full buffers and produces empty buffers.

A simplistic implementation would be a loop: Read a block, write it out and turn around to do the next one. Under this scenario, all of the blocking events associated with the mechanical I/O are serialized. The machine ends up spending most of its time waiting for slow mechanical events, not doing anything useful.

A better solution is to separate those operations into more of an assembly-line style. The reader should be free running, producing full buffers as fast as it can and stockpiling them for the writer, who meanwhile is emptying buffers as fast it can. But we need ways to handshake between the operations so the writer isn't grabbing buffers before the reader is finished filling them.

Threads provide a simple, standard way for the application designer to spawn off those asynchronous operations using the CreateThread call. Companion semaphore and critical section calls allow these activities to be coordinated.

When you call CreateThread, you pass it the name of the procedure you want to run in the new thread, the stack size it needs and a single pointer that will be passed to the child as an argument. CreateThread returns to you a handle to the child you've just created. Both threads—the original one you started with and now the new one—will run independently, each getting slices of the processor. If one thread hits a blocking event, for instance, waiting for the disk, the kernel will automatically begin giving all the cycles to the other thread. The idea is to try to keep the processor busy at all times doing useful work if any can be found. This is in sharp contrast to DOS, where any blocking event causes the whole system to stall, wasting most of the machine's cycles.

Also, NT has a special wrinkle: It can support symmetric multiple processors (SMPs). An SMP is a machine with several processor chips hooked in a symmetric fashion: All the processors are identical and can all do the same things. There aren't too many SMP machines on the market yet but then again, NT is about the only system that would know what to do with them. I mention SMPs because a thread is also NT's unit of work for a processor. Therefore, the moment you create a new thread, your application magically and automatically becomes eligible to use a multiprocessor.

When the child thread wants to exit, it calls ExitThread. You can test whether or not a thread is still running by examining the handle you got back from CreateThread with WaitForSingleObject. If you specify a timeout of 0, WaitForSingleObject returns immediately, returning TRUE if the thread has exited, FALSE otherwise. A timeout of -1 would mean you wanted to wait forever if need be for the thread to exit. It's also possible to specify any arbitrary timeout in milliseconds.

All threads within a process share all the same memory, all the same open file handles and so on. There is nothing to stop one thread from scribbling anywhere it likes on any data it finds lying around. But that's the whole point of a thread: When you create one, you want it to have access to everything so you can easily put it to work reading files you've opened or filling buffers you've allocated and so on. Also, by not having a lot of firewalling between threads, the cost of creating them is reduced. Again, that's what threads are all about: performance and responsiveness.

You can't create a lot of threads and let them run rampant. Their efforts have to be coordinated so they aren't all scribbling on the same variables at the same time and also so that when one needs to hand something off to another, that handshake can be synchronized.

The sharing problem is called mutual exclusion, or mutex for short. The idea is that before a thread tries to use a shared resource, for example, before reading or changing a shared variable, it must cooperate with all the other threads in a convention that assures that only that one thread will have access to that resource. NT provides a couple of mechanisms for mutex. The simplest and fastest is the critical section. However, if your background is in OS/2, let me be quick to point out this is not like an OS/2 critical section. When a thread entered an OS/2 critical section, every other thread was stalled. Also, there was no critical section variable; you just called DosEnterCritSec and every other thread stopped. The other threads did not have to explicitly cooperate in any convention; you just stopped them all in their tracks and did what you wanted.

An NT-critical section is more like an OS/2 RAM semaphore, in that it is kept in a variable within the application's memory space. All threads have to cooperate in the convention of calling EnterCriticalSection before manipulating the shared resource and calling LeaveCriticalSection when they're done. An NT-critical section cannot be used for anything but mutual exclusion, in contrast to OS/2 RAM semaphores, which could also be used for synchronization and signaling. In addition, if a thread wants to wait on more than one thing at a time (for instance, it wants to wait until either the resource it needs is free or it's been signaled that it should do something else) then the full-blown NT mutex mechanism has to be used.

A mutex object can do everything a critical section can do, plus it can be used with the WaitForMultipleObjects system function. Instead of being a variable in an application space, a mutex object is actually kept inside the kernel and only a handle (like the handle to a file or anything else) is given to the application when the mutex is created. That handle can even be passed to other processes.

Threads also have to signal when events have occurred. That's done with an event semaphore. When an event semaphore (or event, for short) is created, the application specifies whether or not it starts off signaled (the event has happened) or not signaled and whether or not it automatically resets itself. By automatic reset, we mean that if several threads are all waiting for the same event, the first one wakes up when it happens, and the event is immediately and automatically reset, meaning no other threads will wake up.

Each of these mutex or event mechanisms has to be initialized before it can be used (again, unlike OS/2 RAM semaphores) and should be explicitly closed (event), deleted (critical section) or destroyed (mutex) when it's no longer needed.

Coming back to our simple producer/consumer problem, a simple multithreaded solution might involve first setting up two queues, one to hold empty buffers and the other to hold full buffers. We'd allocate a lot of empty buffers and put them all on that list. For each queue, we'd create a critical section for mutual exclusion when either the reader or the writer wanted to add to the bottom or pull from the top. We'd also create an event for each queue so that if a thread went to get a new buffer and found the queue empty, it could go to sleep knowing it could be awakened when another buffer was ready.

At that point, we'd have to decide if the current thread is the reader or the writer. (It doesn't matter, but you do have to choose.) Whatever you choose, you start another thread to do the other activity.

Whenever a thread wanted to consume a new buffer, it would enter the critical section for its input queue, take a buffer and exit the critical section. If the queue was empty, it would clear the event, leave the critical section, wait on the event and try again.

When a thread had finished with a buffer, it would enter the critical section for its output queue, add it to the queue, set the event and leave the critical section.

This has been something of a whirlwind tour. But I hope this has been helpful in explaining how threads work, what the mechanisms that NT provide for creating and using threads look like and how it all might be put together in a realistic problem. ■

Douglas Hamilton is president of Hamilton Laboratories in Wayland, Mass., and author of the Hamilton C Shell, an advanced interactive command processor and tools package for OS/2 and NT. Reach Douglas on WIX as hamilton or care of Editor at the address on page 10.