The alertable wait is the non-GUI analog to pumping messages

devblogs.microsoft.com/oldnewthing/20060503-11

May 3, 2006

Raymond Chen

When you are doing GUI programming, you well know that the message pump is the primary way of receiving and dispatching messages. The non-GUI analog to the message pump is the alertable wait.

A user-mode APC is a request for a function to run on a thread in user mode. You can explicitly queue an APC to a thread with the `QueueUserAPC` function, or you can do it implicitly by passing a completion function to a waitable timer or asynchronous I/O. (That's why the code that indicates that a wait was interrupted by an APC is `WAIT_IO_COMPLETION`: Originally, the only thing that queued APCs was asynchronous I/O.)

Of course, when an APC is queued, the function cannot run immediately. Imagine what the world would be like if it did: The function would interrupt the thread in the middle of whatever it was doing, possibly with unstable data structures, leaving the APC function to run at a point where the program is in an inconsistent state. If APCs really did run this way, it would be pretty much impossible to write a meaningful APC function since it couldn't reliably read from or write to any variables (since those variables could be unstable), nor could it call any functions that read from or wrote to variables. Given these constraints, there isn't much left for a function to do.

Instead, APCs are dispatched when you perform what is known as an "alertable wait". The "Ex" versions of most wait functions (for example, `WaitForSingleObjectEx`, `WaitForMultipleObjectsEx`, `MsgWaitForMultipleObjectsEx`, and `SleepEx`) allow you to specify whether you want to wait alertably. If you do wait alertably, and one or more APCs are queued to the thread, then all the pending APCs are run and the wait operation returns with a code indicating that the wait was interrupted by an APC. If the APC you are waiting for has not yet run (maybe you were interrupted by some unrelated APC), then it is your responsibility to restart the wait and try again.

Why doesn't the operating system automatically restart the wait? "Imagine what the world would be like if it did": Suppose you want to issue asynchronous I/O and then go off and do some other stuff, and then wait for the asynchronous I/O to complete so you can use the result.

```
// When an asynchronous read completes, we fire off the next
// read request. When all done, set fCompleted.
BOOL fCompleted = FALSE;
BOOL fSuccess;
void CALLBACK CompletionRoutine(DWORD, DWORD, LPOVERLAPPED)
{
 if (<finished>) {
  fSuccess = TRUE;
  fCompleted = TRUE;
 } else {
  // issue the next read in the sequence
  if (!ReadFileEx(hFile, ..., CompletionRoutine)) {
   fSuccess = FALSE; // problem occurred
   fCompleted = TRUE; // we're done
 }
}
...
// start the read cycle
if (ReadFileEx(hFile, ..., CompletionRoutine)) {
  DoOtherStuffInTheMeantime();
  <wait for fCompleted to be set>
  DoStuffWithResult();
}
```

How would you write the "wait for `fCompleted` to be set" if the operating system auto-restarted waits? If you did an alertable infinite wait, say with `SleepEx(INFINITE, TRUE)`, then the APCs would run, the operating system would auto-restart the waits, and the sleep would just run forever. You would be forced to use a non-`INFINITE` sleep and poll for the completion. But this has two serious flaws: One is that polling is bad. The second is that the rate at which you poll controls how quickly your program reacts to the completion of the read chain. Higher polling rates give you better responsiveness but consume more CPU.

Fortunately, waits are not auto-restarted. This gives you a chance to decide for yourself whether you want to restart them or not:

```
...
// start the read cycle
if (ReadFileEx(hFile, ..., CompletionRoutine)) {
  DoOtherStuffInTheMeantime();
  while (!fCompleted) SleepEx(INFINITE, TRUE);
  DoStuffWithResult();
}
```

The `SleepEx` loop just keeps waiting alertably, processing APCs, until the completion routine finally decides that it's had enough and sets the `fCompleted` flag.

Raymond Chen

Follow