Using opportunistic locks to get out of the way if somebody wants the file

devblogs.microsoft.com/oldnewthing/20130415-00

April 15, 2013

Raymond Chen

Opportunistic locks allow you to be notified when somebody else tries to access a file you have open. This is usually done if you want to use a file provided nobody else wants it.

For example, you might be a search indexer that wants to extract information from a file, but if somebody opens the file for writing, you don't want them to get *Sharing Violation*. Instead, you want to stop indexing the file and let the other person get their write access.

Or you might be a file viewer application like ildasm, and you want to let the user update the file (in ildasm's case, rebuild the assembly) even though you're viewing it. (Otherwise, they will get an error from the compiler saying "Cannot open file for output.")

Or you might be Explorer, and you want to abandon generating the preview for a file if somebody tries to delete it.

(Rats I fell into the trap of trying to motivate a Little Program.)

Okay, enough motivation. Here's the program:

```
#include <windows.h>
#include <winioctl.h>
#include <stdio.h>


OVERLAPPED g_o;


REQUEST_OPLOCK_INPUT_BUFFER g_inputBuffer = {
  REQUEST_OPLOCK_CURRENT_VERSION,
  sizeof(g_inputBuffer),
  OPLOCK_LEVEL_CACHE_READ | OPLOCK_LEVEL_CACHE_HANDLE,
  REQUEST_OPLOCK_INPUT_FLAG_REQUEST,
};


REQUEST_OPLOCK_OUTPUT_BUFFER g_outputBuffer = {
  REQUEST_OPLOCK_CURRENT_VERSION,
  sizeof(g_outputBuffer),
};


int __cdecl wmain(int argc, wchar_t **argv)
{
  g_o.hEvent = CreateEvent(nullptr, FALSE, FALSE, nullptr);


  HANDLE hFile = CreateFileW(argv[1], GENERIC_READ,
    FILE_SHARE_READ, nullptr, OPEN_EXISTING,
    FILE_FLAG_OVERLAPPED, nullptr);
  if (hFile == INVALID_HANDLE_VALUE) {
    return 0;
  }


  DeviceIoControl(hFile, FSCTL_REQUEST_OPLOCK,
      &g_inputBuffer, sizeof(g_inputBuffer),
      &g_outputBuffer, sizeof(g_outputBuffer),
      nullptr, &g_o);
  if (GetLastError() != ERROR_IO_PENDING) {
    // oplock failed
    return 0;
  }


  DWORD dwBytes;
  if (!GetOverlappedResult(hFile, &g_o, &dwBytes, TRUE)) {
    // oplock failed
    return 0;
  }
```

```
  printf("Cleaning up because somebody wants the file…\n");
  Sleep(1000); // pretend this takes some time


  printf("Closing file handle\n");
  CloseHandle(hFile);


  CloseHandle(g_o.hEvent);


  return 0;
}
```

Run this program with the name of an existing file on the command line, say `scratch x.txt`. The program will wait.

In another command window, run the command `type x.txt`. The program keeps waiting.

Next, run the command `echo hello > x.txt`. Now things get interesting.

When the command prompt opens `x.txt` for writing, the `DeviceIoControl` call completes. At this point we print the `Cleaning up…` message.

To simulate the program taking a little while to clean up, we sleep for one second. Observe that the command prompt *has not yet returned*. Instead of immediately failing the request to open for writing with a sharing violation, the kernel puts the open request on hold to give our program time to clean up and close our handle.

Finally, our simulated clean-up is complete, and we close the handle. At this point, the kernel allows the command processor to proceed and open the file for writing so it can write `hello` into it.

That's the basics of opportunistic locks, but your program will almost certainly not be structured this way. You will probably not wait synchronously on the overlapped I/O but rather have the completion queued up to a completion function, an I/O completion port, or have a thread pool task listen on the event handle. When you do that, remember that you need to keep the `OVERLAPPED` structure as well as the `REQUEST_OPLOCK_INPUT_BUFFER` and `REQUEST_OPLOCK_OUTUT_BUFFER` structures valid until the I/O completes.

(You may find the `CancelIo` function handy to try to accelerate the clean-up of the file handle and any other actions that are dependent upon it.)

You can read more about opportunistic locks on MSDN. Note that there are limitations on explicitly-managed opportunistic locks; for example, they don't work across the network.

Raymond Chen

Follow