You can use an OVERLAPPED structure with synchronous I/O, too

devblogs.microsoft.com/oldnewthing/20120405-00

April 5, 2012

Raymond Chen

Even if you didn't open a file with `FILE_FLAG_OVERLAPPED`, you can still use the `OVERLAPPED` structure when you issue reads and writes. Mind you, the I/O will still complete synchronously, but you can take advantage of the other stuff that `OVERLAPPED` has to offer.

Specifically, you can take advantage of the `Offset` and `OffsetHigh` members to issue the I/O against a file location different from the current file pointer. (This is a file pointer in the sense of `SetFilePointer` and not in the sense of the C runtime `FILE*`.) If your program does a lot of reads and writes to random locations in a file, using the synchronous `OVERLAPPED` structure saves you a call to `SetFilePointer` at each I/O.

Let's illustrate this by writing some code to walk through a file format that contains a lot of offsets to other parts of the file: The ICO file format. First, the old-fashioned way:

```
#define UNICODE
#define _UNICODE
#include <windows.h>
#include <pshpack1.h>
struct ICONDIRHEADER {
    WORD idReserved;
    WORD idType;
    WORD idCount;
};
struct ICONDIRENTRY {
    BYTE bWidth;
    BYTE bHeight;
    BYTE bColorCount;
    BYTE  bReserved;
    WORD  wPlanes;
    WORD  wBitCount;
    DWORD dwBytesInRes;
    DWORD dwImageOffset;
};
#include <poppack.h>
BOOL ReadBufferAt(__in HANDLE hFile,
    __out_bcount(cbBuffer) void *pvBuffer,
    DWORD cbBuffer,
    DWORD64 offset)
{
 LARGE_INTEGER li;
 DWORD cbRead;
 li.QuadPart = offset;
 return SetFilePointerEx(hFile, li, nullptr, FILE_BEGIN) &&
        ReadFile(hFile, pvBuffer, cbBuffer, &cbRead, nullptr) &&
        cbBuffer == cbRead;
}
int __cdecl wmain(int argc, wchar_t **argv)
{
 HANDLE hFile = CreateFile(argv[1], GENERIC_READ,
  FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
  nullptr, OPEN_EXISTING, 0, nullptr);
 if (hFile != INVALID_HANDLE_VALUE) {
  ICONDIRHEADER hdr;
  if (ReadBufferAt(hFile, &hdr, sizeof(hdr), 0) &&
      hdr.idReserved == 0 && hdr.idType == 1) {
   for (UINT uiIcon = 0; uiIcon < hdr.idCount; uiIcon++) {
    ICONDIRENTRY entry;
    if (ReadBufferAt(hFile, &entry, sizeof(entry),
                     sizeof(hdr) + uiIcon * sizeof(entry))) {
     void *pvData = LocalAlloc(LMEM_FIXED, entry.dwBytesInRes);
     if (pvData) {
      if (ReadBufferAt(hFile, pvData,
                       entry.dwBytesInRes, entry.dwImageOffset)) {
       // process one image in the icon
      }
      LocalFree(pvData);
```

```
     }
    }
   }
  }
  CloseHandle(hFile);
 }
 return 0;
}
```

Run this program with the name of an icon file on the command line, and nothing interesting happens because the program doesn't generate any output. But if you step through it, you can see that we start by reading the `ICONDIRHEADER` to verify that it's an icon and determine the number of images. We then loop through the images: For each one, we read the `ICONDIR-ENTRY` (specifying the explicit file offset), then read the image data (again, specifying the explicit file offset).

We use the `ReadBufferAt` function to read data from the file. For each read, we first call `SetFilePointer` to position the file pointer at the byte we want to read, then call `Read-File` to read it.

Let's change this program to take advantage of our newfound knowledge:

```
BOOL ReadBufferAt(__in HANDLE hFile,
    __out_bcount(cbBuffer) void *pvBuffer,
    DWORD cbBuffer,
    DWORD64 offset)
{
 OVERLAPPED o = { 0 };
 o.Offset = static_cast<DWORD>(offset);
 o.OffsetHigh = static_cast<DWORD>(offset >> 32);
 DWORD cbRead;
 return ReadFile(hFile, pvBuffer, cbBuffer, &cbRead, &o) &&
        cbBuffer == cbRead;
}
```

We merge the `SetFilePointer` call into the `ReadFile` by specifying the desired byte offset in the optional `OVERLAPPED` structure. The I/O will still complete synchronously (since we opened the handle synchronously), but we saved ourselves the hassle of having to call two functions when it could be done with just one.

Raymond Chen

Follow