You can use a file as a synchronization object, too

devblogs.microsoft.com/oldnewthing/20140905-00

September 5, 2014

Raymond Chen

A customer was looking for a synchronization object that had the following properties:

- Can be placed in a memory-mapped file.
- Can be used by multiple processes simultaneously. Bonus if it can even be used by different machines simultaneously.
- Does not leak resources if the file is deleted.

It turns out there is already a synchronization object for this, and you've been staring at it the whole time: The file.

File locking is a very old feature that most people consider old and busted because it's just one of those dorky things designed for those clunky database systems that use tape drives like they have in the movies. While that may be true, it's still useful.

The idea behind file locking is that every byte of a file can be a synchronization object. The intended pattern is that a database program indicates its intention to access a section of a file by locking it, and this prevents other processes from accessing that same section of the file. This allows the database program to update the file without race conditions. When the database program is finished with that section of the file, it unlocks it.

One interesting bit of trivia about file locking is that you can lock bytes that don't even exist. It is legal to lock bytes beyond the end of the file. This is handy in the database case if you want to extend the file. You can lock the bytes you intend to add, so that nobody else can extend the file at the same time.

The usage pattern for byte-granular file locks maps very well to the customer's requirements. The synchronization object is… the file itself. And you put it in the file by simply choosing a byte to use as the lock target. (And the byte can even be imaginary.) And if you delete the file, the lock disappears with it.

Note that the byte you choose as your lock target need not be dedicated for use as a lock target. You can completely ignore the contents of the file and simply agree to use byte zero as the lock target. You just have to understand that when the byte is locked, only the owner of the lock can access it via the `ReadFile` and `WriteFile` family of functions. (Reading or writing a byte that is locked by somebody else will fail with `ERROR_LOCK_VIOLATION`. Note that access via memory-mapping is not subject to file locking, which neatly lines up with the customer's first requirement.)

To avoid the problem with locking an actual byte, you can choose imaginary bytes at ridiculously huge offsets purely for locking. Since those bytes don't exist, you won't interfere with other code that tries to read and write them. For example, you might agree to lock byte 0xFFFFFFFF`FFFFFFFF, on the assumption that the file will never become four exabytes in size.

File locking supports the reader/writer lock model: You can claim a lock for shared access (read) or for exclusive access (write).

The basic `LockFile` function is a subset of the more general `LockFileEx` function, so let's look at the general function.

To lock a portion of a file, you call `LockFileEx` with the range you want to lock, the style of lock (shared or exclusive), and how you want failed locks to be handled. To release the lock, you pass the *same range* to `UnlockFileEx`. Note that ranges cannot be chopped up or recombined. If you lock bytes 0–10 and 11–19 with separate calls, then you must unlock them with separate matching calls; you can't make a single bulk call to unlock bytes 0–19, nor can you do a partial unlock of bytes 0–5.

Most of the mechanics of locking are straightforward, except for the "how you want failed locks to be handled" part. If you specify `LOCKFILE_FAIL_IMMEDIATELY` and the lock attempt fails, then the call simply fails with `ERROR_LOCK_VIOLATION` and that's the end of it. It's up to you to retry the operation if that's what you want.

On the other hand, if you do not specify `LOCKFILE_FAIL_IMMEDIATELY`, and the lock attempt fails, then the behavior depends on whether the handle is synchronous or asynchronous. If synchronous, then the call blocks until the lock is acquired. If asynchronous, then the call returns immediately with `ERROR_IO_PENDING`, and the I/O completes when the lock is acquired.

The documentation in MSDN on how lock failures are handled is a bit confusing, thanks to tortured sentence structure like "X behaves like Y if Z unless Q." Here is the behavior of lock failures in table form:

	Handle type	
If `LockFileEx` fails	**Asynchronous**	**Synchronous**

`LOCKFILE_FAIL_IMMEDIATELY` specified	Returns `FALSE` immediately. Error code is `ERROR_LOCK_VIOLATION`.	
`LOCKFILE_FAIL_IMMEDIATELY` not specified	Returns `FALSE` immediately. Error code is `ERROR_IO_PENDING`. I/O completes when lock is acquired.	Blocks until lock is acquired, returns `TRUE`.

Here’s a little test app that exercises all the options. Run the program with two command line options. The first is the name of the file you want to lock, and the second is a string describing what kind of lock you want. Pass zero or more of the following letters:

- “o” to open an overlapped (asynchronous) handle; otherwise, it will be opened non-overlapped (synchronous).
- “e” to lock exclusively; otherwise, it will be locked shared
- “f” to fail immediately; otherwise, it will wait

For example, you would pass “ef” to open a synchronous handle and request an exclusive lock that fails immediately if it cannot be acquired. If you want all the defaults, then pass “” as the options.

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
int __cdecl _tmain(int argc, TCHAR **argv)
{
 // Ensure correct number of command line arguments
 if (argc < 3) return 0;
 // Get the options
 DWORD dwFileFlags = 0;
 DWORD dwLockFlags = 0;
 for (PTSTR p = argv[2]; *p; p++) {
  if (*p == L'o') dwFileFlags |= FILE_FLAG_OVERLAPPED;
  if (*p == L'e') dwLockFlags |= LOCKFILE_EXCLUSIVE_LOCK;
  if (*p == L'f') dwLockFlags |= LOCKFILE_FAIL_IMMEDIATELY;
 }
 // Open the file
 _tprintf(TEXT("Opening the file '%s' as %s\n"), argv[1],
          (dwFileFlags & FILE_FLAG_OVERLAPPED) ?
          TEXT("asynchronous") : TEXT("synchronous"));
 HANDLE h = CreateFile(argv[1], GENERIC_READ,
                FILE_SHARE_READ | FILE_SHARE_WRITE,
                NULL, OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL | dwFileFlags, NULL);
 if (h == INVALID_HANDLE_VALUE) {
  _tprintf(TEXT("Open failed, error = %d\n"), GetLastError());
  return 0;
 }
 // Set the starting position in the OVERLAPPED structure
 OVERLAPPED o = { 0 };
 o.Offset = 0; // we lock on byte zero
 // Say what kind of lock we want
 if (dwLockFlags & LOCKFILE_EXCLUSIVE_LOCK) {
  _tprintf(TEXT("Requesting exclusive lock\n"));
 } else {
  _tprintf(TEXT("Requesting shared lock\n"));
 }
 // Say whether we're going to wait to acquire
 if (dwLockFlags & LOCKFILE_FAIL_IMMEDIATELY) {
  _tprintf(TEXT("Requesting immediate failure\n"));
 } else if (dwFileFlags & FILE_FLAG_OVERLAPPED) {
  _tprintf(TEXT("Requesting notification on lock acquisition\n"));
  // The event that will be signaled when the lock is acquired
  // error checking deleted for expository purposes
  o.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
 } else {
  _tprintf(TEXT("Call will block until lock is acquired\n"));
 }
 // Okay, here we go.
 _tprintf(TEXT("Attempting lock\n"));
 BOOL fRc = LockFileEx(h, dwLockFlags, 0, 1, 0, &o);
 // If the lock failed, remember why.
 DWORD dwError = fRc ? ERROR_SUCCESS : GetLastError();
 _tprintf(TEXT("Wait %s, error code %d\n"),
          fRc ? TEXT("succeeded") : TEXT("failed"), dwError);
 if (fRc) {
  _tprintf(TEXT("Lock acquired immediately\n"));
 } else if (dwError == ERROR_IO_PENDING) {
  _tprintf(TEXT("Waiting for lock\n"));
  WaitForSingleObject(o.hEvent, INFINITE);
  fRc = TRUE; // lock has been acquired
 }
 // If we got the lock, then hold the lock until the
 // user releases it.
 if (fRc) {
  _tprintf(TEXT("Hit Enter to unlock\n"));
  getchar();
  UnlockFileEx(h, 0, 1, 0, &o);
 }
 // Clean up
 if (o.hEvent) CloseHandle(o.hEvent);
 CloseHandle(h);
 return 0;
}
```

When you run this program, it will try to acquire the lock in the manner requested, and if the lock is successfully acquired, it will wait for you press Enter, then it will release the lock.

You naturally need to run multiple copies of this program to see how the flags interact. (If you run only one copy, then it will always succeed.)

Exercise: What changes would you make if you wanted to wait at most 5 seconds to acquire the lock? (Hint.)

Raymond Chen

Follow