How can I append to a file and know where it got written, even if the file is being updated by multiple processes?

devblogs.microsoft.com/oldnewthing/20151127-00

November 27, 2015

Raymond Chen

A customer had a collection of processes, all of which are writing to a single file. Each process wants to append some data to a file and also know where the appended data was written, because the location of the appended record needed to be saved somewhere else. “We are currently using a named mutex derived from the path to the file. To add a new record, we take the mutex, set the file pointer to the end of the file, record the current position, write the data, then release the mutex. This works, but it feels clunky, and it is vulnerable to multiple names for the same file, or multiple computers trying to append to the same file. Is there a better way?”

Now, if the program needed to append data but didn’t care where it got appended, then it could make the file system do the work: Open the file for `FILE_APPEND_DATA | SYNCHRONIZE` and *nothing else*. (In particular, do not open for `FILE_WRITE_DATA`.) This is documented as meaning that the caller can write only to the end of the file, and any offset information provided in the write operation is ignored. Unfortunately, the technique doesn’t tell you where the data got written,[1] so it doesn’t help in this case.

This is a job for `LockFile`. In fact, this is not only a job for `LockFile`, this is precisely the job that `LockFile` was created to solve. The `LockFile` is so proud of its job that there’s even a sample program right there in MSDN showing how to use file locking to append data. But that sample isn’t quite the scenario we have here, because that sample assumes that only one process is writing (because it opens the file in deny-write mode), and it merely needs to lock out reads. In our case, we also want to permit others to write to the file, except when we are extending.

I sketched out a few different algorithms for the customer. First, you could agree that byte zero is the “I am appending” signal. This is merely using the file as its own synchronization object.

```
// Requires that everybody agree that byte 0 is the lock
AppendData()
{
 LockFile(from 0 to 0);
 size = GetFileSize;
 WriteAt(size, data);
 UnlockFile(from 0 to 0);
}
```

But choosing byte zero makes that byte of the file inaccessible while the lock is held, even though it is unrelated to the append operation. Therefore, you are probably better locking a nonexistent byte well beyond the anticipated maximum file size. Byte `0xFFFFFFFF`FFFFFFFF` will probably do nicely.

Better would be to use file locking in the way it was intended: To assert access to a range of bytes in the file because you actually want to access them. (File locking comes from the database world, where you would lock a record, perform an update, then unlock the record.)

```
AppendData()
{
 originalSize = GetFileSize;
 LockFile(from originalSize to 0xFFFFFFFF`FFFFFFFF);
 actualSize = GetFileSize;
 WriteAt(actualSize, data);
 UnlockFile(from originalSize to 0xFFFFFFFF`FFFFFFFF);
}
```

The idea here is that you lock the entire remainder of the file, from its current size out to infinity. If the file size changes before the lock, that's okay, because the file only grows in size, so we locked more than necessary.

If it's possible for the file to shrink in size, then you need to detect that case and expand the lock so that it covers the region you intend to write to.

```
AppendData()
{
 originalSize = GetFileSize;
 LockFile(from originalSize to 0xFFFFFFFF`FFFFFFFF);
 actualSize = GetFileSize;
 if (actualSize < originalSize) {
  UnlockFile(from originalSize to 0xFFFFFFFF`FFFFFFFF);
  originalSize = actualSize;
  LockFile(from originalSize to 0xFFFFFFFF`FFFFFFFF);
 }
 WriteAt(actualSize, data);
 UnlockFile(from originalSize to 0xFFFFFFFF`FFFFFFFF);
}
```

Or you can be sloppy and just lock the entire file. It's more expansive than you need, but it'll get the job done.

```
AppendData()
{
 LockFile(from 0 to 0xFFFFFFFF`FFFFFFFF);
 size = GetFileSize;
 WriteAt(size, data);
 UnlockFile(from 0 to 0xFFFFFFFF`FFFFFFFF);
}
```

The customer was okay with the sloppy version, and noted that using file locks also solves the problem of files with multiple names (due to hard links or network aliasing), as well as permitting multiple computers to operate on the file simultaneously.

[1] You might hope that the `OVERLAPPED.Offset` member would be updated with the actual file offset used, but sadly it isn't.

Raymond Chen

Follow