How to do atomic writes in a file

Adi Oltean 28 Dec 2005 5:51 PM 3

Let's assume that you want is to write some simple code that writes to a text file. A few assumptions:
1) You need avoid corruptions of any kind.
2) Either all of your writes have to make it to the disk, or none of them.
3) The file is updated serially - no concurrent updates from separate processes are allowed. So only one process writes to the file at a time.
4) No, you cannot use cool new techologies like TxF.

Remember, all you want is just to write to a text file - no fancy code allowed.

What are the possible problems?

Many people mistakenly think that writing to a file is an atomic operation. In other words, this sequence of function calls is not going to cause garbage in your file. Wrong. Can you guess why? (don't peek ahead for the response).

```
echo This is a string >> TestFile.txt
```

The problem is that the actual write operation is not atomic. A potential problem is when the machine reboots during actual write. Let's assume that your file write is ultimately causing two disk sectors to be overwritten with data. Let's even assume that each of these sectors is part of a different NTFS clusters, and these two clusters are part of the same TestfFile.txt file. The end of the first sector contains the string "This is" and the beginnning of the second sector "a string". What if one of the corresponding hardware write commands to write these sectors is lost, for example due to a machine reboot? You ended up with only one of these sectors overwritten, but not the other. Corruption!

Now, when the machine reboots, there will be no recovery at the file contents level. This is by design with NTFS, FAT, and in fact with most file systems, irrespective to the operating systems. The vast majority of file systems do **not** support atomicity in data updates. (That said, note that NTFS does have recovery at the metadata level - in other words, updates concerning file system metadata are always atomic. The NTFS metadata will not become corrupted during a sudden reboot)

The black magic of caching

So in conclusion you might end up with the first sector written, but not with the second sector. Even if you are aware of this problem you might still mistakenly think that the first sector is always written **before** the second one. In other words, assuming that "this is" is always written before "a string" in the code below:

```
using System;
using System.IO;

class Test
{
    public static void Main()
    {
        using (StreamWriter sw = new StreamWriter("TestFile.txt"))
        {
            sw.Write("This is");
            sw.Write("a string");
        }
    }
}
```

This assumption is again wrong. You again can have a rare situation where the machine crashes during your update, and "a string" can end up in the file, but "This is" not saved. Why?

One potential explanation is related with the caching activity. Caching happens at various layers in the storage stack. The .NET Framework performs its own caching in the Write method above. This can interfere with your actual intended order of writes.

So let's ignore .NET and let's present a second example, this time using pure Win32 APIs:

```
   WCHAR wszString1[] = "This is";
   WCHAR wszString2[] = "a string";

   fSuccess = WriteFile(hTempFile, wszString1, sizeof(WCHAR) * wcslen(wszString1),
&dwBytesWritten, NULL);
   if (!fSuccess)
      ...
   fSuccess = WriteFile(hTempFile, wszString2, sizeof(WCHAR) * wcslen(wszString2),
&dwBytesWritten, NULL);
```

Again, here you can also have caching at the operating system level, in the Cache Manager, where the file contents can be split across several in-memory data blocks. These blocks are not guaranteed to be written in their

natural order. For example, the lazy writer thread (a special thread used by Cache Manager that flushes unused pages to disk) can cause an out-of-order flush. There are other considerations that can cause an out-of-order data flush, but in general you need to be aware that any cache layers in your I/O can cause writes to be randomly reordered.

The same reasoning applies to our third example:

```
echo This is  >> TestFile.txt
echo a string >> TestFile.txt
```

Again, you cannot be sure that the file will not end up corrupted - you can have rare scenarios where the resultant file with contain either the word "This" or the word "string" but not both!

The solution? One idea is to use special write modes like FILE_FLAG_WRITE_THROUGH or FILE_FLAG_NO_BUFFERING, although in these cases you lose the obvious benefit of caching. You have to pass these flags to CreateFile(). Another idea is to manually flush the file contents through the FlushFileBuffers API.

So, how to do atomic writes, then?

From the example above, it looks like it is entirely possible that our writes migth complete partially, even if this case is extremely rare. How we can make sure that these writes are remaining atomic? In other words, my write to this file should either result in the entire write being present in the file, or no write should be present at all. Seems like an impossible problem, but that's not the case.

The solution? Let's remember that metadata changes are atomic. Rename is such a case. So, we can just perform the write to a temporary file, and after we know that the writes are on the disk (completed and flushed) then we can interchange the old file with the new file. Something like the sequence below (I used generic shell commands like copy/ren/del below but in reality you need to call the equivalent Win32 APIs):

Write process (on Foo.txt):
- Step W1: Acquire "write lock" on the existing file. (this is usually part of your app semantics, so you might not need any Win32 APIs here)
- Step W2: Copy the old file in a new temporary file. (copy Foo.txt Foo.Tmp.txt)
- Step W3: Apply the writes to the new file (Foo.Tmp.txt).
- Step W4: Flush all the writes (for example those being remaining in the cache manager).
- Step W5: Rename the old file in an Alternate form (ren Foo.txt Foo.Alt.txt)
- Step W6: Rename the new file into the old file (ren Foo.Tmp.txt Foo.txt)
- Step W7: Delete the old Alternate file (del Foo.Alt.txt)
- Step W8: Release "write lock" on the existing file.

This solution has now another drawback - what if the machine reboots, or your application crashes? You end up either with an additional Tmp or Alt file, or with a missing Foo.txt but with one or two temporary files like Foo.Alt.txt or Foo.Tmp.txt). So you need some sort of recovery process that would transparently "revert" the state of this file to the correct point in time. Here is a potential recovery process:

Recovery from a crash during write (on Foo.txt):
- Step R1: If Foo.txt is missing but we have both Foo.Alt.txt and Foo.Tmp.txt present, then we crashed between Step W5 and Step W6. Retry from Step W6.
- Step R2: If Foo.txt is present but Foo.Tmp.txt is also present, then we crashed before Step W5. Delete the Foo.Tmp.txt file.
- Step R3: If Foo.txt is present but Foo.Alt.txt is also present, then we crashed between Step W6 and Step W7. Delete the Foo.Alt.txt file.

More and more problems...

The sequence of operations above looks good, but we are not done yet. Why? Sometimes shell operations like Delete, Rename can fail for various reasons.

For example, it might just happen that an antivirus or content indexing application randomly scans the whole file system once in a while. So, potentially, the file Foo.Tmp.txt will be opened for a short period which will cause either the step W7 or R1..R3 to fail due to the failed delete. And, not only that, but also Rename can fail if the old file already exists, and someone has an open handle on it. So even the steps W2 or W5 can fail too...

The fix would be to always use unique temporary file names. In addition, during the recovery process, we will want to clean up all the "garbage" from previous temporary file leftovers. So, instead of files like Foo.Tmp.txt or Foo.Alt.txt, we should use Foo.TmpNNNN.txt and Foo.AltNNNN.txt, together with a smart algorithm to clean up the remaining "garbage" during recovery. Here is the overall algorithm:

Write process (on Foo.txt):
- Step W1: Acquire "write lock" on the existing file.
- Step W2: Copy the old file in a new unique temporary file. (copy Foo.txt Foo.TmpNNNN.txt)
- Step W3: Apply the writes to the new file (Foo.TmpNNNN.txt).
- Step W4: Flush all the writes (for example those being remaining in the cache manager).
- Step W5: Rename the old file in a new unique Alternate form (ren Foo.txt Foo.AltNNNN.txt)
- Step W6: Rename the new file into the old file (ren Foo.TmpNNNN.txt Foo.txt)
- Step W7: Delete the old Alternate file (del Foo.AltNNNN.txt). If this fails, simply ignore. The file will be deleted later during the next recovery.

- Step W8: Release "write lock" on the existing file.

Recovery from a crash during write (on Foo.txt):
- Step R1: If Foo.txt is missing but we have both Foo.AltNNNN.txt and Foo.TmpNNNN.txt present, then we crashed between Step W5 and Step W6. Retry from Step W6.
- Step R2: If Foo.txt is present but Foo.TmpNNNN.txt is also present, then we crashed before Step W5. Try to delete all Foo.TmpNNNN.txt files and ignore failures.
- Step R3: If Foo.txt is present but Foo.AltNNNN.txt is also present, then we crashed between Step W6 and Step W7. Try to delete all Foo.AltNNNN.txt files and ignore failures.

That's it!

Comments

Brad Gessler 28 Dec 2005 6:32 PM #

So does the transactional file system that is to be included with Vista abstract this 'mess' away from developers? Or when they want to write atomicly to a file, do they still need to worry about all of the above?

As a side note, you wouldn't believe the problems I have with 'developers' that don't know about atomicity on a larger scale. I'm currently in charge of maintaining an MS Access... <cring>yes, Access application that translates data from a legacy mainframe database into an MS SQL database. The developer who created the "solution" paid no attention to atomicity which has caused all sorts of headaches (ie: when the application fails, and this sucker fails, where did it fail?). I suppose I shouldn't expect a developer who chose Access as a data transformation engine would not understand this concept. :)

Adi Oltean 28 Dec 2005 7:06 PM #

>>> So does the transactional file system that is to be included with Vista abstract this 'mess' away from developers? Or when they want to write atomicly to a file, do they still need to worry about all of the above?

Yes. TxF allows you to write to the file system while preserving ACID semantics (including atomic writes). So you don't need all this mess with TxF.

In addition, you can also have transactions that span multiple resource managers. For example you can have a transaction that contains some SQL commands and some unrelated file system operations.

余啊雷 26 Mar 2008 2:04 PM #

PingBack from http://izumi.plan99.net/blog/index.php/2008/03/26/making-pstore-reaaaally-fast/