How do I delete bytes from the beginning of a file?

devblogs.microsoft.com/oldnewthing/20101201-00

December 1, 2010

Raymond Chen

It's easy to append bytes to the end of a file: Just open it for writing, seek to the end, and start writing. It's also easy to delete bytes from the end of a file: Seek to the point where you want the file to be truncated and call `SetEndOfFile`. But how do you delete bytes from the beginning of a file?

You can't, but you sort of can, even though you can't.

The underlying abstract model for storage of file contents is in the form of a chunk of bytes, each indexed by the file offset. The reason appending bytes and truncating bytes is so easy is that doing so doesn't alter the file offsets of any other bytes in the file. If a file has ten bytes and you append one more, the offsets of the first ten bytes stay the same. On the other hand, deleting bytes from the front or middle of a file means that all the bytes that came after the deleted bytes need to "slide down" to close up the space. And there is no "slide down" file system function.

One reason for the absence of a "slide down" function is that disk storage is typically not byte-granular. Storage on disk is done in units known as *sectors*, a typical sector size being 512 bytes. And the storage for a file is allocated in units of sectors, which we'll call *storage chunks* for lack of a better term. For example, a 5000-byte file occupies ten sectors of storage. The first 512 bytes go in sector 0, the next 512 bytes go in sector 1, and so on, until the last 392 bytes go into sector 9, with the last 120 bytes of sector 9 lying unused. (There are exceptions to this general principle, but they are not important to the discussion, so there's no point bringing them up.)

To append ten bytes to this file, the file system can just store them after the last byte of the existing contents. leaving 110 bytes of unused space instead of 120. Similarly, to truncate those ten bytes back off, the logical file size can be set back to 110, and the extra ten bytes are "forgotten."

In theory, a file system could support truncating an integral number of storage chunks off the front of the file by updating its internal bookkeeping about file contents without having to move data physically around the disk. But in practice, no popular file system implements

this, because, as it turns out, the demand for the feature isn't high enough to warrant the extra complexity. (Remember: Minus 100 points.)

But what's this "you sort of can" tease? Answer: Sparse files.

You can use an NTFS sparse file to decommit the storage for the data at the start of the file, effectively "deleting" it. What you've really done is set the bytes to logical zeroes, and if there are any whole storage chunks in that range, they can be decommitted and don't occupy any physical space on the drive. (If somebody tries to read from decommitted storage chunks, they just get zeroes.)

For example, consider a 1MB file on a disk that uses 64KB storage chunks. If you decide to decommit the first 96KB of the file, the first storage chunk of the file will be returned to the drive's free space, and the first 32KB of the second storage chunk will be set to zero. You've effectively "deleted" the first 96KB of data off the front of the file, but the file offsets haven't changed. The byte at offset 98,304 is still at offset 98,304 and did not move to offset zero.

Now, a minor addition to the file system would get you that magic "deletion from the front of the file": Associated with each file would be a 64-bit value representing the *logical byte zero* of the file. For example, after you decommitted the first 96KB of the file above, the *logical byte zero* would be 98,304, and all file offset calculations on the file would be biased by 98,304 to convert from logical offsets to physical offsets. For example, when you asked to see byte 10, you would actually get byte 98314.

So why not just do this? The *minus 100 points* rule applies. There are a lot of details that need to be worked out.

For example, suppose somebody has opened the file and seeked to file position 102,400. Next, you attempt to delete 98,304 bytes from the front of the file. What happens to that other file pointer? One possibility is that the file pointer offset stays at 102,400, and now it points to the byte that used to be at offset 200,704. This can result in quite a bit of confusion, especially if that file handle was being written to: The program writing to the handle issued two consecutive write operations, and the results ended up 96KB apart! You can imagine the exciting data corruption scenarios that would result from this.

Okay, well another possibility is that the file pointer offset moves by the number of bytes you deleted from the front of the file, so the file handle that was at 102,400 now shifts to file position 4096. That preserves the consecutive read and consecutive write patterns but it completely messes up another popular pattern:

```
off_t oldPos = ftell(fp);
fseek(fp, newPos, SEEK_SET);
... do stuff ...
fseek(fp, oldPos, SEEK_SET); // restore original position
```

If bytes are deleted from the front of the file during the *do stuff* portion of the code, the attempt to restore the original position will restore the wrong original position since it didn't take the deletion into account.

And this discussion still completely ignores the issue of file locking. If a region of the file has been locked, what happens when you delete bytes from the front of the file?

If you really like this *simulate deleting from the front of the file by decommitting bytes from the front and applying an offset to future file operations* technique, you can do it yourself. Just keep track of the magic offset and apply it to all your file operations. And I suspect the fact that you can simulate the operation yourself is a major reason why the feature doesn't exist: Time and effort is better-spent adding features that applications couldn't simulate on their own.

[Raymond is currently away; this message was pre-recorded.]

Raymond Chen

Follow