The Old New Thing

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

1 Dec 2010 7:00 AM | 29

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.]

Blog - Comment List MSDN TechNet

Comments

Nerf
1 Dec 2010 8:00 AM
#

Sparse files also lead to some interesting backup situations. (My example is from unix, but the same theory applies)

So in some version of linux you would see this:

```
# ls -lh /var/log/lastlog

-r--------    1 root     root         1.1T Nov 23 16:39 /var/log/lastlog

# df -h /var

Filesystem            Size  Used Avail Use% Mounted on

/dev/Volume00/LogVol05

                      5.8G  868M  4.7G  16% /var
```

For those not familiar, that a 1.1 terabyte file on a filesystem that's only 5.8 gigabytes (and only 868 megabytes used).

But some backup software isn't slick enough to realize this is a sparse file, so it backs up all 1.1TB. You can imagine the fun it provides with restores.

The solution in this case it to ignore the file - it's not important enough to back up.

pete.d
1 Dec 2010 8:05 AM
#

"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." "

Just goes to show the danger of wandering away from home, leaving a blog on autopilot. :)

Surely (insert Leslie Nielsen homage joke here) if the file was originally 5000 bytes in length, and then 10 bytes are appended, then removed again, the logical file size winds up being "set back to 5000", rather than "to 110".

Bill P. Godfrey
1 Dec 2010 8:11 AM
#

My 20 year old self likes this concept of decommitting blocks in the middle of a file. He thinks it would be useful for deleting records in the middle of a file without moving blocks around.

My current 36 year old self uses a database library, but misses the days when he would design his own custom file formats.

The MAZZTer
1 Dec 2010 8:20 AM
#

@Nerf that reminds me of AV software that isn't aware of NTFS Junctions... Vista makes use of them to allow for new paths for user profiles while allowing poorly coded legacy

apps to work using the old XP paths, but AV software can end up in an infinite loop because (IIRC) %USERPROFILE%\AppData\Local\Application Data points to %USERPROFILE%\AppData\Local and so pre-Vista AV software, since most XP systems didn't have NTFS Junctions, would end up recursing infinitely until they overflowed their path string buffers and crashed.

The MAZZTer
1 Dec 2010 8:23 AM
#

@pete.d: He meant the size of the data stored in the last cluster, not the total file size.

man
1 Dec 2010 1:14 PM
#

man split

Alex Grigoriev
1 Dec 2010 1:16 PM
#

I'm afraid this horse already has been beaten to death. These questions "how do I insert bytes into beginning/middle of the file" are of "pls send me teh codez" class.

joh6nn
1 Dec 2010 1:55 PM
#

@Alex Grigoriev

hi thanx. you know how i do insert bytes into beginning/middle of the file? pls send me the codes. most welcome.

John Muller
1 Dec 2010 3:21 PM
#

I once made a 5 1/4 floppy that contained an infinite file. I manually edited the 'next cluster' data for one of the clusters to point to itself.

For another good laugh, and an excellent test for anti-virus software, lookup 'zip quine'

Kert
1 Dec 2010 4:39 PM
#

dd if=input.txt of=output.txt ibs=1 skip=number_of_bytes_to_skip

Falcon
1 Dec 2010 5:51 PM
#

@Dan Bugglin:

I thought those junctions had ACLs that denied List folder permission to Everyone. Raymond discussed this in his Windows Confidential column in the December 2009 issue of TechNet Magazine. There should be no loops or duplicate folders unless someone/something has messed with the ACLs.

Blake
1 Dec 2010 6:10 PM
#

"exciting data corruption scenarios" - I am going to have to use that phrase.

Crescens2k
1 Dec 2010 6:15 PM
#

@Falcon

The thing you missed in this equation is the backup privilege. Don't forget that can be used to access files the user normally doesn't have access to in order to back them up. The thing this privilege does is that when it is on, it will give you read privilege regardless of the ACL.

Joseph Koss
1 Dec 2010 6:44 PM
#

Remember that the history of Microsoft's file system is that its originally based off of CP/M's system.

Thats where we got 8.3 filenames, where we got drive letters, and also where we got the EOF marker in text files.. and this last one is on point with this topic.

CP/M's file system only stored the number of sectors (128 byte records) that a file consumed, so it did not even support "deleting bytes" from the end of a file.. at least not in the way we mean it here.

pete.d
1 Dec 2010 7:50 PM
#

@Dan Bugglin

I find it remarkable that you know what he meant. Did he actually tell you that, or are you psychic?

Regardless, whatever he meant, the text doesn't make sense as written. He _might_ have meant the number and written the wrong words, or he might have meant the words he wrote and written the wrong number. Either way, it's something that ought to be fixed.

Falcon
1 Dec 2010 10:14 PM
#

@Crescens2k:

Yeah, you're right. I guess it makes sense for AV to make use of the privilege as well.

Jonathan
2 Dec 2010 1:58 AM
#

Of course, if the FS were to associate a "logical zero" index to a file, it would have to account for it in all actions pertaining the file - interpreting file pointers, seek, lock, etc.

This is definitely not something I'd expect to be handled at the FS level.

@Bill P. Godfrey: ++

COW?
2 Dec 2010 5:32 AM
#

NTFS has a copy-on-write by way of single-instance-storage or volume-shadow-copy. If one could create such a copy-on-write that shared a tail of a file then one could create a new file that had the first X bytes removed very cheaply and without any races or data corruption scenarios.

f0dder
2 Dec 2010 5:50 AM
#

@Kert: doesn't apply.

1) you create a new file rather than updating the existing.

2) you copy all pre- and post-data, instead of supporting a fast "punch-holes" operation.

f0dder
2 Dec 2010 5:51 AM
#

@Kert: doesn't apply.

1) you create a new file rather than updating the existing.

2) you copy all pre- and post-data, instead of supporting a fast "punch-holes" operation.

Alex Hart
2 Dec 2010 6:12 AM
#

I don't see why you wouldn't open the file in write mode (truncate), copy the contents into volatile memory (into a character array), then write the file, starting at the desired offset. With files < 1 GB this really shouldn't be a huge issue, especially if you break larger files into chunks (like 50 MB at a time)

Pádraig Brady
2 Dec 2010 7:43 AM
#

From the UNIX angle, to replace the file efficiently & atomically you could:

(dd bs=1 skip=100; cat) <in >tmp && mv tmp in

For caveats on using mv for the atomic replace, see:
www.pixelbeat.org/.../unix_file_replacement.html

Note GNU/Linux will soon get arbitrary offset deallocation support with fallocate (fd, FALLOC_FL_PUNCH_HOLE, offset, len);

Gabe
2 Dec 2010 9:48 AM
#

Pádraig Brady: You're right about caveats -- your link doesn't even mention them all!

If you used your trick to shorten your web server's log file, you'd probably end up with new log entries going to the old log file, taking up space in a file that you can't see. Then when you restart the web server to make the old log file stop taking up space, it can't use the new shorter file because it has the wrong file ownership/permissions.

640k
2 Dec 2010 11:42 AM
#

1. Only 1 handle can be opened to a file with write access.

2. If a file is opened with write access, no other handle (read or write) can be done.

3. Setting start of file requires an opened write handle.

This is the simple readers-writers-problem, fundamental to concurrent data access. How on earth can you not have a clue about this?

Falcon
2 Dec 2010 8:14 PM
#

@640k:

Wouldn't points 1 and 2 depend on the sharing mode of open handles? According to MSDN docs, it IS possible to open multiple handles for write access. Whether this is desirable or not is another matter.

f0dder
4 Dec 2010 5:00 AM
#

@Alex Hart: works, but is much slower than if you could simply "punch holes" - and there's data-consistency issues if you want to handle crashes gracefully, and on top of that you'll need to have the file open in exclusive mode for the full duration of the move-data-down operation.

@Pádraig Brady: unless 'dd' can do that without *copying* all the remaining bytes, I wouldn't call it efficient - and indeed the mv is problematic for reasons mentioned by others. FALLOC_FL_PUNCH_HOLE sounds *very* interesting, though!

zzz
4 Dec 2010 12:02 PM
#

I don't see why anyone would want to delete bytes from the beginning.

However most average people want to split files, large media files that is, without copying them in the process.

As volume is X and the file is only slightly smaller than X, the problem that filesystem of any merit should be solving is to split that file at arbitrary point without copying more than 2 sectors of data in the process.

Similarly, you should have ability to merge smaller file into the beginning of the larger file without moving any bytes of the larger file. This is so that headers can be added to split files.

> [*Can you name any filesystems of any merit? Or are you saying that all file systems are meritless? -Raymond*]

Gabe
6 Dec 2010 1:15 AM
#

A common reason for wanting to delete bytes from the beginning of a file is a produce/consumer scenario. A program is producing data at the end of a persistent queue and one or more programs are consuming data at the beginning. Ideally the consumer could delete the data at the beginning of the file once it was done with it, leading to a file that was only the size of the unconsumed data in the queue.

When the NTFS team was faced with this problem while implementing the USN Journal, they came up with FSCTL_SET_ZERO_DATA. Ever-growing sparse files have their own problems, but those problems are negligible compared to those Raymond elucidated.

Pádraig Brady
7 Dec 2010 7:52 AM
#

@f0dder Oops right, I forgot the count=0 parameter to that dd command