The Old New Thing

Why do some file operations take file names and others take handles?

15 Jun 2009 10:00 AM | 29

Commenter Brian Friesen asks why some functions (like SetFileAttributes) take a file name, while others (like SetFileTime) take a handle and why we can't have two versions of every API, one for each pattern.

Second question first: No need to wait for the kernel folks to write such a function; you can already do it yourself!

```
// Following "pidgin Hungarian" naming convention, which appears
// to be dominant in <winbase.h>...
BOOL SetFileTimesByNameW(LPCWSTR lpFileName,
                         CONST FILETIME *lpCreationTime,
                         CONST FILETIME *lpLastAccessTime,
                         CONST FILETIME *lpLastWriteTime)
{
  BOOL fRc = FALSE;
  HANDLE hFile = CreateFileW(lpFileName,
                             FILE_WRITE_ATTRIBUTES,
                             FILE_SHARE_READ | FILE_SHARE_WRITE |
                             FILE_SHARE_DELETE,
                             NULL, OPEN_EXISTING,
                             FILE_ATTRIBUTE_NORMAL, NULL);
  if (hFile != INVALID_HANDLE_VALUE) {
    fRc = SetFileTime(hFile, lpCreationTime, lpLastAccessTime,
                      lpLastWriteTime);
    CloseHandle(hFile);
  }
  return fRc;
}
```

Actually, there wouldn't be two versions of every API but three: (1) handle-based, (2) ANSI, (3) Unicode. (Unless you decide to follow the lead of the NLS team and simply stop adding new ANSI interfaces, in which case you're back to two.)

Back to the first question: Why are there some functions that take file names and some functions that take handles? Because that's how MS-DOS did it back in 1983! The function to get file attributes took a file name, but the function to modify the file times took a handle. This convention carried over to Windows because Windows preserved the old MS-DOS calling convention for performing disk operations, even though by Windows for Workgroups was released in 1992, you weren't really talking to MS-DOS any more. All that was left was the interface.

The NT team rewrote the operating system from scratch, and under the NT model, practically nothing was done by name. Nearly everything was done by handle. For example, to delete a file, you opened a handle to it and set the "delete on close" attribute, and then you closed the handle.

Of course, when NT became *Windows* NT, they had to adapt their interface to the old Windows way of doing things, although (whisper) under the covers, they just open the file

and perform the operation on the handle, like our wrapper function above does.

Blog - Comment List MSDN TechNet

Comments

mvadu
15 Jun 2009 10:21 AM
#

In that case can't you just document the "under the hood" function, so code which used the file name based function will continue to work, and new code can use the handle based function?

Karellen
15 Jun 2009 10:22 AM
#

That's a really interesting example to choose, because unlike SetFileAttributes() and SetFileTime() which operate purely on the file "inode", DeleteFile() modifies the directory that contains the link to the file that you want to delete.

Some time ago, this wouldn't be a problem, but ever since NTFS implemented hard links, there must no longer be a 1:1 mapping between "inodes" and directory entries. Therefore you should not be able to get "the" filename, (i.e. the directory entry to delete) from a file handle, because a file can have multiple filenames.

Any idea how that might work?

Euro
15 Jun 2009 10:52 AM
#

@Karellen: A file handle is not an INode. An INode is an implementation artifact of certain file systems; file handles are file-system neutral. So, your perceived problem doesn't exist. The kernel tracks all the information it needs to find the exact directory entry on which the handle was opened.

Also, as far as I know NTFS has supported hard links since day one; am I wrong?

Karellen
15 Jun 2009 11:06 AM
#

"An INode is an implementation artifact of certain file systems;"

Hence "inode" being in quotes.

"The kernel tracks all the information it needs to find the exact directory entry on which

the handle was opened."

Oh. I thought that recent versions of NTFS allowed you to rename open files. That must not be the case if the above is true then?

Gabe
15 Jun 2009 11:35 AM
#

Since NT had POSIX compliance as a design requirement, NTFS has always supported hard links (implemented as multiple filenames for a given file, like 8.3 names). The filename is stored not just in the directory entry, but in a list in the "inode" itself also. Presumably when you open a file, it remembers which filename you used to open it so that it knows which one you want to delete.

I'm pretty sure that POSIX also requires open files to be rename-able, so that has always been possible too. In order to do this, the handle must be opened with the right sharing parameters.

KJK::Hyperion
15 Jun 2009 12:46 PM
#

Karellen: just like an open file in UNIX is composed of two distinct objects (file descriptor, representing how the file was opened, and file node, representing where the file is in the filesystem), an open file in NT is composed of three distinct objects: file object (filesystem-independent information on how the file was opened), cache control block (filesystem-dependent information on how the file was opened) and file control block (filesystem-dependent information). The name a file was opened with can be stored in the CCB

That said, NT has no equivalent of POSIX unlink. You'll see that Interix uses a rename/move instead, hiding delete-pending entries in a system directory under the drive root

Doug
15 Jun 2009 12:57 PM
#

File operations by "name" do have a race condition, or a potential to operate on the wrong file.

In a multi tasking system, some other process or thread could rename/delete the file, and put another in its place. Which may or may not be your intended target....

Leo Davidson
15 Jun 2009 1:28 PM
#

When copying lots of small files the SetFileAttributes calls (assuming you copy the attributes as well as the data) seem to take significant time.

(With large files it's less significant because most of the time is spend writing the actual data.)

I wonder if being able to set the attributes via the file handle (or as part of the CreateFile call) would speed things up?

Thinking about it naively, it seems like it should be a "free" operation to specify the attributes of a file as it's created. Instead we create the file with default attributes, write the data, then close the file; then we set the attributes which involves opening a new handle to the file and re-writing the metadata.

I may be wrong about how the filesystem and caching works, though. Perhaps there's another reason that copying attributes slows things down.

(When I say attributes I mean the timestamps etc., not just the HSRA etc. flags which you can specify in the CreateFile call.)

Oh, I just checked MSDN and I see there is a SetFileInformationByHandle which was added in Vista. Says it's part of the separately-downloaded Win32 FileID API Library, rather than the main SDK, which makes me wonder how it works (it could be even slower if it maps handle to name and then calls the normal SetFileAttributes API). Worth a try though.

> [*Even if you could set attributes at creation, that won't help you since the act of copying the file itself changes the attributes. (Last-modified time? Archive attribute?) -Raymond*]

Leo Davidson
15 Jun 2009 3:36 PM
#

[Even if you could set attributes at creation, that won't help you since the act of copying the file itself changes the attributes. (Last-modified time? Archive attribute?) -Raymond]

That's true. Would it work if you set the attributes after the Write and before the Close?

e.g. (These are meant to be low-level filesystem calls rather than Win32 API calls.)

h = Open(path);

Write(h);

SetAttr(h);

Close(h);

Assuming the above sequence works (which might be wrong), and that SetFileInformationByHandle is more-or-less a direct call to the filesystem (which might also be wrong), I'd expect it to be faster. That's based on my assumption that copying a file and then using Win32 SetFileAttributes is like doing this:

h = Open(path);

```
Write(h);

Close(h);

h2 = Open(path);

SetAttr(h2);

Close(h2);
```

Unless the extra Open and Close are trivial and it's the SetAttr itself which takes the time.

I've made a note-to-self to test out the performance at some point, assuming it works at all.

(Checking it works should be quick & easy but I've always found it time consuming to speed-test file-copy code as it takes lots of tests to average the results and you need to ensure each test copies fresh data rather than data cached by the previous test.)

> [*My gut feeling tells me that you're right but like you I haven't tested it. -Raymond*]

Random832
15 Jun 2009 3:54 PM
#

@Gabe "I'm pretty sure that POSIX also requires open files to be rename-able, so that has always been possible too." Yeah, but POSIX also only renames by name, so the same issue doesn't apply here.

tobi
15 Jun 2009 5:41 PM
#

"No need to wait for the kernel folks to write such a function; you can already do it yourself!"

wrong: by centralizing the api developer productivity is increased and wrong code is avoided. also compatibility is improved as the windows team can rewrite the function because they own it.

Leo Davidson
15 Jun 2009 6:04 PM
#

@tobi: I agree that, in a perfect world, the OS API would do everything for us -- and it's obviously better for something to be written and tested once than reinvented over and over -- but the world's not perfect what Raymond said is still true:

If you need such a function then there's no need to wait (a potentially infinite amount of time) for one to be provided since you can make it yourself.

(As it turns out, Vista already has a function for setting the filetime by handle but not everyone can use that yet and what Raymond said is still true in general for any other commands that take names rather than handles.)

I bet most of the people reading these posts have their own personal (or team) collection of wrapper code to make the things they do commonly easier to do. Sometimes you gotta wrap stuff yourself.

Yuhong Bao
15 Jun 2009 8:38 PM
#

"even though by Windows for Workgroups was released in 1992, you weren't really talking to MS-DOS any more"

Actually WfW 3.11 released in 1993.

Trivia: the WfW 3.11 feature of "32-bit file access" used what was really an early version of the Chicago (Win95) VFAT, IOS and IFSMGR subsystems. Thus it was the part of WfW 3.11 that was filled with the most bugs.

[*I added that link specifically to pre-empt you, but apparently you just interpret it as encouragement to duplicate what I already wrote... -Raymond*]

Karellen
16 Jun 2009 6:49 AM
#

"File operations by name have a race condition, or a potential to operate on the wrong file"

"POSIX also only renames by name"

Hmmm....that is interesting. Consider the following:

Process A: open file "foo"

Process B: rename "foo" to "bar"

Process A: rename open file "foo" to "baz"

So, on NT, step 3 operating by the file handle would succeed, and the file that was called "foo" will end up being called "baz".

On POSIX, step 3 operating by file name fails, and the file ends up being called "bar".

Fascinating. Now I'm trying to figure out which behaviour makes more sense.

Part of me is leaning towards POSIX, but I suspect that's just because it's the behaviour I'm most used to. Both definitely have their strengths and weaknesses.

Interesting to compare what happens if step 3 is a delete/unlink. Or if the original filename was in a "temp" directory, and the new filename is meant to be somewhere more permenant.

SuperKoko
16 Jun 2009 9:59 AM
#

> In that case can't you just document the "under the hood" function ...

It is documented.

http://msdn.microsoft.com/en-us/library/dd445543.aspx

http://msdn.microsoft.com/en-us/library/ms804363.aspx

[*Notice that the documentation you link to is from the DDK, not the SDK. This is information for driver authors, not for application authors. The functions are not part of Win32 and applications should not call them. (It's like injecting work orders into a subcontractor's task list instead of going through the builder.) -Raymond*]

Brian
16 Jun 2009 11:42 AM
#

Interesting, I always suspected that's what was happening, but I thought there was a possibility the functions which took a file name could have some short cuts in them which make them slightly faster than open/set/close.

Alexandre Grigoriev
16 Jun 2009 1:36 PM
#

@Leo Davidson,

I think if you do the following sequence:

h = Open(path);

Write(h);

// before the handle is closed:

SetFileAttributes(path,....);

Close(h);

Then SFA will be almost as fast as by-handle operation. Could anybody test that and confirm whether it's so?

Anonymous Coward
16 Jun 2009 7:47 PM

#

Thanks Raymond, articles like these and the linked article about 95 are what I read TONT for. Concise, clear and in an entertaining style, they always contain some new gem to find.

Anyway, back to the topic of this and (to a lesser extent) the linked article, it always strikes me that people who say things like 'Why does[n't] Windows do X? Windows sucks!' obviously never tried to investigate, using a debugger or whatever, to try to answer the question themselves. But then again, maybe they, knowing the answer, don't feel the need to bug the internet with it and all those left are those who didn't.

Craig Barkhouse (MS)
16 Jun 2009 9:29 PM
#

@Karellen "That's a really interesting example to choose, because unlike SetFileAttributes() and SetFileTime() which operate purely on the file "inode", DeleteFile() modifies the directory that contains the link to the file that you want to delete."

Actually SetFileAttributes() and SetFileTime() do operate on the parent directory in addition to the file itself. In NTFS, a file's timestamps and attributes are duplicated in the parent directory. (Also, as others have pointed out, there's absolutely nothing like an i-node in NTFS. The closest concept would be an MFT entry, but even this is considerably different.)

@Karellen "Some time ago, this wouldn't be a problem, but ever since NTFS implemented hard links, there must no longer be a 1:1 mapping between "inodes" and directory entries. Therefore you should not be able to get "the" filename, (i.e. the directory entry to delete) from a file handle, because a file can have multiple filenames."

You can get "the" filename for a handle, because the FILE_OBJECT associated with the handle remembers what name you used to open it. However, there's an interesting exception: Open-By-Id. If you open a file by ID, then you can only get "a" filename for a handle. The name you get is essentially randomly chosen from the different names the file has.

@Karellen: "Oh. I thought that recent versions of NTFS allowed you to rename open files."

I think this has always been possible. Note that it doesn't automatically succeed. If an app opens a handle with FILE_SHARE_DELETE sharing mode, then it is explicitly allowing the file to be deleted or renamed while the handle is still open. Otherwise a delete or rename will fail.

@Leo Davidson "Oh, I just checked MSDN and I see there is a SetFileInformationByHandle which was added in Vista. Says it's part of the separately-downloaded Win32 FileID API Library, rather than the main SDK"

I have no idea what you mean by that second part. SetFileInformationByHandle() is a Win32 function and you can simply call it. It's exported out of kernel32.dll. On prior versions of Windows, you can use the native API NtSetInformationFile() directly. It's exported out of ntdll.dll. In fact SetFileInformationByHandle() is not much more than a wrapper around NtSetInformationByHandle().

Craig Barkhouse (MS)
16 Jun 2009 9:34 PM
#

@myself "In fact SetFileInformationByHandle() is not much more than a wrapper around NtSetInformationByHandle()."

... a wrapper around NtSetInformationFile().

Yuhong Bao
16 Jun 2009 9:42 PM
#

[I added that link specifically to pre-empt you, but apparently you just interpret it as encouragement to duplicate what I already wrote... -Raymond]

Because it was inaccurate. WfW 3.1 in 1992 did not have 32-bit file access.

> [*Oh. But that doesn't explain the second half of your comment, which is basically a duplicate of what I wrote in my earlier article. -Raymond*]

Karellen
17 Jun 2009 10:51 AM
#

"Actually SetFileAttributes() and SetFileTime() do operate on the parent directory in addition to the file itself."

You mean the parent *directories* - plural - right? Because if a file has multiple hardlinks then it can exist in multiple directories, which means that... hmmm... setting file attributes is O(n) on the number of hard links to it because you have to update that many directory entries?

Also, when you say "duplicated in the parent directory", does that mean there's one copy of the data that is in some way attached to the file data (in the MFT?) and then another copy in each directory that has a link to the file? What if there are 3 links to a file? 2 from directory A and one from directory B. Does "A" maintain 2 copies of the file attributes, or just one?

So... uh... what are the benefits of all this complexity?

Craig Barkhouse (MS)
17 Jun 2009 1:56 PM
#

Karellen, it's just the parent directory that gets updated. Remember you don't just open "a file", you open "a link". Each link has a distinguished parent directory. (Exception as I mentioned earlier is opening by ID, but even then we designate an essentially random link to have been opened, so there is still a distinguished parent directory.) So it's an O(1)

operation. Not that it would be a huge deal to be O(#links), because the number of links tends to be quite small, often 1. Incidentally the update to both the file (i.e. the MFT entry) and the parent directory occur on handle close.

Your understanding of duplicated to the parent directory is correct. A directory entry consists of a name, timestamps, attributes, and sizes. (Look at the documentation for FindFirstFile... the WIN32_FIND_DATA struct corresponds roughly to what is stored in a directory entry.) If a file has two links in the same directory, it will indeed have two directory entries, with the respective names as well as duplicated copies of timestamps, attributes, and sizes. Upon handle close, only the entry corresponding to that link will get updated in the directory. This works fine for files with only one link (the vast majority of files). For files with multiple links it is a little bit quirky, as the "duplicated" information stored in the directories can get out of sync for the various links. However we also update the duplicated information for any given link when you open that link. Here, it's easy to see this behavior:

echo foo>linkA

mklink /h linkB linkA

dir (timestamps and sizes match)

echo bar>>linkB

dir (sizes + possibly timestamps differ)

type linkA

dir (duplicated info back in sync)

Think of the duplicated information as a sort of cache. The reason for the complexity is the same for any other cache: performance. This speeds up directory queries tremendously (such as "dir" above, or Windows Explorer) where auxiliary information such as timestamps, attributes, and sizes need to be displayed. If we didn't cache this in the directory, you'd have to actually open a handle to each and every file and query for this information, meaning you'd have to read MFT entries that possibly sprawl all over the disk. Trust me, this is a huge perf win in this scenario. If you are enumerating and really only care about the names, then it is not a perf win.

The upshot however is that the duplicated information should be treated as a hint, not as authoritative (and note it wouldn't be authoritative anyway even for a single-link file, as the file may have been modified right after you enumerated it). If an application really needs to be sure of this information, it must open a handle and query for it.

Karellen
17 Jun 2009 4:26 PM
#

"Trust me, this is a huge perf win in this scenario."

Yes, but it's *wrong*. If the "dir" program doesn't have to produce correct results, I can write a replacement that runs in zero time and doesn't hit the disk once!

Sacrificing correctness for speed? That really makes my skin crawl. Ewww.

Yuhong Bao
17 Jun 2009 7:05 PM
#

"Notice that the documentation you link to is from the DDK, not the SDK. This is information for driver authors, not for application authors."

But it does describe a function that is callable from user mode via ntdll.dll.

"The functions are not part of Win32"

True, they are part of the so called NT native API, callable from both user and kernel mode.

"applications should not call them. (It's like injecting work orders into a subcontractor's task list instead of going through the builder.)"

Would you say the same about any ntdll.dll function?

[*It depends. Is it in the "Windows Driver Kit" part of the documentation? Then it's for drivers. I fail to see how this is so difficult to understand. -Raymond*]

James
18 Jun 2009 12:43 AM
#

Of course, this just raises the question: why did some file operations in MS-DOS take file names and others take handles?

@Gabe:

NT had POSIX as a design requirement? Why is there no fork() then? Why is there the separate SFU package then? Or did they start off with that as a requirement but then relax it later?

[*Um, you sort of answered the question yourself. fork() exists - in the POSIX subsystem, which is now an optional component. NT supports POSIX; Win32 doesn't. -Raymond*]

Yuhong Bao
18 Jun 2009 1:45 AM
#

[It depends. Is it in the "Windows Driver Kit" part of the documentation? Then it's for drivers. I fail to see how this is so difficult to understand. -Raymond]

Indeed, most of these functions aren't even callable from user mode, I was referring to one specific section of the WDK.

At lease some of the function from that specific section even have this as a clue for how to call it from user mode:

"Note If the call to the ZwOpenEvent function occurs in user mode, you should use the

name "NtOpenEvent" instead of "ZwOpenEvent"."

[Um, you sort of answered the question yourself. fork() exists - in the POSIX subsystem, which is now an optional component. NT supports POSIX; Win32 doesn't. -Raymond]

Both Win32 and POSIX subsystems are implemented using a common NT native API implemented in ntdll.dll, which is the most direct path to kernel mode.

> [*Please do not encourage people to write Win32 programs that bypass the Win32 layer. Final warning. -Raymond*]

Yuhong Bao
18 Jun 2009 2:05 AM
#

[Please do not encourage people to write Win32 programs that bypass the Win32 layer. Final warning. -Raymond]

I will stop now, but not before I mention ...

Is that OK?

> [*"I will stop but not before I ..." = "I won't stop." You already got your final warning. -Raymond*]

Yuhong Bao
19 Jun 2009 2:23 PM
#

["I will stop but not before I ..." = "I won't stop." You already got your final warning. -Raymond]

OK, I may post the whole thing to off-road. It was about [deleted repeat of material that you keep promising to stop doing but somehow can't bring yourself to stop]

> [*That was your final warning. I'm pretty sure you're doing this on purpose now. -Raymond*]