Renaming a file is a multi-step process, only one of which is changing the name of the file

devblogs.microsoft.com/oldnewthing/20211022-00

October 22, 2021

Raymond Chen

A customer reported that the `ReadDirectoryChangesW` function was reporting changes too soon. No, it wasn't generating changes from the future, *à la* *Minority Report*. Rather, it generated rename notifications before the rename was complete.

The customer came to this conclusion because they observed their program behaving like this:

Thread 1	Thread 2
	`ReadDirectoryChangesW( FILE_NOTIFY_CHANGE_FILE_NAME)` .
Call `MoveFileEx` to rename a file.	
	`ReadDirectoryChangesW` reports a rename occurred.
	Tries to read the renamed file and gets `ERROR_SHARING_VIOLATION` .
`MoveFileEx` returns.	
	Tries to read the renamed file and succeeds.

The `ReadDirectoryChangesW` function reports the rename before the `MoveFileEx` function returns, and consequently before the rename has completed.

What's going on here?

Well, the first thing to observe is that the customer's conclusion doesn't match the evidence. Observe that the attempt to open the renamed file failed with `ERROR_SHARING_VIOLATION` , whereas they expected error would be `ERROR_FILE_NOT_FOUND` if the file

hadn't been renamed yet. The fact that they're getting `ERROR_SHARING_VIOLATION` means that the rename *really did occur*, but they are unable to access the renamed file due to a sharing violation.

Okay, let's look at how renaming a file is performed internally. It's a multi-step operation.

1. Open the file with `DELETE` permission.
2. Call `NtSetInformationFile` with `FileRenameInformation`.
3. Close the handle.

Opening with `DELETE` permission grants permission to rename the file. The required permission is `DELETE` because the old name is being deleted.

The call with `FileRenameInformation` is what actually renames the file, and it is here that the `ReadDirectoryChangesW` is signaled.

Now that the rename is complete, the handle can be closed.

It is technically correct for the `ReadDirectoryChangesW` to be signaled once the `NtSetInformationFile` is done, because the file is well and truly renamed.

Let's look at that sharing violation again. The customer explained that they tried to open the file by doing this:

```
std::ifstream file(path, std::ios::binary, _SH_DENYNO);
```

The `_SH_DENYNO` indicates that no sharing operations are denied. So why is sharing denied?

You were faked out by a flag name that makes sense in context, but has ended up being confusing due to the passage of time.

Let's look at those sharing flags in context:

Flag	Meaning	Mnemonic
`_SH_DENYRD`	Deny read, allow write.	Deny read.
`_SH_DENYWR`	Allow read, deny write.	Deny write.
`_SH_DENYRW`	Deny read, deny write.	Deny read and write.
`_SH_DENYNO`	Allow read, allow write.	Deny none.

The mnemonic for `_SH_DENYNO` is "Deny none", but the word "none" is only with the context of read and write. You could say that it denies neither country *nor* western.

The important sharing mode here is neither read nor write. It's `FILE_SHARE_DELETE`, which means "I'm okay with letting someone delete or rename the file while I have it open."[1] This is a sharing flag that programs really should be using more often than they do, and the fact that the C runtime doesn't give you an easy way to set this sharing flag may be a contributing factor.

If you call the `CreateFile` function directly, then you can pass the `FILE_SHARE_DELETE` sharing flag, and then you'll be able to open the file even before `MoveFileEx` cleans up its handle.

"So why not have `ReadDirectoryChangesW` wait until the handle is closed before raising the rename notification?"

Well, for one thing, the file really has been renamed as soon as the `NtSetInformationFile` is complete, so delaying the notification would be a little disingenuous. But seeing as it's just a small delay, maybe that's okay, seeing as the whole thing is a notification anyway, and notifications can be delayed for other reasons.

But the real reason is that delaying the notification until the close of the handle could delay it indefinitely. The caller is not required to close the handle immediately after the `NtSetInformationFile` returns. It could leave the handle open so it can perform other operations on the file. For example, maybe it's a log file that is being renamed while it is still being actively written to. That log file's new name takes effect immediately, but the handle won't be closed for a long time yet.

The customer confirmed that switching to a direct `CreateFile` with `FILE_SHARE_DELETE` allowed them to open and read the file immediately after it was renamed.

Moral of the story: Don't forget `FILE_SHARE_DELETE`. It lets you coexist with code that is deleting or renaming the file you are looking at.

[1] My colleague Malcolm Smith, whom I rely on for all things filesystem, notes that the name `FILE_SHARE_DELETE` is rather misleading. Because the fact that you opened the file for `FILE_SHARE_DELETE` prevents it from being deleted, even though you're allowing it! In Windows, when you mark a file for deletion, the deletion doesn't take effect until all outstanding handles are closed, and holding a file open for `FILE_SHARE_DELETE` means that the last handle isn't closed yet. What `FILE_SHARE_DELETE` does is allow the file to be opened by others in `DELETE` mode, which as it happens is a prerequisite for both deleting and renaming files.

Raymond Chen

Follow