What is the order in which a POSIX system clears the file locks that were not unlocked cleanly?

The POSIX specification for `fcntl()` states:

> All locks associated with a file for a given process shall be removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates.

Is this operation of unlocking the file segment locks that were held by a terminated process atomic per-file? In other words, if a process had locked byte segments B1..B2 and B3..B4 of a file but did not unlock the segments before terminating, when the system gets around to unlocking them, are segments B1..B2 and B3..B4 both unlocked before another `fcntl()` operation to lock a segment of the file can succeed? If not atomic per-file, does the order in which these file segments are unlocked by the system depend on the order in which the file segments were originally acquired?

The specification for `fcntl()` does not say, but perhaps there is a general provision in the POSIX specification that mandates a deterministic order on operations to clean up after a process that exits uncleanly or crashes.

c | posix | race-condition | file-locking | fcntl

edited **23 hours ago**

asked **23 hours ago**

Daniel Trebbien
7,484 1 5 27
100% accept rate

2 Just out of curiosity, why do you care? (Creating a situation that depends on the order of lock release takes some effort... Or so I thought.) – Nemo 23 hours ago

@Nemo: I extracted—at least I think I did—SQLite's file locking algorithms so that I could implement file locking-based transactions on arbitrary file contents. The algorithms are implemented with locks on three segments, and sometimes two of these are locked at the same time. One difference between my version and SQLite's is that I wait for file locks whereas they choose to re-try a certain number of times, sleeping in between tries. It appears to me that they don't block for locks because they want compatibility with Win98, but I am trying to determine if there is another reason. – Daniel Trebbien 22 hours ago

feedback

2 Answers

I don't think the POSIX specification stipulates whether the releasing of locks is atomic or not, so you should assume that it behaves as inconveniently as possible for you. If you need them to be atomic, they aren't; if you need them to be handled separately, they're atomic; if you don't care, some machines will do it one way and other machines the other way. So, write your code so that it doesn't matter.

I'm not sure how you'd write code to detect the problem.

In practice, I expect that the locks would be released atomically, but the standard doesn't say, so you should not assume.

answered **20 hours ago**

Jonathan Leffler
108k 6 82 202

Hello Jonathan. Thanks for your answer. I ended up reading section 2.9.7 which answers the question for regular files and when a process uncleanly closes a file descriptor associated with a file. However, there seem to be issues with the wording of the effects of process termination. – Daniel Trebbien 19 hours ago

Even though the specification requires certain behavior, I think that I will follow your advice to assume otherwise because there may be issues with a POSIX system that make it non-compliant. – Daniel Trebbien 19 hours ago

feedback

There's a partial answer in section 2.9.7, Thread Interactions with Regular File Operations, of the POSIX specification:

> All of the functions *chmod*(), *close*(), *fchmod*(), *fcntl*(), *fstat*(), *ftruncate*(), *lseek*(), *open*(), *read*(), *readlink*(), *stat*(), *symlink*(), and *write*() shall be atomic with respect to each other in the effects specified in IEEE Std 1003.1-2001 when they operate on regular files. If two threads each call one of these functions, each call shall either see all of the specified effects of the other call, or none of them.

So, for a regular file, if a thread of a process holds locks on segments of a file and calls `close()` on the last file descriptor associated with the file, then the effects of `close()` (including removing all outstanding locks on the file that are held by the process) are atomic with respect to the effects of a call to `fcntl()` by a thread of another process to lock a segment of the file.

The specification for `exit()` states:

> These functions shall terminate the calling process with the following consequences:
>
> - All of the file descriptors, directory streams[, conversion descriptors, and message catalog descriptors] open in the calling process shall be closed.
> - ...

Presumably, open file descriptors are closed *as if* by appropriate calls to `close()`, but unfortunately the specification does not say how open file descriptors are "closed".

The 2004 specification seems even more vague when it comes to the steps of *abnormal* process termination. The only thing I could find is the documentation for `abort()`. At least with the 2008 specification, there is a section titled Consequences of Process Termination on the page for `_Exit()`. The wording, though, is still:

> - All of the file descriptors, directory streams, conversion descriptors, and message catalog descriptors open in the calling process shall be closed.

edited **26 mins ago**

answered **19 hours ago**

Daniel Trebbien
7,484 1 5 27

feedback

Not the answer you're looking for? Browse other questions tagged c posix race-condition file-locking fcntl or ask your own question.

question feed