N2830: C2x `fopen("x")` and `fopen("a")`

Document #:	N2830
Date:	2021-10-11
Project:	Programming Language C

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

The C11 standard introduced normative wording standardising `fopen('x')`. Unfortunately the current wording allows multiple mutually incompatible implementation semantics which are not only non-portable, but would actively cause user data corruption and loss if left standardised as-is. It is expected that the upcoming C++ 23 IS will replicate C2x's definition when extending C++ `iostreams` to support exclusive file creation, so fixing this now is important.

I have also taken the opportunity to propose making `fopen('a')` atomic as this operation is guaranteed atomic on POSIX, and atomic file appends are a very useful guarantee. As you will see below, all the major implementations bar Microsoft's whose source code is available to me already implement `fopen('a')` atomically.

This paper stemmed from my comments at the London WG14 meeting in 2019 during the discussion of [N2357] *Change Request for fopen exclusive access*. I promised at the time to supply improved normative wording, which I did at the time but it would appear something went astray. This paper re-proposes the improved normative wording I supplied in 2019, hopefully in time for the C2x IS.

The principle difference between the proposal in this paper and the proposal from [N2357] *Change Request for fopen exclusive access* is that the latter proposed to refine rather than remove entirely the second choice of implementation for exclusive file open.

Contents

1 The issue **2**
- 1.1 `fopen('x')` . . . 2
- 1.2 `fopen('a')` . . . 3

2 Proposed improved wording **4**
- 2.1 7.21.5.3.5 . . . 4
- 2.2 7.21.5.3.6 . . . 4

3 Platform compatibility **5**
- 3.1 `fopen('x')` . . . 5
- 3.2 `fopen('a')` . . . 5

4 Acknowledgements **6**

5 References 6

1 The issue

1.1 `fopen('x')`

The current wording in [N2573] the current draft C2x standard from 7.21.5.3.5 is this:

> Opening a file with exclusive mode (`'x'` as the last character in the mode argument) fails if the file already exists or cannot be created. Otherwise, the file is created with exclusive (also known as non-shared) access to the extent that the underlying system supports exclusive access.

The problem with this wording is that it conflates two completely separate notions of 'exclusivity': (i) filesystem modification (ii) use semantics. Program code will be written to assume one, but on a particular platform you might get the other, and there is no way for the program code to portably detect which.

[N2357] *Change Request for fopen exclusive access* written by an Austin Working Group member gives a more abstract explanation, but to crystalise the issue more concretely, here are five different semantics compatible with the above wording:

1. (a) Create file, but only if there is no file currently there.
2. (a) Create file, replacing any already there.
   (b) Immediately unlink file.
   (c) Now file can only be used exclusively.
3. As Windows used to define 'exclusive' in its legacy C library:
   (a) Create file with `ShareMode = 0`, replacing any already there.
   (b) Any attempt by anyone else to open that file fails with an `AccessDenied` error code, which makes you think it's permissions on the file, but those are in fact totally independent and probably do allow access.
4. (a) Create file, replacing any already there.
   (b) Take an exclusive mandatory lock on the file.
   (c) Other open file calls succeed, but any i/o on the file blocks forever for no obvious reason.
5. The 'do both' approach:
   (a) Create file, but only if there is no file currently there.
   (b) Set permissions on the file so nobody else can access it.
   (c) Take an exclusive mandatory lock on the file.

There are in fact more than a dozen combinations of exclusive filesystem modification and exclusive file usage possible here, all compatible with the current wording.

If a C program asked for `fopen('x')` and was written assuming implementation semantics 1, you would see data loss if implementation semantics were actually 2. Furthermore, there is no race free way of detecting implementation semantics here: there is a TOCTOU race between examining the file system for the lack of a file entry and creating a file, because the filesystem can always be modified concurrently.

For comparison, this is what POSIX.2017[1] defines for `O_CREAT|O_EXCL`:

> `O_EXCL`: If `O_CREAT` and `O_EXCL` are set, `open()` shall fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist shall be atomic with respect to other threads executing open() naming the same filename in the same directory with `O_EXCL` and `O_CREAT` set. If `O_EXCL` and `O_CREAT` are set, and path names a symbolic link, `open()` shall fail and set `errno` to `[EEXIST]`, regardless of the contents of the symbolic link. If `O_EXCL` is set and `O_CREAT` is not set, the result is undefined.

I have aimed to replicate the POSIX definition of exclusive file open in the proposed replacement wording below. This would remove entirely the 'Otherwise, ...' choice of implementation going forth, and also strengthen the guarantees of the one remaining choice to match those of POSIX.

As you will see below, all major platform implementations of `fopen('x')` already implement the proposed new wording, so WG14 would be adjusting its specification to match existing practice.

1.2 `fopen('a')`

The current wording in [N2573] from 7.21.5.3.6 is this:

> Opening a file with append mode (`'a'` as the first character in the mode argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to the `fseek` function. In some implementations, opening a binary file with append mode (`'b'` as the second or third character in the above list of `mode` argument values) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

For comparison, this is what POSIX.2017[2] defines for `O_APPEND`:

> If the `O_APPEND` flag of the file status flags is set, the file offset shall be set to the end of the file prior to each write and no intervening file modification operation shall occur between changing the file offset and the write operation.

In other words, the POSIX definition adds a requirement of *atomicity* to file appends i.e. two processes with the same file opened for append if they both write to that file concurrently, the writes are guaranteed to never be interleaved.

[1] https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/functions/open.html
[2] https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/functions/write.html

This is a very useful property: imagine a log file shared between multiple processes as an example. You can actually implement a multi-entity file-based mutual exclusion lock which works over network file systems using only atomic appends[3]. If C could tighten its definition for append-only files to require atomicity, this would be a useful improvement.

Obviously C's `FILE` when referring to a seekable file is buffered by default, so exactly when a true write occurs can be somewhat later, or more partial, than the writes via `fwrite()`. However that buffering is well specified by `setbuf()` and `setvbuf()`, so I propose an enhanced wording for `fopen('a')` below adding in atomicity which can be accepted or rejected by WG14 independently of changes for `fopen('x')`.

As you will see below, most implementations of `fopen('a')` targeting POSIX probably use POSIX `O_APPEND`, and therefore already implement this proposed change.

2 Proposed improved wording

2.1 7.21.5.3.5

Opening a file with exclusive mode (`'x'` as the last character in the mode argument) fails if the file already exists or cannot be created. ~~Otherwise, the file is created with exclusive (also known as non-shared) access to the extent that the underlying system supports exclusive access.~~ The check for the existence of the file and the creation of the file if it does not exist will be atomic with respect to other threads executing `fopen` upon the same file, if `'x'` is also specified to that `fopen`. If the implementation is not capable of performing the check for the existence of the file and the creation of the file atomically, it must fail instead of performing a non-atomic check and creation.

> [*Note:* The last sentence is important: if a program is written assuming that the check is atomic, and it is not atomic, then data loss or corruption would occur. It is better to return an error here so the program can adapt rather than silently allow data loss or corruption. – end note]

2.2 7.21.5.3.6

Opening a file with append mode (`'a'` as the first character in the mode argument) causes all subsequent writes to the file to be forced to the then current end-of-file at the point of buffer flush or actual write, regardless of intervening calls to the `fseek` function. The incrementing of the current end-of-file by the amount of data written will be atomic with respect to other threads executing writes upon the same file if it was also opened in append mode. If the implementation is not capable of performing the incrementing of the current end-of-file atomically, it must fail instead of performing non-atomic end-of-file writes. In some implementations, opening a binary file with append mode (`'b'` as the second or third character in the above list of `mode` argument values) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

[3] https://ned14.github.io/llfio/classllfio__v2__xxx_1_1algorithm_1_1shared__fs__mutex_1_1atomic_append.html

[*Note:* This text only guarantees the atomicity of the increment of the end of file, NOT the atomicity of the write of the data. This difference is important: no additional locking is needed here on platforms capable of atomic integer increment. – end note]

3 Platform compatibility

I checked whether the proposed new wording would break any existing platforms implementing C11:

3.1 `fopen('x')`

- Linux (glibc): Existing implementation is compatible.
- FreeBSD: Existing implementation is compatible.
- NetBSD: Existing implementation is compatible.
- OpenBSD: Existing implementation is compatible.
- MacOS: Existing implementation is compatible.
- Microsoft VS2019: `fopen('x')` not supported. Win32 `CreateFile()` is compatible via flag `CREATE_NEW`.
- QNX: `fopen('x')` not supported. `open()` is compatible.
- HPUX: `fopen('x')` not supported. `open()` is compatible.

The excellent compatibility story here is almost certainly due to POSIX `O_EXCL` creating an easy choice for how to implement `fopen('x')`.

3.2 `fopen('a')`

- glibc implements `fopen('a')` as `O_APPEND`, so appends are atomic across the system as per the proposed wording. https://sourceware.org/git/?p=glibc.git;a=blob;f=libio/fileops.c;h=0986059e7b16f885f8ab62bc9hb=HEAD#l237.
- BSD libc implements `fopen('a')` as `O_APPEND`, so appends are atomic across the system as per the proposed wording. https://svnweb.freebsd.org/base/head/lib/libc/stdio/flags.c?revision=326025&view=markup#l72
- Microsoft UCRT implements `fopen('a')` as `_O_APPEND`:

```
    case 'a':
        result._lowio_mode = _O_WRONLY | _O_CREAT | _O_APPEND;
        result._stdio_mode = _IOWRITE;
        break;
```

Then:

```
    // Set FAPPEND flag if appropriate. Don't do this for devices or pipes:
    if ((options.crt_flags & (FDEV | FPIPE)) == 0 && (oflag & _O_APPEND))
        _osfile(*pfh) |= FAPPEND;
```

Then:

```
    if (_osfile(fh) & FAPPEND)
        (void)_lseeki64_nolock(fh, 0, FILE_END);
```

Which eventually calls Win32 `SetFilePointerEx()`. This means appends are atomic within the local process per file descriptor, but are not atomic per inode in the local process, nor atomic across the system.

I suspect that this is an implementation oversight considering there are two forms of whole system atomic append supported on Windows:

1. Win32 `CreateFile()` when opened with `GENERIC_READ | FILE_WRITE_ATTRIBUTES | STANDARD_RIGHTS_WRITE | FILE_APPEND_DATA` instead of `GENERIC_READ | GENERIC_WRITE` does perform atomic appends across the system.
2. Win32 `WriteFile()` when supplied with an offset to write value of all bits one will perform an atomic append for that specific write across the system.

The source code of other platform's `fopen()` implementation was not easily available to me, so I cannot say more about how those implement `fopen('a')`.

4 Acknowledgements

Thanks to Robert Secord for his help in drafting the proposed normative wording. Thanks to Aaron Ballman for coordinating the late submission of this paper, and reminding me of the existence of [N2357]. Thanks to Nick Stoughton for writing the original paper raising this issue.

5 References

[N2357] Stoughton, Nick
Change Request for fopen exclusive access
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2357.htm

[N2573] *C2x Working Draft*
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2573.pdf

[POSIX.2017] *The 2017 POSIX standard*
https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/functions/contents.html