

# P1883R0: Walkthrough of P1031's `file_handle` for LEWG-I

Document #: P1883R0  
Date: 2019-09-26  
Project: WG21 Programming Language C++  
Library Evolution Working Group Incubator  
Reply-to: Niall Douglas  
<[s\\_sourceforge@nedprod.com](mailto:s_sourceforge@nedprod.com)>

A step-by-step walkthrough with explanation of each API's rationale of a single class in [P1031] *Low level file i/o*, specifically `file_handle`, to help LEWG-I reach consensus on whether or not to continue the low level approach to modern i/o.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Background and Context</b>	<b>3</b>
<b>3</b>	<b>Design principles</b>	<b>4</b>
<b>4</b>	<b>Walkthrough of creation, flags, options</b>	<b>4</b>
4.1	Mode flags	4
4.2	Creation flags	5
4.3	Kernel caching	6
4.3.1	Why cache control is important to higher level i/o layers	8
4.4	Handle flags	9
4.4.1	<code>flag::unlink_on_first_close</code>	9
4.4.2	<code>flag::disable_safety_barriers</code>	10
4.4.3	<code>flag::disable_safety_unlinks</code>	10
4.4.4	<code>flag::disable_prefetching</code>	10
4.4.5	<code>flag::maximum_prefetching</code>	10
4.4.6	<code>flag::win_disable_sparse_file_creation</code>	10
4.4.7	<code>flag::win_create_case_sensitive_directory</code>	11
<b>5</b>	<b>Summary of filesystem path interpretation semantics</b>	<b>11</b>
<b>6</b>	<b>Walkthrough of <code>path_handle</code></b>	<b>13</b>
6.1	<code>path_handle</code>	13
6.2	Example of use of <code>path_handle</code>	13
<b>7</b>	<b>Walkthrough of <code>path_discovery</code></b>	<b>14</b>

<b>8</b>	<b>Walkthrough of <code>iovec_buffer</code></b>	<b>15</b>
<b>9</b>	<b>Walkthrough of <code>file_handle</code></b>	<b>15</b>
9.1	Type aliases	16
9.2	i/o request type	17
9.3	i/o result type	17
9.4	Constructors	18
9.5	Static constructors	20
9.5.1	Creating randomly named files	20
9.5.2	Creating randomly named temporary files	20
9.5.3	Creating anonymous temporary inodes	21
9.6	Observers	22
9.6.1	Retrieving the current path of an open handle	22
9.6.2	Retrieving the maximum scatter-gather buffer quantity	23
9.6.3	Retrieving a handle to the parent directory of an open handle	24
9.6.4	Retrieving the unique identifier of an open handle	25
9.6.5	Retrieving the current maximum extent of an open handle	25
9.6.6	Retrieving the current valid extents of an open handle	25
9.6.7	Miscellaneous observers	26
9.7	Operations	27
9.7.1	Relinking	27
9.7.2	Linking	28
9.7.3	Inode cloning	28
9.7.4	Unlinking	29
9.7.5	Truncating extents and setting a new maximum extent	29
9.7.6	Deleting specific allocated extents	30
9.7.7	Write reordering barriers	31
9.7.8	Extent guards	33
9.8	Scatter reading from a file offset	35
9.9	Gather writing to a file offset	37
<b>10</b>	<b>Conclusion</b>	<b>37</b>
<b>11</b>	<b>References</b>	<b>38</b>

## 1 Introduction

In Cologne, LEWG-I discussed [P1031] *Low level file i/o*, taking a number of straw polls. The single most obvious pattern was one of neutral opinion, much to the great frustration of the LEWG-I chair at the time. The feeling in the room at the time seemed to be one of mental overload, that the size and unfamiliarity of file i/o made it impossible to take a definitive opinion one way or another on a particular design choice.

To help progress review of P1031, I offered at that review to walk through a single class in P1031 in a standalone paper, explaining the rationale for each API and design choice in turn. This is that

paper.

## 2 Background and Context

iostreams has proved to be remarkably resilient within the C++ ecosystem, despite its age. Despite individual claims to the contrary, it can be persuaded to perform well for a wide variety of use cases, albeit that higher performance use cases involve bypassing much of iostreams, which was designed to perform well on the CPUs, hard drives and C++ of the early 1990s.

‘Modern i/o’ proposals therefore have a steep hill to climb. WG21 decided to adopt a low level i/o approach for one part of the Networking TS, in that `std::net::basic_socket` and the other socket i/o classes in `<experimental_socket>` provide a barebones low level i/o facility. However, nobody would be expected to use those classes directly in most use cases, as the Networking TS is designed around the assumption that all socket i/o will be asynchronously completed using an Executor.

P1031 is intended to be used directly by end users where that end user needs specific detailed control of i/o, and maximum possible performance, concurrency guarantees, and flexibility as offered by the host operating system. It therefore has more knobs and switches than would be typical in a standard C++ library. It does not do object serialisation nor text interpretation (e.g. Unicode). It generally does not allocate memory, it does not copy memory, it does not take mutexes nor spin locks (i.e. is wait free), and it does not throw dynamic exceptions. It does not (directly) understand threads. It is Freestanding C++ compatible, though most of the i/o classes would not be present on embedded platforms. It has been designed to work well at compile time, if WG21 chooses later to make file i/o available to `constexpr` code. Most i/o handles do not implement usermode buffering, so there is generally a one-one relationship between i/o operation upon the P1031 C++ class and the corresponding i/o operation with the kernel.

P1031 is specifically intended to be a ‘one stop shop’ for implementing higher level i/o libraries e.g. the putative forthcoming Ranges i/o proposal, the putative iostreams v1 refresh, or the putative iostreams v2 redesign, which are the three alternative ‘modern i/o’ proposals that this author is currently aware of. It is not the purpose of this paper to discuss those putative proposals, but only to mention that P1031 was designed to aid implementation of all those higher level i/o layers, and to provide ‘escape hatching’ for when the user of those higher level i/o layers needs more detailed control than the higher level i/o layer can provide.

P1031 does directly support coroutinised i/o, but only on very specific platforms where resumable i/o can be particularly efficiently implemented using OS-specific facilities. If the host OS is not one of those very specific platforms, no direct support for coroutinised i/o is provided in the standards proposal (the reference implementation library provides extensions), and a higher level i/o layer would have to use a threadpool Executor to emulate resumable i/o using blocking i/o LLFIO classes.

### 3 Design principles

1. No statistically measurable runtime overhead above that of the underlying syscalls, no matter what kind of i/o is performed (e.g. million buffer scatter-gather i/o).
2. Pass through as much of the host OS concurrent i/o concurrency guarantees as possible.
3. Designed around the POSIX race free filesystem path lookup extensions [POSIXext].
4. It should always be possible to avoid all memory copies between the C++ i/o and the final storage device, anywhere in the system.
5. Expose kernel i/o cache control and virtual memory control facilities.
6. Expose facilities to detect and avoid races on the filesystem, such that code which is completely free of races introduced by concurrent third party modification of the filesystem can be written, at least to the extent that the host OS permits.

### 4 Walkthrough of creation, flags, options

#### 4.1 Mode flags

When opening a handle or file descriptor, one can request fewer privileges in order to (a) reduce scope of operation (b) improve performance (c) work with read-only storage.

In the comments below, I show the corresponding Microsoft Windows or POSIX privileges requested as passed to `CreateFile()` or `open()`.

```
1  //!< The behaviour of the handle: does it read, read and write, or atomic append?
2  enum class mode : unsigned char // bit 0 set means writable
3  {
4      unchanged = 0,
5      none = 2,      //!< No ability to read or write anything, but can synchronise (SYNCHRONIZE or 0)
6
7      attr_read = 4,  //!< Ability to read attributes (FILE_READ_ATTRIBUTES|SYNCHRONIZE or 0_RDONLY)
8
9      attr_write = 5, //!< Ability to read and write attributes (FILE_READ_ATTRIBUTES|
10     //!< FILE_WRITE_ATTRIBUTES|SYNCHRONIZE or 0_RDONLY)
11
12     read = 6,      //!< Ability to read (READ_CONTROL|FILE_READ_DATA|FILE_READ_ATTRIBUTES|
13     //!< FILE_READ_EA|SYNCHRONIZE or 0_RDONLY)
14
15     write = 7,     //!< Ability to read and write (READ_CONTROL|FILE_READ_DATA|FILE_READ_ATTRIBUTES|
16     //!< FILE_READ_EA|FILE_WRITE_DATA|FILE_WRITE_ATTRIBUTES|FILE_WRITE_EA|
17     //!< FILE_APPEND_DATA|SYNCHRONIZE or 0_RDWR)
18
19     append = 9     //!< All mainstream OSs and CIFS guarantee this is atomic with respect to all
20     //!< other appenders (FILE_APPEND_DATA|SYNCHRONIZE or 0_APPEND)
21 };
```

The `attr_read` and `attr_write` modes are particularly useful on Microsoft Windows, which skips the expensive setting up of kernel resources to do i/o. This considerably improves handle open performance. On Linux, the kernel does not allocate kernel resources to do i/o until the first i/o is performed. There are many use cases for opening files without wanting to do i/o e.g. a directory contents listing display, where doubling or trebling your speed would be advantageous.

A yet-to-be-discussed naming alternative would be `metadata_read` and `metadata_write`, which may be more descriptive, but is longer to type.

`none` does have a use case interestingly, it opens extremely quickly as you can do nothing with it, and thus the kernel does almost no work. But if it opens, then you know that the file in question is openable i.e. *it exists*, even if not readable by the current user. This is very useful, as it is reliable, unlike POSIX's `access()`, and it is race free without TOCTOU attacks, because one can reopen the unprivileged handle with privileges with the hard guarantee that the referenced inode will be the same as that originally opened with null privileges.

`append` means that all writes to the handle will be performed at the current maximum extent of the file, independent of other concurrent writers to the file in any other process. The actual algorithm is that the maximum extent is atomically incremented by the size of the bytes to be appended, and the bytes to be written happen at some point later into the empty slot created by the maximum extent increment.

It is very important to note that **all** of the above modes are implemented by the kernel. We do not, for example, implement atomic appends in userspace. If the OS returns an error when you ask for something, we pass that error right back to the user, no emulation.

## 4.2 Creation flags

If trying to open a file or directory whose leafname does not exist, what should happen? Again, the comments show the corresponding Microsoft Windows or POSIX flags.

```
1  //! On opening, do we also create a new file or truncate an existing one?
2  enum class creation : unsigned char
3  {
4      open_existing = 0, //!< Filesystem entry must already exist (OPEN_EXISTING or 0)
5
6      only_if_not_exist, //!< Filesystem entry must NOT exist, and is atomically created by the
7                          //!< success of this operation (CREATE_NEW or 0_CREAT|0_EXCL)
8
9      if_needed,         //!< If filesystem entry exists that is used, else one is created
10                         //!< (OPEN_ALWAYS or 0_CREAT)
11
12      truncate_existing, //!< Filesystem entry must already exist. It is atomically truncated on open,
13                          //!< leaving creation date and unique identifier unmodified (TRUNCATE_EXISTING
14                          //!< or 0_TRUNC)
15
16      always_new         //!< A newly created inode is already created, replacing any existing
17                          //!< (CREATE_ALWAYS or <synthesised>)
18  };
```

The ability to atomically detect and replace/truncate existing inodes is crucial to the building of resilient filesystem code, and is not currently possible in either the standard C or C++ libraries<sup>1</sup>. Another major use case is lock files, via which one can serialise filesystem modifications with other concurrent processes.

The ability to retain the inode unique identifier when replacing inode contents is particularly important for writing race free filesystem code. Imagine two processes, each pinned to a specific set of inodes (the process of race free inode pinning is very expensive, but it is a once off investment), and the first process needs to race free update the inode's contents without causing read tearing for the second process. What one does is to open a new handle with `truncate_existing`, this atomically sets the maximum extent to zero on the current inode. The second process, if it attempts a read, will always see either zero bytes read, for which it would sleep and reattempt, or a complete and valid file. With the newly opened handle, the first process now writes a complete set of new content with a single write i/o. If the host OS implements acquire/release semantics for file content i/o, the second process will only ever see either a completely empty file, or a complete file, and never any torn reads nor writes. Note that no additional synchronisation would be needed.

Consider another situation, which is where the first process is working with the contents of a file, and a second process wishes to replace the content of the file without disturbing any current concurrent users. In this situation, the second process uses `always_new` to atomically create a new inode and unlink the existing inode. The unlinked inode will continue to exist until the last open handle to it is closed, and thus anyone reading that inode before the replacement will see the previous version of the file and none of the changes in the new inode.

On POSIX, `always_new` may be synthesised by the implementation as no standard POSIX flags exist. On Linux, the synthesis is easy, use `O_TMPFILE` and `linkat()`. On BSD, one would create a new inode using a unique name, and atomically rename it over the destination.

### 4.3 Kernel caching

The OS kernel offers extensive facilities for caching data within kernel memory for filesystem metadata and file and directory contents. When a handle is opened, one can tell the kernel how much i/o on that handle ought to be cached within system memory. Knowing this in advance, the kernel can employ different (sometimes radically different<sup>2</sup>) codepaths for i/o which will be more efficient than caching all i/o, and forcing it out to storage later (which implies an avoidable memory copy, plus eviction of other cached data better kept in cache).

Note that if you want to map file data into memory, this is equivalent to full caching, as RAM pages used in the map *are* the kernel page cache.

Just to be clear, none of these options involve C++ userspace caching for `file_handle`. They wholly and exclusively refer to kernel caching only, and the equivalent POSIX flags for `open()` are described in the comments below. The mention of `flag_disable_safety_barriers` is explained

---

<sup>1</sup>The 'e' modifier for `fopen()` does not do what one would think.

<sup>2</sup>For example, ZFS may employ a specialised fast hardware device for non-cached i/o which permits non-cached i/o to complete immediately, and ZFS will asynchronously flush data stored to that device to the main storage pool later.

shortly. Other kinds of i/o handles in LLFIO *may* implement userspace caching based on these flags.

```
1  //! What i/o on the handle may complete immediately due to kernel caching
2  enum class caching : unsigned char // bit 0 set means safety fsyncs enabled
3  {
4      unchanged = 0,
5      none = 1,           //!< No caching whatsoever, all reads and writes come from storage
6                          //!< (i.e. <tt>O_DIRECT|O_SYNC</tt>). Align all i/o to 4Kb boundaries
7                          //!< for this to work. <tt>disable_safety_barriers</tt> can be used
8                          //!< here i.e. safety barriers are enabled by default here.
9
10     only_metadata = 2,  //!< Cache reads and writes of metadata but avoid caching data
11                          //!< (<tt>O_DIRECT</tt>), thus i/o here does not affect other cached data
12                          //!< for other handles. Align all i/o to 4Kb boundaries for this to work.
13
14     reads = 3,          //!< Cache reads only. Writes of data and metadata do not complete until
15                          //!< reaching storage (<tt>O_SYNC</tt>).
16                          //!< <tt>disable_safety_barriers</tt> can be used here.
17
18     reads_and_metadata = 5, //!< Cache reads and writes of metadata, but writes of data do not
19                          //!< complete until reaching storage (<tt>O_DSYNC</tt>).
20                          //!< <tt>disable_safety_barriers</tt> can be used here.
21
22     all = 6,            //!< Cache reads and writes of data and metadata so they complete
23                          //!< immediately, sending writes to storage at some point when the kernel
24                          //!< decides (this is the default file system caching on a system).
25
26     safety_barriers = 7, //!< Cache reads and writes of data and metadata so they complete
27                          //!< immediately, but issue safety barriers at certain points. See
28                          //!< documentation for <tt>flag_disable_safety_barriers</tt>.
29
30     temporary = 8      //!< Cache reads and writes of data and metadata so they complete
31                          //!< immediately, only sending any updates to storage on last handle
32                          //!< close in the system or if memory becomes tight as this file is
33                          //!< expected to be temporary (Windows and FreeBSD only).
34 };
```

For all values where bit 0 is set (`caching::none`, `caching::reads`, `caching::reads_and_metadata`, `caching::safety_barriers`), LLFIO implements additional safety barriers at key points, where all dirty metadata and data is flushed to storage via `.barrier()` described later. The semantic requested is that the modification would be retrievable, without inconsistency, after a sudden power loss. This is supposed to be guaranteed by POSIX's `O_SYNC`, but some POSIX implementations implement `O_SYNC` as `O_DSYNC` in some corner cases. If an implementation does implement `O_SYNC` correctly, then issuing a write reordering barrier immediately afterwards will be fast (relative to the operation of metadata synchronisation), as no further metadata needs to be flushed.

These safety barrier points are *currently* the following on all platforms:

1. Truncation or extension of file length either explicitly or during file open. This prevents files with a mix of old and new content appearing after sudden power loss.
2. Closing of the handle either explicitly or in the destructor. This ensures that all modifications to the handle preceding close will appear after sudden power loss, however it also means that

closing a handle may block for a very significant period of time.

Additionally, on Linux only, to prevent sudden power loss corruption of file metadata these are the additional safety barrier points *currently* for ext(2|3|4):

3. On the parent directory whenever a file might have been created.
4. On the parent directory on file close.

These prevent a mismatch of file metadata to file contents after sudden power loss. ext4 shouldn't do this anyway thanks to its journal, but better safe than sorry.

The reason why LLFIO implements these barriers instead of requiring the user to do it by hand is because what is involved varies between OS and filesystem. If LLFIO implements the semantic requested, it can optimise based on runtime parameters in a way which would have to be (buggily) replicated by user code.

Not all filesystems support  `caching::none`  and  `caching::only_metadata` . Some may return an error, others ignore the request. This is outside the control of LLFIO.

Write reordering barriers, and thus safety barriers, may be completely ignored by a system, this is permitted by POSIX, and this is outside the awareness of LLFIO. In this situation, the sole and **only** mechanism for ensuring sudden power loss safety is  `caching::reads`  and  `caching::reads_and_metadata` , and code which must ensure sudden power loss safety under any circumstances must use those flags, and not rely on write reordering barriers working as described.

#### 4.3.1 Why cache control is important to higher level i/o layers

The proposed level of cache control detail will seem excessive to most WG21 readers, though in fact it represents a subset of what is available. The reason it is necessary is to enable efficient or power loss durable i/o for higher level i/o frameworks.

A key observation is that uncached i/o is quite similar to non-temporal memory loads and stores, uncached i/o does not impact content currently in the kernel page cache. This allows bulk i/o e.g. copying of a directory tree from A to B, to execute at the speed of the storage device without evicting all of the contents of the kernel page cache. This in turn enables the system to not swap heavily as it pages programs back in after the copy of a large directory tree.

Uncached i/o is more useful than that again, though. Imagine Ranges i/o, which would need to perform individual byte i/o efficiently. Calling a syscall per individual byte i/o would not be efficient, so a C++ userspace cache would be needed, like  `iostreams`  currently does. If working on a high end storage device, i/o will be a non-trivial fraction of  `memcpy()` 's performance. If Ranges i/o keeps say a 1Kb local buffer, every time that 1Kb local buffer is flushed to storage, that implies a  `memcpy()`  of 1Kb into the kernel page cache. That could lop perhaps 10-15% off i/o performance, which would make Ranges i/o not an option for users with high end storage.

Consider if Ranges i/o instead keeps a local buffer list of several 4Kb pages. When one buffer is full, it is asynchronously written to storage using uncached i/o. Uncached i/o usually is implemented as a direct DMA from userspace to the storage device, so no memory is copied. Once the uncached i/o



completes, it is returned to the free buffer list. Ranges i/o therefore maxes out the full performance of any possible storage device, at least at its QD1<sup>3</sup> level.

The reason why we cannot implement this sort of caching at a layer below Ranges i/o is because we don't know what the context of the i/o is, and only Ranges i/o would. For example, the above uncached i/o pattern only makes sense if we know for a fact that the data written will not be used by others soon. If it is the case that the data written *will* be used by other soon, then you must absolutely use cached i/o instead, so the kernel can service the reads of your writes as quickly as possible.

Only the highest level i/o implementation can know what the end user intends. That is where userspace caching ought to be implemented in my opinion. And that is why detailed kernel cache control is needed at the low i/o level.

## 4.4 Handle flags

There are a series of flags used to enable or disable default behaviours. Some of these are kernel-supported on some platforms but not others, so LLFIO creates consistency across platforms.

There is an assumption that some sort of proper constexpr bitfield enumeration has been added to C or C++, and that `file_handle::flag` would be of that bitfield.

### 4.4.1 `flag::unlink_on_first_close`

Upon the first `.close()` on an open handle, the **current** filesystem path entry for that open handle is immediately removed.

This may seem like an anodyne feature, however there is a lot more to it than appears on the surface. Firstly, on some platforms (e.g. Microsoft Windows), the delete-on-close semantic is implemented by the kernel upon each inode. Different semantics are applied by Windows to an inode after any handle to it is opened with delete-on-close semantics: caching algorithms change, subsequent handle opens upon that inode gain different semantics, what happens after sudden power loss changes, and so on.

Secondly, the unlink is *race free*. Whatever is exactly the current path of the originally opened file, invariant to third party concurrent changes to the filing system, is the file entry removed.

Thirdly, the facility to have a file unlinked on first close is a primitive semantic used by many filesystem algorithms. Within LLFIO alone, the shared filing system mutex implementation uses it extensively; path discovery uses it; storage profiles uses it; anonymous shared memory uses it – and most of the custom applications I have built for clients uses it. It is most commonly used where you need a temporary piece of file storage within some filesystem algorithm, and you want its public discoverability to last only as long as all users are using it i.e. you specifically want it to become undiscoverable as soon as any one concurrent user finishes using it.

---

<sup>3</sup>QD1 = queue depth 1. Many storage devices perform at their maximum at QD16 or QD32, and offer a fraction of their maximum performance at QD1. The standard workaround is to increase the buffer size from 4Kb, so 16Kb buffers would have the device run at QD4.

Indeed, it is exactly because it is such a useful primitive building block that Microsoft Windows implemented direct support for delete on close semantics into its kernel. The emulation on POSIX is not perfect, but it's close enough that most code works identically on all platforms.

#### 4.4.2 `flag::disable_safety_barriers`

This disables the safety barriers applied by LLFIO for some caching modes described earlier. This would be used for performance by applications which implement safety barriers manually.

#### 4.4.3 `flag::disable_safety_unlinks`

On some platforms, there is no direct syscall support for unlinking a file entry invariant to concurrent filesystem modification. On such platforms, race free file entry unlinking is emulated using a non-deterministic algorithm, whose average overhead is usually no more than 10%. If the end user wishes to achieve maximum performance file entry removal, they would open the handle with this flag.

Note that if not specifying this flag, opening a directory or path handle fetches the inode for the filesystem entry. If it fails to fetch the inode because the path refers to a device, this flag is automatically set so one can detect when a directory or path handle refers to a device, and not a directory.

#### 4.4.4 `flag::disable_prefetching`

Ask the OS to disable prefetching of data. This can improve random i/o performance.

#### 4.4.5 `flag::maximum_prefetching`

Ask the OS to maximise prefetching of data, possibly prefetching the entire file into the kernel cache. This can improve sequential i/o performance.

#### 4.4.6 `flag::win_disable_sparse_file_creation`

This flag can be present on non-Windows systems, where it will be ignored.

On Microsoft Windows, due to legacy compatibility reasons, newly created files on NTFS by default do not expose to the application their extents-based storage. As the extents-based storage model is the default on POSIX, to match POSIX semantics, LLFIO creates files on Windows with extents-based storage exposed to application code by setting the appropriate property on the inode after new inode creation. If an end user does not want this, they would pass in this flag.

#### 4.4.7 `flag::win_create_case_sensitive_directory`

This flag can be present on non-Windows systems, where it will be ignored.

On recent versions of Microsoft Windows, case insensitive filesystem entry lookup has been globally disabled via a system registry entry. This causes filesystem code written for POSIX to fail when run on Windows.

One can, on Microsoft Windows, create directories within which leafname lookups will be performed case sensitively, as-if by `memcmp()`. POSIX code works perfectly when run within such directories. However, the case sensitivity property applies to **all** Windows code which uses that directory, which may break some applications. It is therefore not applied by default, and must be explicitly requested when creating a directory.

## 5 Summary of filesystem path interpretation semantics

It should be noted that LLFIO exclusively consumes filesystem paths via [P1030] `std::filesystem::path_view`. Path view consumers define how they interpret a path view, and the following are how LLFIO interprets path views.

These are the path interpretation semantics applied to consuming path views by LLFIO on POSIX:

- `char` = Unix format paths, native filesystem encoding.
- `wchar_t` = Unix format paths (UTF-32). This is converted C++-side to the native filesystem encoding at the point of use, if necessary.
- `char8_t` = Unix format paths (UTF-8). Input must be valid UTF-8. This is converted C++-side to the native filesystem encoding at the point of use, if necessary.
- `char16_t` = Unix format paths (UTF-16). Input must be valid UTF-16. This is converted C++-side to the native filesystem encoding at the point of use, if necessary.
- `byte` = Unique variable width binary number identifier. POSIX does not currently implement a standard API for these kind of paths, but proprietary APIs exist for various filesystems and hardware devices (e.g. ZFS, Samsung KV-SSD).

These are the path interpretation semantics applied to emitting path views by LLFIO on POSIX:

- Directory enumeration produces either the native filesystem encoding in `char`, or a unique variable width binary number identifier in `byte`.

These are the path interpretation semantics applied to consuming path views by LLFIO on Microsoft Windows:

- `char` = Compatibility DOS format paths, narrow system encoding (program locale determined). Compatibility DOS format paths start with `X:\`, or no prefix at all. These call the ANSI editions of Win32 APIs.
- `wchar_t` = Compatibility DOS format paths, wide system encoding (UTF-16). These call the Unicode editions of Win32 APIs.
- `char16_t` = Compatibility DOS format paths in UTF-16. Input must be valid UTF-16. These call the Unicode editions of the Win32 APIs.
- `char8_t` = Compatibility DOS format paths in UTF-8. Input must be valid UTF-8. This is converted C++-side to UTF-16 at the point of use, and the Unicode editions of Win32 APIs are called.
- `char` = Extended DOS format paths, narrow system encoding (program locale determined). As per Win32 API documentation, extended DOS format paths are prefixed with `\\?\` or `\\.\`. These call the ANSI editions of Win32 APIs.
- `wchar_t` = Extended DOS format paths, wide system encoding (UTF-16). These call the Unicode editions of Win32 APIs.
- `char16_t` = Extended DOS format paths in UTF-16. Input must be valid UTF-16. These call the Unicode editions of the Win32 APIs.
- `char8_t` = Extended DOS format paths in UTF-8. Input must be valid UTF-8. This is converted C++-side to UTF-16 at the point of use, and the Unicode editions of Win32 APIs are called.
- `char` = NT format paths, narrow system encoding (program locale determined). This is a LLFIO-only extension, NT format paths are prefixed with `\\!\\`. Paths prefixed with this never use the Win32 APIs, only the NT kernel APIs.
- `wchar_t` = NT format paths, wide system encoding (UTF-16).
- `char16_t` = NT format paths in UTF-16. Input must be valid UTF-16.
- `char8_t` = NT format paths, UTF-8. This is converted C++-side to UTF-16 at the point of use.
- `byte` = Unique variable width binary number identifier (NTFS and ReFS permit a 128-bit key-value lookup of inodes, this may be accelerated in hardware by suitable storage devices).

These are the path interpretation semantics applied to emitting path views by LLFIO on Microsoft Windows:

- Directory enumeration produces either the native filesystem encoding in `wchar_t`, or a unique variable width binary number identifier in `byte`.

## 6 Walkthrough of `path_handle`

We need to cover `path_handle` before we can cover `file_handle`, because one cannot understand how to open a file handle without understanding `path_handle`.

`path_handle` is an anchor point to somewhere in the filing system, from where non-absolute path lookups can be performed invariant to changes in the path of the anchor point.

To understand, imagine a simple C++ program. If it opens files, it works out of its current working directory. Non-absolute file lookups are performed relative to the current working directory, as-if the current working directory's path is prepended to the non-absolute file path lookup.

A `path_handle` is like a current working directory, in the sense that non-absolute path lookups occur relative to the instantaneous current path of that inode right at this exact moment in time. A third party operating concurrently can be rapidly permuting the current path of the inode referenced by the path handle, and it will make absolutely no difference to filesystem path lookups relative to the path handle.

Syscall support for this kind of filesystem path lookup was proposed for POSIX in 2006 [POSIXext]. It was merged into POSIX.2017. All the major POSIX implementations, and at least one of the embedded system POSIX implementations, have implemented these syscalls for many years now, as does Microsoft Windows. Support is therefore excellent across the most popular platforms. Note that LLFIO does not *require* this support, if you try to perform path handle relative filesystem lookups on a system which does not implement the requisite syscalls, the call simply fails.

### 6.1 `path_handle`

Path handle is deliberately extremely simple. You can open a path handle upon some path, or upon a relative path anchored from another path handle instance, and you can close it. You can feed an open path handle reference to all other APIs, which if default constructed (i.e. is null), means don't do path lookup from this path handle. That's it.

Most kernels provide a special lightweight filesystem anchor open facility which is extremely fast, but the resulting handle has no privileges other than acting as a filesystem anchor. All kernels permit a `directory_handle` to act as a path handle, and in LLFIO `path_handle` is a base class for `directory_handle`. Some kernels permit a `file_handle` to act as a path handle, but the user would have to explicitly convert a file handle into a path handle for this to work.

`path_handle` is default constructible, move only (actually move relocating, see [P1029] *SG14 move = relocates*), cloneable<sup>4</sup>, swappable, and closeable. Nothing else is provided.

### 6.2 Example of use of `path_handle`

```
1 // Open /home/ned/foo as a path handle
2 llfio::path_handle base("/home/ned/foo");
```

<sup>4</sup>C++ copy construction is disabled as duplicating a handle with the kernel is expensive. You can copy a handle instance using `.clone()`.

```

3 llfio::file_handle fileA(base, "fileA");
4 llfio::file_handle fileB(base, "fileB");

```

Something not obvious is that path handle based filesystem lookups have much better worst case performance than absolute path filesystem lookups. This is because absolute path filesystem lookups which miss the kernel path cache must traverse all the inodes in the absolute path, usually taking and releasing a mutex upon each one. Hence, if we were opening a few hundred files in `/home/ned/foo`, the path handle based lookup method demonstrated above could<sup>5</sup> be markedly quicker, as the kernel could avoid traversing all the inodes in each component of the path.

## 7 Walkthrough of `path_discovery`

We also need to cover a relevant subset of `path_discovery` before we can cover `file_handle`, because some of the static file handle constructors use path discovery.

Path discovery examines, at runtime, the environment experienced by the program looking for filesystem paths suitable for various use cases. It probes those directories for suitability, as the paths declared to programs by the operating system are often unavailable, or don't have the characteristics they are supposed to. It then ranks those directories according to a set of heuristics, and will choose the highest ranked path for a series of static query functions. Path discovery normally runs once per process invocation, and statically caches its results. It can be asked to be rerun, and extra paths to check can be suggested by user code.

The only category of paths relevant to file handle constructors is:

```

1 namespace path_discovery
2 {
3     /*! \brief Returns a reference to an open handle to a verified temporary directory where files
4     created are stored in a filesystem directory, usually under the current user's quota.
5
6     This is implemented by iterating all of the paths returned by 'verified_temporary_directories()'
7     and checking what file system is in use. The following regex is used:
8
9     'btrfs|cifs|exfat|ext(2|3|4)|f2fs|hfs|jfs|lxf|nfs|nilf2|ufs|vfat|xfs|zfs|msdosfs|newnfs|ntfs|smbfs|
10     unionfs|fat|fat32'
11
12     The handle is created during 'verified_temporary_directories()' and is statically cached thereafter.
13     */
14     const path_handle &storage_backed_temporary_files_directory() noexcept;
15 }

```

As the comment mentions, the path handle returned is to somewhere on the filesystem suitable for temporary files whose storage quota is allocated against that of the current user running the program. On Ubuntu Linux, this is typically `$XDG_RUNTIME_DIR (/run/user/<uid>)`; on Microsoft Windows, this is typically `%TMP%` (`C:\Users\<user>\AppData\Local\Temp`).

<sup>5</sup>Kernels implement an inode path cache which avoids inode path traversal for recently used paths, however most implementations invalidate rather than update the entries upon modification. Therefore opening many files in a directory experiencing modification tends to hit the worst case frequently.

Unless one calls a function which calls `storage_backed_temporary_files_directory()`, no runtime path discovery is run i.e. only upon first invocation to a function requiring a path discovered path handle is path discovery performed.

## 8 Walkthrough of `iovec_buffer`

LLFIO differs from the Networking TS by insisting that scatter-gather buffers are layout compatible with those used by the host OS, which in POSIX's case is a contiguous array of `struct iovec`. For Linux and most POSIX implementations, `struct iovec` is defined to be:

```
1 struct iovec {
2     void *iov_base;    /* Starting address */
3     size_t iov_len;    /* Number of bytes to transfer */
4 };
```

By insisting upon layout compatibility, LLFIO guarantees that the compiler can optimise out the scatter gather buffer repacking which the Networking TS in its current presentation can silently do. This repacking particularly hurts use cases with long scatter gather buffer lists, which can be much longer for file i/o than socket i/o (though with recent NIC hardware evolution, this has been changing).

`iovec_buffer` is a C++ type however, though guaranteed to be a trivial type with standard layout (i.e. C compatible). It additionally implements:

- `.data()`
- `.size()`
- `.begin()`, `.cbegin()`, `.rbegin()`, `.crbegin()`
- `.end()`, `.cend()`, `.rend()`, `.crend()`
- ... and any other types and member functions which `span<byte>` would implement.

If the standard library implements `span<byte>` with a layout compatible with the host OS scatter gather buffer, `SYSTEMIOVECEQUIVALENT_buffer` can alias `span<byte>` instead of being a separate type.

## 9 Walkthrough of `file_handle`

File handles, like almost all the classes in LLFIO, are designed to be as lightweight as possible in their header interfaces, pushing as much implementation detail as possible into a library. That said, LLFIO implements a header-only edition, a static library edition, and a dynamic library edition, all with strong ABI guarantees.

## 9.1 Type aliases

```
1 class file_handle
2 {
3 public:
4     ///! The path type used by this handle
5     using path_type = filesystem::path;
6
7     ///! The file extent type used by this handle
8     using extent_type = unsigned long long;
9
10    ///! The memory extent type used by this handle
11    using size_type = size_t;
12
13    ///! The scatter buffer type used by this handle.
14    using buffer_type = <system iovec equivalent>;
15
16    ///! The gather buffer type used by this handle.
17    using const_buffer_type = <const system iovec equivalent>;
18
19    ///! The scatter buffers type used by this handle.
20    using buffers_type = span<buffer_type>;
21
22    ///! The gather scatter buffers type used by this handle.
23    using const_buffers_type = span<const_buffer_type>;
24
25    ///! The device id type
26    using dev_t = <system specific>;
27
28    ///! The inode id type
29    using ino_t = <system specific>;
30
31    ///! The path view type used by this handle
32    using path_view_type = path_view;
33
34    ///! The unique identifier type used by this handle
35    using unique_id_type = <system specific>;
```

Every handle type in LLFIO capable of i/o implements a `buffers_type` and a `.read()` member function if it is capable of synchronous and blocking reading, and a `const_buffers_type` and a `.write()` member function if it is capable of synchronous and blocking writing. For file handles, these are spans of `buffer_type` and `const_buffer_type`, but for other i/o handles in LLFIO these can be *very* different.

The `path_view` type comes from [P1030] `std::filesystem::path_view`.

The unique identifier type is some type large enough to uniquely identify this inode anywhere on this computer. On POSIX, which guarantees that an inode can be uniquely identified by combining the device and inode ids, it is literally those numbers joined together.



## 9.2 i/o request type

All i/o handle types in LLFIO which implement i/o always implement an i/o request type which is passed to `.read()` and `.write()`. This is done to avoid passing long lists of parameters to `.read()` and `.write()`, which introduces ABI flexibility problems. The i/o request type can be *very* different in other LLFIO i/o classes, but for file handle it is:

```
1  //! The i/o request type used by this handle. Guaranteed to be 'TrivialType' apart from
2  //! construction, and 'StandardLayoutType'.
3  template <class T> struct io_request
4  {
5      T buffers{};
6      extent_type offset{0};
7
8      constexpr io_request();
9      constexpr io_request(T _buffers, extent_type _offset);
10 };
```

The only extra i/o request information which file handle takes in addition to the scatter gather buffers is a file extent to perform the i/o at.

The buffers inside the `buffers_type` or `const_buffers_type` supplied in the i/o request will be modified by file handle's `.read()` and `.write()` to indicate which buffer was partially filled. You must ensure that they are stored somewhere which permits modification.

## 9.3 i/o result type

All i/o handle types in LLFIO which implement i/o usually implement an i/o result type which is returned by `.read()` and `.write()`. This permits the i/o handle type to be informed when the lifetime of the i/o result has ended, or simply to provide extra metadata associated with the i/o result e.g. 'this i/o was partially completed'.

For file handle, the i/o result type is:

```
1  //! The i/o result type used by this handle. Guaranteed to be 'TrivialType' apart from
2  //! construction, and 'StandardLayoutType'.
3  template <class T> struct io_result
4  {
5      T buffers{};
6
7      constexpr io_result();
8
9      //! Return the number of bytes transferred.
10     constexpr size_type bytes_transferred() const noexcept;
11
12     //! Return whether this i/o was incomplete.
13     constexpr bool is_incomplete() const noexcept;
14 };
```

Something important to mention is that for other i/o handle types, the buffers in the i/o result returned **may be completely different** to the buffers supplied to `.read()` and `.write()`. However,

for `file_handle` specifically, you have the extra guarantee that the buffers returned will be a subset of the buffers supplied, and that each buffer supplied will be completely filled before the subsequent buffer is filled.

File handle modifies each buffer to indicate the extent to which each was filled/drained. The span of buffers returned indicates the total number of buffers filled/drained. It is thus very important to always use the buffers returned, and not assume anything about the buffers input.

This choice of semantic is considered unfortunate by some, as it prevents the use of file handle `.read()` and `.write()` with static const scatter gather buffers, which in turn would mean that scatter gather buffers assembled by a future constexpr Reflection based serialisation layer could not be used directly, and would require to be copied beforehand.

However there is good reason for this choice. Some i/o handles can fail after partially completing the i/o. In this situation, one needs to return failure, but *also* indicate how much of the i/o was completed.

This situation applies to Microsoft Windows, which does not currently implement a general purpose scatter-gather i/o syscall. On Microsoft Windows, one must usually issue each buffer in the scatter gather buffer list manually, and thus any individual buffer i/o could fail. `file_handle` specifically guarantees that each buffer fill is done in turn, but other kinds of handle capable of file i/o issue all the buffer fills in parallel for improved performance.

As you will see below, there are convenience overloads for `.read()` and `.write()` which do take initialiser lists, but they are markedly more inefficient as they clone the buffer list, and should not be called from within high performance loops.

## 9.4 Constructors

```
1  //!< Default constructor
2  constexpr file_handle();
3
4  /*! Create a file handle opening access to a file on path.
5  \param base Handle to a base location on the filing system. Pass '{}' to indicate that path will
6  be absolute.
7  \param path The path relative to base to open.
8  \param _mode How to open the file.
9  \param _creation How to create the file.
10 \param _caching How to ask the kernel to cache the file.
11 \param flags Any additional custom behaviours.
12
13 \errors Any of the values POSIX open() or CreateFile() can return.
14 */
15 constexpr file_handle(const path_handle &base,
16                       path_view_type path,
17                       mode _mode = mode::read,
18                       creation _creation = creation::open_existing,
19                       caching _caching = caching::all,
20                       flag flags = flag::none) throws(file_io_error);
21
22 //!< Construct a handle from a supplied native handle
23 constexpr file_handle(native_handle_type h,
```

```

24         dev_t devid,
25         ino_t inode,
26         caching caching = caching::none,
27         flag flags = flag::none) noexcept;
28
29     //! No copy construction (use clone())
30     file_handle(const file_handle &) = delete;
31
32     //! No copy assignment
33     file_handle &operator=(const file_handle &) = delete;
34
35     //! Implicit move construction of file_handle permitted
36     constexpr file_handle(file_handle &&) noexcept;
37
38     //! Explicit conversion from handle and io_handle permitted
39     explicit constexpr file_handle(handle &&,
40                                   dev_t devid,
41                                   ino_t inode) noexcept;
42
43     //! Move assignment of file_handle permitted
44     file_handle &operator=(file_handle &&) noexcept;
45
46     //! Swap with another instance
47     void swap(file_handle &) noexcept;
48
49     /*! Clone this handle (copy constructor is disabled to avoid accidental copying),
50     optionally race free reopening the handle with different access or caching.
51
52     Microsoft Windows provides a syscall for cloning an existing handle but with new
53     access. On POSIX, if not changing the mode, we change caching via 'fcntl()', if
54     changing the mode we must loop calling 'current_path()',
55     trying to open the path returned and making sure it is the same inode.
56
57     \errors Any of the values POSIX dup() or DuplicateHandle() can return.
58     \mallocs On POSIX if changing the mode, we must loop calling 'current_path()' and
59     trying to open the path returned. Thus many allocations may occur.
60     */
61     file_handle clone(mode mode_ = mode::unchanged,
62                     caching caching_ = caching::unchanged,
63                     deadline d = std::chrono::seconds(30)) const throws(file_io_error);
64
65     //! Immediately close the native handle type managed by this handle
66     void close() throws(file_io_error);
67
68     //! Release the native handle type managed by this handle
69     native_handle_type release() noexcept;
70
71     /*! Changes whether this handle is append only or not.
72
73     \warning On Windows this is implemented as a bit of a hack to make it fast like on POSIX,
74     so make sure you open the handle for read/write originally. Note unlike on POSIX the
75     append_only disposition will be the only one toggled, seekable and readable will remain
76     turned on.
77
78     \errors Whatever POSIX fcntl() returns. On Windows nothing is changed on the handle.
79     \mallocs No memory allocation.

```

```

80  */
81  void set_append_only(bool enable) throws(file_io_error);

```

File handles can be invalid. This state occurs due to default construction, or after `.close()` has been called upon a open handle, or when a handle has been moved from. This mirrors the Networking TS.

File handles can adopt a native handle type, which allows externally created native handle types to be managed and used by file handle. They can also release from ownership their internal native handle type. This mirrors the Networking TS.

File handles are not implicitly copyable as duplication involves a kernel syscall. `.clone()`, which all LLFIO i/o handles implement, provides explicit copying. File handle extends cloning with optional reopening, so the duplicated file handle can have more, or less, privileges or caching than the source file handle. This permits race free downgrading, or upgrading, of whatever inode is currently opened by a handle, invariant to concurrent filesystem modification.

File handles can be flipped quickly between atomic append only, and normal offset-based read/write. This involves a syscall on POSIX, but none on Microsoft Windows.

## 9.5 Static constructors

### 9.5.1 Creating randomly named files

```

1  /*! Create a file handle creating a randomly named file on a path.
2  The file is opened exclusively with 'creation::only_if_not_exist' so it
3  will never collide with nor overwrite any existing file. Note also
4  that caching defaults to temporary which hints to the OS to only
5  flush changes to physical storage as lately as possible.
6
7  \errors Any of the values POSIX open() or CreateFile() can return.
8  */
9  static file_handle random_file(const path_handle &dirpath,
10                               mode _mode = mode::write,
11                               caching _caching = caching::temporary,
12                               flag flags = flag::none) throws(file_io_error);

```

The facility to create a randomly named sibling file is frequent enough to warrant a dedicated function to do it correctly, and indeed all of LLFIO's handle types implement a randomly named static creation function. The randomly chosen name is a 256-bit random number in hexadecimal with a guaranteed zero chance of accidentally overwriting an existing file entry.

### 9.5.2 Creating randomly named temporary files

```

1  /*! Create a file handle creating the named file on some path which
2  the OS declares to be suitable for temporary files. Most OSs are
3  very lazy about flushing changes made to these temporary files.
4  Note the default flags are to have the newly created file deleted
5  on first handle close.

```

```

6  Note also that an empty name is equivalent to calling
7  'random_file(path_discovery::storage_backed_temporary_files_directory())' and the creation
8  parameter is ignored.
9
10 \note If the temporary file you are creating is not going to have its
11 path sent to another process for usage, this is the WRONG function
12 to use. Use 'temp_inode()' instead, it is far more secure.
13
14 \errors Any of the values POSIX open() or CreateFile() can return.
15 */
16 static file_handle temp_file(path_view_type name = path_view_type(),
17                               mode _mode = mode::write,
18                               creation _creation = creation::if_needed,
19                               caching _caching = caching::temporary,
20                               flag flags = flag::unlink_on_first_close) throws(file_io_error);

```

A very common use case is to create a randomly named file inside one of the temporary directory path handles returned by path discovery. One then queries its current path, and transmits that path to a second process. Once the second process has opened the file, it is unlinked, thus forming a file shared between the processes with other processes cannot discover easily, and whose contents will be discarded when both processes have closed the file.

### 9.5.3 Creating anonymous temporary inodes

```

1  /*! \em Securely create a file handle creating a temporary anonymous inode in
2  the filesystem referred to by \em dirpath. The inode created has
3  no name nor accessible path on the filing system and ceases to
4  exist as soon as the last handle is closed, making it ideal for use as
5  a temporary file where other processes do not need to have access
6  to its contents via some path on the filing system (a classic use case
7  is for backing shared memory maps).
8
9  \errors Any of the values POSIX open() or CreateFile() can return.
10 */
11 static file_handle temp_inode(const path_handle &dirh = path_discovery::
12                               storage_backed_temporary_files_directory(),
13                               mode _mode = mode::write,
14                               flag flags = flag::none) throws(file_io_error);

```

This function is the core API for using files as a source for sparse dynamic memory allocations which exceed the paging limit of the system. All of the containers in LLFIO use this function to allocate backing storage which is guaranteed to be inaccessible to other processes, and which will always be thrown away on close or process exit.

Linux and Windows provide direct support for creating anonymous temporary inodes. On other systems, a randomly named file is created without the ability for anyone else to access it, and it is immediately unlinked.

On some systems (Linux, Windows), temporary anonymous inodes can be converted into a normal file by hard linking them to somewhere. See `.link()`, later.

## 9.6 Observers

### 9.6.1 Retrieving the current path of an open handle

```
1  /*! Returns the current path of the open handle as said by the operating system. Note
2  that you are NOT guaranteed that any path refreshed bears any resemblance to the original,
3  some operating systems will return some different path which still reaches the same inode
4  via some other route e.g. hardlinks, dereferenced symbolic links, etc. Windows and
5  Linux correctly track changes to the specific path the handle was opened with,
6  not getting confused by other hard links. MacOS nearly gets it right, but under some
7  circumstances e.g. renaming may switch to a different hard link's path which is almost
8  certainly a bug.
9
10 If LLFIO was not able to determine the current path for this open handle e.g. the inode
11 has been unlinked, it returns an empty path. Be aware that FreeBSD can return an empty
12 (deleted) path for file inodes no longer cached by the kernel path cache, LLFIO cannot
13 detect the difference. FreeBSD will also return any path leading to the inode if it is
14 hard linked. FreeBSD does implement path retrieval for directory inodes
15 correctly however, and see 'algorithm::cached_parent_handle_adapter<T>' for a handle
16 adapter which makes use of that.
17
18 On Linux if '/proc' is not mounted, this call fails with an error. All APIs in LLFIO
19 which require the use of 'current_path()' can be told to not use it e.g.
20 'flag::disable_safety_unlinks'.
21 It is up to you to detect if 'current_path()' is not working, and to change how you
22 call LLFIO appropriately.
23
24 \warning This call is expensive, it always asks the kernel for the current path, and no
25 checking is done to ensure what the kernel returns is accurate or even sensible.
26 Be aware that despite these precautions, paths are unstable and can change randomly at
27 any moment. Most code written to use absolute file systems paths is racy, so don't
28 do it, use 'path_handle' to fix a base location on the file system and work from that anchor
29 instead!
30
31 \mallocs At least one malloc for the 'path_type', likely several more.
32
33 \sa 'algorithm::cached_parent_handle_adapter<T>' which overrides this with an
34 implementation based on retrieving the current path of a cached handle to the parent
35 directory. On platforms with instability or failure to retrieve the correct current path
36 for regular files, the cached parent handle adapter works around the problem by
37 taking advantage of directory inodes not having the same instability problems on any
38 platform.
39 */
40 path_type current_path() const throws(file_io_error);
```

This API is probably the most contentious in feedback received from WG21, as it involves calling a proprietary syscall, as no support exists in POSIX for retrieving the current path of an open file descriptor. Perhaps much more contentious to some WG21 members is the claim that Linux tracks changes to the path as originally opened, and does not return any path by which the inode can be reached. Yet, I have supplied test programs which prove that Linux does do this. The response, so that readers are aware, is that Linux does not *guarantee* that its proprietary syscall is stable in this regard, and that it might change in the future.

Personally speaking, seeing as two of the major platforms make a convincing attempt to track

changes to the originally opened path, I think that good enough to merit standardising this behaviour:

- Renames change the current path of the open handle.
- Unlinks remove the current path of the open handle.
- Hard links do not affect the current path of the open handle unless it has no current path due to having been unlinked (i.e. you can give a name to an anonymous inode).

Implied in this is that changes to the originally opened path is tracked by the system, which in turn means that file descriptors remember which path they were opened with. Linux and Windows implement the above perfectly for file and directory inodes, FreeBSD and Mac OS implement this perfectly for directory inodes perfectly.

For those platforms which don't fully implement originally opened path tracking (FreeBSD), or don't implement it without bugs (Mac OS), I don't think it a big ask for those platforms to fix their kernels. For those with kernels completely lacking support, standard library implementations can implement originally opened path tracking for all programs using the standard library implementation using shared memory. For those systems without shared memory support, the chances are high that the end user controls all the programs able to run on the embedded system, so concurrent path changes are not a problem.

This is my opinion, though I am very sure some reviewers will disagree. I will say that this API is a hard requirement, it cannot be omitted without severely compromising the usefulness of low level file i/o. It gets used to retrieve a path to a temporary inode when you wish to give another program access to that temporary inode; it gets used in a number of algorithms which avoid races on the filesystem; it gets used when printing debug and logging information; the shared filing system mutexes rely upon it extensively. It lets you examine what Unicode normalisation has been performed by the filing system. It enables the LLFIO handle classes to be move relocatable and thus extremely lightweight, because no strings need to be stored. It is a built-in canonicalisation of paths; it is very useful for indexing a container of open handles and detecting equivalence.

An interesting side effect of this function is caused by the lack of one-one mapping of paths on Microsoft Windows, as the path returned will never be in DOS format, as the Windows NT kernel does not use DOS format paths. Instead the path returned will be in NT kernel format (i.e. a `\!\!\` path, see the description of path view interpretation semantics earlier), which does not have a one-one mapping to DOS format paths.

In case people are wondering how to pass what is returned from `.current_path()` to Win32 functions requiring a Win32 format path, LLFIO provides an (expensive) utility function which attempts to map a NT kernel path onto some DOS format path, but it is unavoidably imprecise – it can only return a list of potential DOS format paths for the NT kernel path, and it is on the utility function caller to choose whichever one it thinks is most appropriate.

### 9.6.2 Retrieving the maximum scatter-gather buffer quantity

```
1  /*! \brief The *maximum* number of buffers which a single read or write syscall can process at a
2  time for this specific open handle. On POSIX, this is known as 'IOV_MAX'.
```

```

3
4 Note that the actual number of buffers accepted for a read or a write may be significantly
5 lower than this system-defined limit, depending on available resources. The 'read()' or 'write()'
6 call will return the buffers accepted.
7
8 Note also that some OSs will error out if you supply more than this limit to 'read()' or 'write()',
9 but other OSs do not. Some OSs guarantee that each i/o syscall has effects atomically visible or not
10 to other i/o, other OSs do not.
11
12 OS X does not implement scatter-gather file i/o syscalls.
13 Thus this function will always return '1' in that situation.
14
15 Microsoft Windows may implement scatter-gather file i/o under very limited circumstances.
16 Most of the time this function will return '1'.
17 */
18 size_t max_buffers() const noexcept;

```

This unavoidably runtime observer lets programs query what the maximum number of scatter-gather buffers are for this open file handle. It won't change after the file has been opened. This value can have particular importance on systems which implement POSIX read-write atomicity guarantees, as this would be the maximum limit on gather buffers which can be atomically applied to a file in a single, non-read-tearing, write operation.

### 9.6.3 Retrieving a handle to the parent directory of an open handle

```

1 /*! Obtain a handle to the path currently containing this handle's file entry.
2
3 \warning This call is \b racy and can result in the wrong path handle being returned. Note that
4 unless 'flag::disable_safety_unlinks' is set, this implementation opens a
5 'path_handle' to the source containing directory, then checks if the file entry within has the
6 same inode as the open file handle. It will retry this matching until
7 success until the deadline given.
8
9 \mallocs Calls 'current_path()' and thus is both expensive and calls malloc many times.
10
11 \sa 'algorithm::cached_parent_handle_adapter<T>' which overrides this with a zero cost
12 implementation, thus making unlinking and relinking very considerably quicker.
13 */
14 path_handle parent_path_handle(deadline d = std::chrono::seconds(30)) const throws(file_io_error);

```

A primitive operation used in many other places is to obtain a path handle to the *current* parent directory for an open handle. This is a fundamentally racy call, as whilst the path handle returned is correct at the instantaneously moment of return, a microsecond later somebody else on the filing system may have concurrently renamed the current open handle, and thus the path handle returned is no longer to the parent inode.

Setting aside that, this is not an operation which is deterministic on any of the platforms. The current filesystem path is fetched, its parent path is opened, and a leafname corresponding to the unique identifier of the current open handle is checked for. This is looped, until success.



For inodes with exclusively binary identifiers, the path handle returned is to their owning volume or device.

#### 9.6.4 Retrieving the unique identifier of an open handle

```
1  //!< The device id of the file when opened
2  dev_t st_dev() const noexcept;
3
4  //!< The inode of the file when opened. When
5  //!< combined with st_dev(), forms a unique identifier on this system
6  ino_t st_ino() const noexcept;
7
8  //!< A unique identifier for this handle across the entire system. Can be used in hash tables etc.
9  unique_id_type unique_id() const noexcept;
```

Every file handle is guaranteed to have a valid unique id, and thus can be placed into maps etc. Other types of handle may not have a unique id.

These values are cached upon first retrieval, so the first call to any one of them will involve a syscall.

#### 9.6.5 Retrieving the current maximum extent of an open handle

```
1  /*! Return the current maximum permitted extent of the file.
2
3  \errors Any of the values POSIX fstat() or GetFileInformationByHandleEx() can return.
4  */
5  extent_type maximum_extent() const throws(file_io_error);
```

Files are stored as a list of valid extents, and the file offset of the end of the last valid extent is retrievable via this observer. Note that the value returned is unavoidably racy, it can be incorrect immediately after this function returns due to concurrent file modification.

Many people like to think of files as having a length. The maximum extent value is typically reported by the command line as the file length for legacy compatibility. It is useful because the maximum extent defines the maximum amount of memory necessary to completely read in a file's contents, assuming that file is not being concurrently modified.

#### 9.6.6 Retrieving the current valid extents of an open handle

```
1  /*! \brief Returns a list of currently valid extents for this open file. WARNING: racy!
2  \return A vector of pairs of extent offset + extent length representing the valid extents
3  in this file. Filing systems which do not support extents return a single extent matching
4  the length of the file rather than returning an error.
5  */
6  std::vector<std::pair<extent_type, extent_type>> extents() const throws(file_io_error);
```

This is one of the few functions in LLFIO to dynamically allocate memory, as the call is very expensive, and there is no way of asking how long the list returned will be. The implementation

returns corrupted data on POSIX if any concurrent party modifies the file, this is unavoidable due to a very poor choice of API design by POSIX. LLFIO regularises the list returned so it is at least self-consistent (i.e. extent ends are always after extent starts), but there is little more one can do.

Equally, if one is working with files containing only a few valid extents but a very large maximum extent, then by working in terms of extents one changes a Terabyte-complexity algorithm into a Kilobyte-complexity algorithm. Readers may not consider Terabyte maximum extent files a common use case, however this would change if LLFIO is chosen by WG21. This is because one can very efficiently implement on-disk open addressed hash tables by simply mapping a Terabyte maximum extent file into memory, and reading and writing entries ‘normally’. Due to the sparse underlying storage, only the extents modified are stored on storage, so a file which appears to be a Terabyte long may only consist of a few Megabytes of actual storage.

If you imagine needing to copy such a sparse file from A to B, copying the entire Terabyte is extremely inefficient. Instead, enumerate the extents using this observer, and copy only valid extents.

### 9.6.7 Miscellaneous observers

```
1  /// True if the handle is valid (and usually open)
2  bool is_valid() const noexcept;
3
4  /// True if the handle is readable
5  bool is_readable() const noexcept;
6  /// True if the handle is writable
7  bool is_writable() const noexcept;
8  /// True if the handle is append only
9  bool is_append_only() const noexcept;
10
11 /// True if overlapped
12 bool is_overlapped() const noexcept;
13 /// True if seekable
14 bool is_seekable() const noexcept;
15 /// True if requires aligned i/o
16 bool requires_aligned_io() const noexcept;
17
18 /// True if a regular file or device
19 bool is_regular() const noexcept;
20 /// True if a directory
21 bool is_directory() const noexcept;
22 /// True if a symlink
23 bool is_symlink() const noexcept;
24 /// True if a multiplexer like BSD kqueues, Linux epoll or Windows IOCP
25 bool is_multiplexer() const noexcept;
26 /// True if a process
27 bool is_process() const noexcept;
28 /// True if a memory section
29 bool is_section() const noexcept;
30
31 /// Kernel cache strategy used by this handle
32 caching kernel_caching() const noexcept;
33 /// True if the handle uses the kernel page cache for reads
34 bool are_reads_from_cache() const noexcept;
```

```

35  //!< True if writes are safely on storage on completion
36  bool are_writes_durable() const noexcept;
37  //!< True if issuing safety fsyncs is on
38  bool are_safety_barriers_issued() const noexcept;
39
40  //!< The flags this handle was opened with
41  flag flags() const noexcept;
42  //!< The native handle used by this handle
43  native_handle_type native_handle() const noexcept;

```

The above probably don't need much explanation, other than the choice of the `is_*` and `are_*` naming. There is some overlap between them, but the overlap is orthogonal, and it makes for much more readable code which switches logic based on characteristics about the input file handle.

## 9.7 Operations

### 9.7.1 Relinking

```

1  /*! Relinks the current path of this open handle to the new path specified. If 'atomic_replace' is
2  true, the relink \b atomically and silently replaces any item at the new path specified. This
3  operation is both atomic and silent matching POSIX behaviour even on Microsoft Windows where
4  no Win32 API can match POSIX semantics.
5
6  \warning Some operating systems provide a race free syscall for renaming an open handle (Windows).
7  On all other operating systems this call is \b racy and can result in the wrong file entry being
8  relinked. Note that unless 'flag::disable_safety_unlinks' is set, this implementation opens a
9  'path_handle' to the source containing directory first, then checks before relinking that the item
10 about to be relinked has the same inode as the open file handle. It will retry this matching until
11 success until the deadline given. This should prevent most unmalicious accidental loss of data.
12
13 \param base Base for any relative path.
14 \param path The relative or absolute new path to relink to.
15 \param atomic_replace Atomically replace the destination if a file entry already is present there.
16 Choosing false for this will fail if a file entry is already present at the destination, and may
17 not be an atomic operation on some platforms (i.e. both the old and new names may be linked to the
18 same inode for a very short period of time). Windows and recent Linuxes are always atomic.
19 \param d The deadline by which the matching of the containing directory to the open handle's inode
20 must succeed, else 'errc::timed_out' will be returned.
21 \mallocs Except on platforms with race free syscalls for renaming open handles (Windows), calls
22 'current_path()' via 'parent_path_handle()' and thus is both expensive and calls malloc many times.
23 */
24 void relink(const path_handle &base,
25             path_view_type path,
26             bool atomic_replace = true,
27             deadline d = std::chrono::seconds(30)) throws(file_io_error);

```

The ability to rename whatever is the current path at this moment in time of an open file is useful, hence this operation. It is called relinking rather than renaming because there is an implied atomic new-hard-link-plus-unlink operation involved, with the emphasis on the atomic part. The difference between relinking and linking + unlinking is that relinking will silently atomically replace its destination, if the parameter is true. This atomic replacement is very useful for a multitude of

filesystem algorithms e.g. write out a new version of a file into a temporary file, then atomically swap the finished file over and replacing the previous version of a file.

### 9.7.2 Linking

```
1  /*! Links the inode referred to by this open handle to the path specified. The current path
2  of this open handle is not changed, unless it has no current path due to being unlinked.
3
4  \warning Some operating systems provide a race free syscall for linking an open handle to a new
5  location (Linux, Windows).
6  On all other operating systems this call is \b racy and can result in the wrong inode being
7  linked. Note that unless 'flag::disable_safety_unlinks' is set, this implementation opens a
8  'path_handle' to the source containing directory first, then checks before linking that the item
9  about to be hard linked has the same inode as the open file handle. It will retry this matching
10 until success until the deadline given. This should prevent most unmalicious accidental loss of
11 data.
12
13 \param base Base for any relative path.
14 \param path The relative or absolute new path to hard link to.
15 \param d The deadline by which the matching of the containing directory to the open handle's inode
16 must succeed, else 'errc::timed_out' will be returned.
17 \mallocs Except on platforms with race free syscalls for renaming open handles (Windows), calls
18 'current_path()' via 'parent_path_handle()' and thus is both expensive and calls malloc many times.
19 */
20 void link(const path_handle &base,
21           path_view_type path,
22           deadline d = std::chrono::seconds(30)) throws(file_io_error);
```

Hard linking the inode of an open handle to a new location is very useful. Lots of filesystem algorithms use hard linking, the classic one is to implement a content-addressable index of an existing directory hierarchy by calculating the hash of the content of each file, and hard linking each file to something like `by-content/<HexSHA>`. If one always modifies files within the directory hierarchy using copy-on-write semantics (open existing file, create anonymous file, copy old content to new using valid extents making any changes, atomically rename anonymous file over existing file), one cheaply implements history and file version tracking for some directory hierarchy.

The only annoyance is the lack of support for hard links on some common filing systems such as FAT, but that is outside the scope of LLFIO or the standard C++ library.

### 9.7.3 Inode cloning

```
1  /*! Clones the inode referenced by the open handle into a new inode referencing the same extents
2  for the file content, with a copy of the same metadata, apart from ownership which is for the
3  current user. Changes to either inode are guaranteed to not be seen by the other inode i.e. they
4  become completely independent, even though they initially reference the same file content extents.
5  If your filing system supports this call (at the time of writing - Linux: XFS, btrfs, ocfs2, smbfs;
6  Mac OS: APFS; Windows: ReFS, CIFS), this is a very cheap way of copying even very large files.
7  Be aware that on Samba/CIFS, rather than erroring out if the storage filesystem doesn't implement
8  support, this call is implemented by having the remove machine perform the file
9  copy, which is usually much faster than doing the copy over the network.
10
```

```

11  \return A file handle instance referring to the cloned inode.
12  \param base Base for any relative path.
13  \param path The relative or absolute new path to clone the inode to.
14  */
15  file_handle clone_inode(const path_handle &base,
16                          path_view_type path) throws(file_io_error);

```

All the major platforms implement cheap copies of files which are nearly instantaneous, irrespective of the size of the file. Mainstream filing system support is rather worse, unfortunately, but at least this call works very nicely over a Samba networked drive.

If you do have support, this copy-on-write facility is extremely useful in a wide range of filesystem algorithms. Think of versioned file history, where only the change deltas over the previous version consume storage space, for example.

### 9.7.4 Unlinking

```

1  /*! Unlinks the current path of this open handle, causing its entry to immediately disappear
2  from the filing system. On Windows before Windows 10 1709 unless
3  'flag::win_disable_unlink_emulation' is set, this behaviour is simulated by renaming the file
4  to something random and setting its delete-on-last-close flag. Note that Windows may prevent
5  the renaming of a file in use by another process, if so it will NOT be renamed.
6  After the next handle to that file closes, it will become permanently unopenable by anyone
7  else until the last handle is closed, whereupon the entry will be eventually removed by the
8  operating system.
9
10 \warning Some operating systems provide a race free syscall for unlinking an open handle (Windows).
11 On all other operating systems this call is \b racy and can result in the wrong file entry being
12 unlinked. Note that unless 'flag::disable_safety_unlinks' is set, this implementation opens a
13 'path_handle' to the containing directory first, then checks that the item about to be unlinked
14 has the same inode as the open file handle. It will retry this matching until success until the
15 deadline given. This should prevent most unmalicious accidental loss of data.
16
17 \param d The deadline by which the matching of the containing directory to the open handle's inode
18 must succeed, else 'errc::timed_out' will be returned.
19 \mallocs Except on platforms with race free syscalls for unlinking open handles (Windows), calls
20 'current_path()' and thus is both expensive and calls malloc many times. On Windows, also calls
21 'current_path()' if 'flag::disable_safety_unlinks' is not set.
22 */
23 void unlink(deadline d = std::chrono::seconds(30)) throws(file_io_error);

```

The ability to unlink whatever is the current path at this moment in time of an open file is useful. You don't need to concern yourself that the path has changed due to concurrent filesystem change. You just unlink.

Note that I have submitted requests for new syscalls to directly unlink an open file descriptor to Linux and FreeBSD. The Linux kernel devs have indicated that they would have no concern implementing that if LLFIO begins standardisation.

### 9.7.5 Truncating extents and setting a new maximum extent

```

1  /*! Resize the current maximum permitted extent of the file to the given extent, avoiding any
2  new allocation of physical storage where supported. Note that on extents based filing systems
3  this will succeed even if there is insufficient free space on the storage medium.
4
5  \return The bytes actually truncated to.
6  \param newsize The bytes to truncate the file to.
7  \errors Any of the values POSIX ftruncate() or SetFileInformationByHandle() can return.
8  */
9  extent_type truncate(extent_type newsize) throws(file_io_error);

```

Extents-based filesystem storage is essentially a kernel-implemented dynamic memory allocator working in page size chunks. If you set the maximum extent of an empty file to 64Tb and map the entire file into memory, upon first write anywhere within the 64Tb space the system will allocate storage for the pages modified only.

If you then wish to throw away all content in the allocated extents after say 32Tb, one would *truncate* those extents by setting the new maximum extent for the file to 32Tb. This is why this function is called `.truncate()`.

An alternative name for this function could be `.maximum_extent(extent_type)` i.e. set a new maximum extent. This name was strongly considered as it matches the maximum extent observer, however because calling this function implies potential data loss, it was decided that it would be safer to name it `truncate`, so people ponder the potential data loss considerations when they call it.

This does however lead to the non-intuitive situation that to *extend* the file, you need to *truncate* it, which many feel makes no sense.

The usual rejoinder is why can't there be a `.maximum_extent(extent_type)` which is always safe i.e. never throws away data, and reserve `.truncate()` for data-loss use cases? That would be great, in fact, ideal, however there is no syscall support for doing this in a race free way. To be specific, one would have to read the maximum extent and then perhaps write a new maximum extent using **two** separate syscalls. This introduces a race: a concurrent program may write considerable data to the file in the moment between reading the maximum extent and deciding to set a new maximum extent in the current program, thus inadvertently destroying the data written by the concurrent program.

Balancing all of these issues together, the name of `.truncate()` was settled upon as the least worst naming choice given the constraints at hand. It is also hoped that if LLFIO began standardisation, the OS kernel maintainers could be persuaded into adding a new syscall which only sets a new maximum extent if no extents would be destroyed. If that were possible, then one would wish to reserve the use of `.maximum_extent(extent_type)` for that use case.

## 9.7.6 Deleting specific allocated extents

```

1  /*! \brief Efficiently zero, and possibly deallocate, data on storage.
2
3  On most major operating systems and with recent filing systems which are "extents based", one can
4  deallocate the physical storage of a file, causing the space deallocated to appear all bits zero.
5  This call attempts to deallocate whole pages (usually 4Kb) entirely, and memset's any excess to all

```

```

6 bits zero. This call works on most Linux filing systems with a recent kernel, Microsoft Windows
7 with NTFS, and FreeBSD with ZFS. On other systems it simply writes zeros.
8
9 \return The bytes zeroed.
10 \param offset The offset to start zeroing from.
11 \param bytes The number of bytes to zero.
12 \param d An optional deadline by which the i/o must complete, else it is cancelled.
13 Note function may return significantly after this deadline if the i/o takes long to cancel.
14 \errors Any of the values POSIX write() can return, 'errc::timed_out', 'errc::operation_canceled'.
15 'errc::not_supported' may be returned if deadline i/o is not possible with this particular
16 handle configuration (e.g. writing to regular files on POSIX or writing to a non-overlapped
17 HANDLE on Windows).
18 \mallocs The default synchronous implementation in file_handle performs no memory allocation.
19 The asynchronous implementation in async_file_handle may perform one calloc and one free.
20 */
21 extent_type zero(extent_type offset,
22                 extent_type bytes,
23                 deadline d = deadline()) throws(file_io_error);

```

As mentioned in the last section, extents-based filesystem storage is essentially a dynamic memory allocator working in page size chunks. If one is on a common filing system on a kernel made in the last decade, one can explicitly deallocate specific extents using this function.

Unallocated extents in a file appear as all bits zero, hence the choice of naming. Extent deallocation is usually page size granularity or worse, so if this function is called with non-page size granularity alignment or amount, `memset()` is used to zero any 'spill' bytes. This does mean that the user must be careful about how to call this function if they are not to accidentally leave around allocated extents of all bits zero.

Due to these facilities as a page-sized dynamic memory allocator, a polymorphic STL allocator can be very usefully built using a file as backing storage, and LLFIO provides extensive convenience facilities for doing exactly that e.g. `trivial_vector<T>`. This allows one to allocate temporary Terabyte-scale sparse-stored arrays far exceeding total system memory and swap space, for example.

### 9.7.7 Write reordering barriers

```

1  //!< The kinds of write reordering barrier which can be performed.
2  enum class barrier_kind
3  {
4      nowait_data_only,  //!<< Barrier data only, non-blocking. This is highly optimised on NV-DIMM
5                          //!<< storage, but consider using 'nvram_barrier()' for even better performance.
6      wait_data_only,   //!<< Barrier data only, block until it is done. This is highly optimised on
7                          //!<< NV-DIMM storage, but consider using 'nvram_barrier()' for even better
8                          performance.
9      nowait_all,       //!<< Barrier data and the metadata to retrieve it, non-blocking.
10     wait_all           //!<< Barrier data and the metadata to retrieve it, block until it is done.
11 };
12
13 /*! \brief Issue a write reordering barrier such that writes preceding the barrier will reach
14 storage before writes after this barrier.
15
16 \warning **Assume that this call is a no-op**. It is not reliably implemented in many common

```

```

16 use cases, for example if your code is running inside a LXC container, or if the user has mounted
17 the filing system with non-default options. Instead open the handle with 'caching::reads' which
18 means that all writes form a strict sequential order not completing until acknowledged by the
19 storage device. Filing system can and do use different algorithms to give much better performance
20 with 'caching::reads', some (e.g. ZFS) spectacularly better.
21
22 \warning Let me repeat again: consider this call to be a hint to poke the kernel with a stick
23 to go start to do some work sooner rather than later. It may be ignored entirely.
24
25 \warning For portability, you can only assume that barriers write order for a single handle
26 instance. You cannot assume that barriers write order across multiple handles to the same inode,
27 or across processes.
28
29 \return The buffers barriered, which may not be the buffers input. The size of each scatter-gather
30 buffer is updated with the number of bytes of that buffer barriered.
31 \param reqs A scatter-gather and offset request for what range to barrier. May be ignored on
32 some platforms which always write barrier the entire file. Supplying a default initialised reqs
33 write barriers the entire file.
34 \param kind Which kind of write reordering barrier to perform.
35 \param d An optional deadline by which the i/o must complete, else it is cancelled.
36 Note function may return significantly after this deadline if the i/o takes long to cancel.
37 \errors Any of the values POSIX fdatasync() or Windows NtFlushBuffersFileEx() can return.
38 \mallocs None.
39 */
40 io_result<const_buffers_type> barrier(io_request<const_buffers_type> reqs = io_request<
41     const_buffers_type>(),
42                                     barrier_kind kind = barrier_kind::nowait_data_only,
43                                     deadline d = deadline()) noexcept = 0;

```

This function provides the oft-asked-for-by-stackoverflow `fdatasync()` and `fsync()` to user code so code can implement durability after sudden power loss. The syscalls actually used are often not those, but rather finer grained ones which can operate upon specific extents, and with more performant semantics.

Something deliberately not done is to return the kind of barriering used, which is something reviewers may wish to change. For example, if you ask for `barrier_kind::nowait_data_only`, on some systems you might actually get `barrier_kind::wait_all`.

The reason we do not bother returning what kind was used is that it is undetectable in many cases, and besides it isn't actually important in practice. Firstly, POSIX implementations may legally ignore the syscall entirely, and many do. Secondly, if you are calling this function, you generally want it to actually work irrespective of performance, because you wouldn't be calling it unless sudden power loss safety, or preventing write i/o cascade stalls, is very important to you.

As the documentation above mentions, due to the unpredictable potential for this operation to be a no-op, any code which *really* cares about sudden power loss durability should use `caching::reads` instead of write reordering barriers. That said, this function remains useful for 'implement durability unless the environment has disabled it' use cases, or for preventing backlogs of writes from forming, hence why it has been retained.

This rationale explains the choice of the default for the kind of barrier: data only, non-blocking. This is a hint to the OS kernel that the data written to the region specified will not be modified



again soon, and so it should be written to storage more urgently than other data. This means that the expected default use of `.barrier()` is not to ensure sudden power loss durability, but rather to prevent write backlogging which causes cascading storms of i/o when timers expire. In other words, if you want your i/o latencies after 95% to be regular, you must issue a write barrier when doing cached i/o writes, especially when to random offsets.

### 9.7.8 Extent guards

```
1  /*! \class extent_guard
2  \brief RAI holder a locked extent of bytes in a file.
3  */
4  class extent_guard
5  {
6  public:
7      ~extent_guard();
8      extent_guard(const extent_guard &) = delete;
9      extent_guard &operator=(const extent_guard &) = delete;
10
11     /*! Default constructor
12     constexpr extent_guard();
13
14     /*! Move constructor
15     extent_guard(extent_guard &&o) noexcept;
16
17     /*! Move assign
18     extent_guard &operator=(extent_guard &&o) noexcept;
19
20     /*! True if extent guard is valid
21     explicit operator bool() const noexcept;
22
23     /*! True if extent guard is invalid
24     bool operator!() const noexcept;
25
26     /*! The io_handle to be unlocked
27     io_handle *handle() const noexcept;
28
29     /*! Sets the io_handle to be unlocked
30     void set_handle(io_handle *h) noexcept;
31
32     /*! The extent to be unlocked
33     std::tuple<extent_type, extent_type, bool> extent() const noexcept;
34
35     /*! Unlocks the locked extent immediately
36     void unlock() noexcept;
37
38     /*! Detach this RAII unlocker from the locked state
39     void release() noexcept;
40 };
41
42
43 /*! \brief Tries to lock the range of bytes specified for shared or exclusive access. Be aware
44 this passes through the same semantics as the underlying OS call, including any POSIX insanity
45 present on your platform:
46
```

```

47 - Any fd closed on an inode must release all byte range locks on that inode for all
48 other fds. If your OS isn't new enough to support the non-insane lock API,
49 'flag::byte_lock_insanity' will be set in flags() after the first call to this function.
50 - Threads replace each other's locks, indeed locks replace each other's locks.
51
52 You almost certainly should use your choice of an 'algorithm::shared_fs_mutex::*' instead of this
53 as those are more portable and performant.
54
55 \warning This is a low-level API which you should not use directly in portable code. Another
56 issue is that atomic lock upgrade/downgrade, if your platform implements that (you should assume
57 it does not in portable code), means that on POSIX you need to *release* the old 'extent_guard'
58 after creating a new one over the same byte range, otherwise the old 'extent_guard''s destructor
59 will simply unlock the range entirely. On Windows however upgrade/downgrade locks overlay, so on
60 that platform you must *not* release the old 'extent_guard'. Look into
61 'algorithm::shared_fs_mutex::safe_byte_ranges' for a portable solution.
62
63 \return An extent guard, the destruction of which will call unlock().
64 \param offset The offset to lock. Note that on POSIX the top bit is always cleared before use
65 as POSIX uses signed transport for offsets. If you want an advisory rather than mandatory lock
66 on Windows, one technique is to force top bit set so the region you lock is not the one you will
67 i/o - obviously this reduces maximum file size to (2^63)-1.
68 \param bytes The number of bytes to lock. Zero means lock the entire file using any more
69 efficient alternative algorithm where available on your platform (specifically, on BSD and OS X use
70 flock() for non-insane semantics).
71 \param exclusive Whether the lock is to be exclusive.
72 \param d An optional deadline by which the lock must complete, else it is cancelled.
73 \errors Any of the values POSIX fcntl() can return, 'errc::timed_out', 'errc::not_supported' may be
74 returned if deadline i/o is not possible with this particular handle configuration (e.g.
75 non-overlapped HANDLE on Windows).
76 \mallocs The default synchronous implementation in file_handle performs no memory allocation.
77 The asynchronous implementation in async_file_handle performs one calloc and one free.
78 */
79 extent_guard lock(extent_type offset,
80                 extent_type bytes,
81                 bool exclusive = true,
82                 deadline d = deadline()) throws(file_io_error);
83
84 //! \overload
85 extent_guard try_lock(extent_type offset,
86                    extent_type bytes,
87                    bool exclusive = true) throws(file_io_error);
88
89 //! \overload Locks for shared access
90 extent_guard lock(io_request<buffers_type> reqs,
91                deadline d = deadline()) throws(file_io_error);
92
93 //! \overload Locks for exclusive access
94 extent_guard lock(io_request<const_buffers_type> reqs,
95                deadline d = deadline()) throws(file_io_error);

```

This is a low level API returning an RAII manager for locking byte ranges (or the whole file) for shared or exclusive access within a file. By low level, we really do mean very very low level, nobody should use this API directly without extremely good reason. Higher level abstractions in LLFIO iron out the many, many platform-specific quirks in this facility which will bite you in the ass.

Some of the particularly fun quirks in this API:

- Locks are advisory on POSIX (i.e. i/o ignores the locks), mandatory on Windows (i.e. on Windows they block all i/o on the locked regions everywhere in the system). On POSIX, these are therefore a pure IPC mechanism, only other concurrent file users also using locks see coordination.
- On some POSIX, but not others, closing any file descriptor to an inode causes all byte range locks to that inode to be immediately dropped, even if other file descriptors to the same inode are open. This is the source of much surprise and bugs, but POSIX mandates this behaviour.
- Locks stack upon one another on Windows, but locks replace one another with potentially non-atomic upgrade/downgrade on POSIX.
- Offset and extent are signed on POSIX due to an awful API choice, so you cannot place locks on the top half of a file, at all.

So why expose this API at all?

There is more than one higher level abstraction for byte range locks in LLFIO, each with a different set of tradeoffs. No one abstraction is ideal, and there will be locking abstractions which we won't think of, and users will need to roll their own. Hence exposing this API so users can roll their own.

The overloads taking a scatter or gather buffer list are mainly for convenience: if you're about to write some data, feeding the same list to `.lock()` will take an exclusive lock on that region about to get written. Similarly, for reading, a shared lock is taken. There is a good argument that these convenience overloads should be omitted if nobody ought to be using this API directly, but I'd like to get a read of committee opinion first. The higher level abstractions are more fussy to use, more cognitive overhead is involved. For simple 'fire and forget and it'll probably work' byte range locking, this very low level API has much in favour of it, especially on Microsoft Windows where it works without surprise.

## 9.8 Scatter reading from a file offset

Most of you who have made it this far probably thought that a file handle would merely be openable, readable, writable, and maybe closeable, and you did not expect the other thirty pages of stuff preceding now. I can empathise – I did not expect it to take me so many weeks to write out an API rationale for a simple file handle either.

Still, we are finally on to the last two operations in `file_handle`, reading and writing!

```
1  /*! \brief Read data from the open handle.
2
3  \warning Depending on the implementation backend, very different buffers may be returned than
4  you supplied. You should always use the buffers returned and assume that they point to different
5  memory and that each buffer's size will have changed.
6
7  \return The buffers read, which may not be the buffers input. The size of each scatter-gather
8  buffer returned is updated with the number of bytes of that buffer transferred, and the pointer
9  to the data may be \em completely different to what was submitted (e.g. it may point into a
10 memory map).
```

```

11  \param reqs A scatter-gather and offset request.
12  \param d An optional deadline by which the i/o must complete, else it is cancelled.
13  Note function may return significantly after this deadline if the i/o takes long to cancel.
14  \errors Any of the values POSIX read() can return, 'errc::timed_out', 'errc::operation_canceled'.
15  'errc::not_supported' may be returned if deadline i/o is not possible with this particular handle
16  configuration (e.g. reading from regular files on POSIX or reading from a non-overlapped HANDLE
17  on Windows).
18  \mallocs The default synchronous implementation in file_handle performs no memory allocation.
19  The asynchronous implementation in async_file_handle performs one calloc and one free.
20  */
21  io_result<buffers_type> read(io_request<buffers_type> reqs,
22                             deadline d = deadline()) throws(file_io_error);
23
24
25  /// Convenience initialiser list based overload for 'read()'
26  size_type read(extent_type offset,
27                std::initializer_list<buffer_type> lst,
28                deadline d = deadline()) throws(file_io_error);

```

As explained earlier, every handle type in LLFIO defines its single buffer type, its plural buffers type, its i/o request type, and its i/o result type. For `file_handle`, that is an `iovec_buffer` which quacks like `span<byte>`; plural buffers is a `span<iovec_buffer>`; i/o request is an input plural buffers type and an offset representing the buffers to be filled; and the i/o result type is an output plural buffers type giving the buffers that were filled.

There is very little to the implementation for `file_handle` – on POSIX, it simply calls `preadv()`, and truncates the buffer list supplied at the point of fill, if necessary.

We deliberately do not support any notion of there being a ‘current’ file offset pointer from and to which i/o occurs. Such things are fundamentally racy in a multithreaded use case – furthermore, one can locally implement a current file offset pointer very easily if one needs that.

As mentioned earlier, the buffers input are modified, which can be irritating to use, so file handle adds a convenience overload which accepts an initialiser list for the scatter buffer list, and which returns the total number of bytes read. This isn’t as efficient, it creates a temporary scatter list either on the stack or in dynamically allocated memory, copies over the list, calls the preceding call, sums the lengths of the buffers returned, throws away the temporary buffers. You should therefore only use this overload in once-off use cases where you don’t need to read buffers filled in case of error, and not inside a loop.

If your kernel and filing system implement the POSIX acquire-release i/o guarantees, up to `.max_buffers()` scatter buffers will be read as a single, atomic operation invariant to concurrent modification i.e. your read buffer will not see torn writes. Subsequent reads and writes to extents overlapping this read operation will not appear to concurrent users to be reordered before this read (for the overlapping bytes only). This is an i/o visibility ordering guarantee to concurrent users only – no ordering is imposed upon reads from storage unless the caching setting is set to require that.

POSIX acquire-release i/o guarantees are widely supported on the major platforms and filing systems, with only ext4 on Linux not implementing them for cached i/o.

## 9.9 Gather writing to a file offset

```
1  /*! \brief Write data to the open handle.
2
3  \warning Depending on the implementation backend, not all of the buffers input may be written.
4  For example, with a zeroed deadline, some backends may only consume as many buffers as the system
5  has available write slots for, thus for those backends this call is "non-blocking" in the sense
6  that it will return immediately even if it could not schedule a single buffer write. Another example
7  is that some implementations will not auto-extend the length of a file when a write exceeds the
8  maximum extent, you will need to issue a 'truncate(newsize)' first.
9
10 \return The buffers written, which may not be the buffers input. The size of each scatter-gather
11 buffer returned is updated with the number of bytes of that buffer transferred.
12 \param reqs A scatter-gather and offset request.
13 \param d An optional deadline by which the i/o must complete, else it is cancelled.
14 Note function may return significantly after this deadline if the i/o takes long to cancel.
15 \errors Any of the values POSIX write() can return, 'errc::timed_out', 'errc::operation_canceled'.
16 'errc::not_supported' may be returned if deadline i/o is not possible with this particular handle
17 configuration (e.g. writing to regular files on POSIX or writing to a non-overlapped HANDLE on
18 Windows).
19 \mallocs The default synchronous implementation in file_handle performs no memory allocation.
20 The asynchronous implementation in async_file_handle performs one calloc and one free.
21 */
22 io_result<const_buffers_type> write(io_request<const_buffers_type> reqs,
23                                   deadline d = deadline()) throws(file_io_error);
24
25
26 ///! \overload
27 size_type write(extent_type offset,
28                std::initializer_list<const_buffer_type> lst,
29                deadline d = deadline()) throws(file_io_error);
```

There is very little to mention here after explaining `.read()`. The array of byte arrays is written to the offset specified, or to the maximum extent if the handle is in atomic append mode. There is actually a fairly long list of platform-specific quirks which is described in P1031, but for most uses, most of the time, this does the job exactly as one would intuitively expect. The convenience overload is similar to that for `.read()`.

If your kernel and filing system implement the POSIX acquire-release i/o guarantees, up to `.max_buffers()` gather buffers will be written as a single, atomic operation invariant to concurrent reading i.e. concurrent reads will not see a torn write. Preceding reads and writes to extents overlapping this write operation will not appear to concurrent users to be reordered after this write (for the overlapping bytes only). This is an i/o visibility ordering guarantee to concurrent users only – no ordering is imposed upon writes to storage unless the caching setting is set to require that.

## 10 Conclusion

All of the above will probably seem like massive overkill for modern file i/o in the C++ standard. Other modern i/o proposals for C++ 23 will appear to be much simpler, and easier to wrap the

mind around.

There is no doubt that the learning curve is steep for the uninitiated. However, the learning curve is probably even steeper for the Networking TS, and WG21 chose the low-level i/o approach for that.

Furthermore, unlike with the Networking TS, if this approach is chosen, it is currently expected that most users won't actually use these APIs directly, because a higher level i/o abstraction (such as a putative Ranges i/o) would be the default initial choice for most users in the same way as people will tend to choose iostreams over the C `FILE` API. One would only drop to low level file i/o when one needs the added control and detail necessary for achieving specific semantics, or performance.

There is no doubt that this approach to file i/o scales to hardware extremely well in a way that the other modern i/o approaches I know of would struggle with. LLFIO reference library users have deployed some very high performance storage solutions – one commercial product based on LLFIO that I can name drop is <http://3misses.com/>, which is orders of magnitude faster than other commercial alternatives at 95% of the latency distribution when run on NV-DIMM storage.

Speaking personally, I've rapidly deployed solutions providing millisecond-level access latencies with over a Petabyte of data before compression, for which you need to keep multiple NVMe SSDs bound in RAID0 fully occupied on a sustainable basis. To give you an idea of the difficulty here, one is operating at a large fraction of main memory bandwidth, so avoiding any memory copy anywhere in the system is paramount, plus one must be extremely careful with kernel cache management to prevent bottlenecks and backlogs forming which could induce i/o stalls. Moreover, these solutions are concurrency-invariant, and thus many independent CPU cores and processes can be deployed safely upon the same directory hierarchy.

Basically, LLFIO *works* for a wide solution space in file i/o, in the same way as ASIO works for a wide solution space in socket i/o. The committee, ultimately, needs to decide on whether it prefers a narrow or wide solution for file i/o.

I am not claiming that building high performance storage solutions with LLFIO is any easier than building an ultra-low-latency trading application using the Networking TS. I **am** claiming that the low level file i/o approach is the only one of the modern i/o proposals that I am aware of which can deliver ~25Gb/sec of uncached i/o bandwidth on current hardware, with strong concurrency and durability guarantees, and with good control of caching.

## 11 References

- [P0709] Herb Sutter,  
*Zero-overhead deterministic exceptions: Throwing values*  
<https://wg21.link/P0709>
- [P0829] Ben Craig,  
*Freestanding proposal*  
<https://wg21.link/P0829>

- [P1028] Douglas, Niall  
*SG14 status\_code and standard error object for P0709 Zero-overhead deterministic exceptions*  
<https://wg21.link/P1028>
- [P1029] Douglas, Niall  
*SG14 move = relocates*  
<https://wg21.link/P1029>
- [P1030] Douglas, Niall  
*std::filesystem::path\_view*  
<https://wg21.link/P1030>
- [P1031] Douglas, Niall  
*Low level file i/o*  
<https://wg21.link/P1031>
- [POSIXext] *The Open Group Technical Standard, 2006, Extended API Set Part 2*  
<https://pubs.opengroup.org/onlinepubs/9699939699/toc.pdf>