P1883R2: `file_handle` and `mapped_file_handle`

Document #:	P1883R2
Date:	2022-11-25
Project:	WG21 Programming Language C++
	Library Evolution Working Group Incubator
	SG1 Concurrency
Reply-to:	Niall Douglas
	<s_sourceforge@nedprod.com>

Two kinds of low level file handle, propagating all the host platform's i/o determinism and progress guarantees, suitable for building high level abstractions on top. They are suitable for inclusion in Freestanding, avoiding dynamic memory allocation in all but one API which wouldn't be useful on Freestanding anyway, and not throwing C++ exceptions.

A reference implementation of the proposed classes can be found at `https://github.com/ned14/llfio/`. They have been found to work well on recent editions of GCC, clang and Microsoft Visual Studio, in x86, x64, ARM and AArch64, on Apple Mac OS, FreeBSD, Linux and Microsoft Windows. They have been in production use for several years now.

Main changes since R1:

- Added examples of use as per WG21 request.
- Removed coroutine APIs, now that there is no obvious path between this and async i/o in the medium term future.
- As per WG21 guidance, removed the `io_request` based APIs (I still think this a missed ABI stabilisation opportunity personally).
- As per WG21 guidance, removed directly mapped storage support.
- In the past two years, it was found useful if the regex string used by `path_discovery::storage_backed_temporary_directory()` were exposed as a constexpr variable.
- `clone_extents_to()` has been implemented by the reference library since R1, and thus is now specified.
- In the past two years, it was found useful to support filesystem extended attributes, so APIs for those have been added.
- Added to mapped file handle the ability to specify the starting offset into the file contents for the map.
- As per WG21 guidance, mapped file handle's address retrieving function now returns a volatile byte pointer. This is probably unduly pessimistic, however it can be cast off if the end user knows the file's contents will not be concurrently modified by others.

Contents

1 Introduction

The Prague 2020 meeting found continuing strong support for standardising these classes, with a fairly small set of changes requested by LEWG-I. It was recommended that SG1 look at proposed `mapped_file_handle`, as they have only reviewed `file_handle` so far. SG1 may also wish to re-discuss my decision to not implement kernel thread aware whole file locks withing these classes – these whole file locks are kernel thread aware only if file locks are kernel thread aware on the host OS.

As with all my in-flight WG21 papers, there has been a two year gap since the last revision of this paper, and a great deal has changed in the C++ world, not least that the i/o train that we thought we were on has dramatically changed. As I describe in [P2586] *Standard Secure Networking*, in my opinion the current formulation of [P2300] *`std::execution`* is not suitable for high performance i/o. Therefore, now is not the time to propose coroutine editions of the APIs either in proposed `std::file_handle` any more than in P2586's `std::byte_socket_handle`. I therefore have dropped those APIs from R2.

Something perhaps not obvious in this proposal is that the `file_handle` i/o API is absolutely identical to P2586's socket handles – in the latter, the (optional) offset is ignored, but the API and more importantly, the ABI, is identical. The reference library, and R1 of this paper, proposed an i/o API based around an `io_request` object which carried the i/o parameters in an ABI stable fashion. It was also thought that maybe it could carry 'flavour' of the most derived class, which might be useful to implementations. However, during intensive quizzing at the Prague meeting, I couldn't really come up with concrete reasons to standardise it, so the feedback was to drop it entirely. I still don't agree with this decision – I may not be able to give concrete reasons why to keep it, but I feel in my gut it's the right API design. Still, I defer to the committee's judgement, and the `io_request` based API and ABI is gone from R2.

The last big change is to remove API support for directly mapped storage. Yes I know directly mapped storage launches onto the consumer market in 2023 (there are already some devices on the consumer market claiming support, however Microsoft's DirectStorage hasn't shipped yet), so culling that support now does seem unwise. However we can always add back in the relevant flags and APIs in a later paper, and the reference library still contains the support. Let's bite off a smaller slice first.

Finally, the reference implementation of both these classes continue to chew through Terabytes of data per day capturing all trades on various markets. They are extremely well tested by this point, and during the past two years I think only one major bug was found, and that was due to me foolishly editing code late on a Friday evening in a rush which is never a wise thing to do. In any case, no changes occurred to the API proposed here two years ago.

It is only due to my current client MayStreet London Stock Exchange Group allowing me a few work hours per month to work on standards papers that you see this R2 now and not yet more months from now. I hope that normal service will resume in 2023. I currently expect to be at the Issaquah meeting in early 2023.

2 Design principles

1. The default parameters or configuration choose security over performance, but one can always explicitly opt out on a case by case basis.
2. 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).
3. Pass through as much of the host OS concurrent i/o concurrency guarantees as possible.
4. Designed around the POSIX race free filesystem path lookup extensions [POSIXext].
5. It should always be possible to avoid all memory copies between the C++ i/o and the final storage device, anywhere in the system.
6. Expose kernel i/o cache control and virtual memory control facilities.

7. 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.

3 Examples of use

In the Prague meeting LEWG-I asked for examples of use to be in R2. Note that this is a `result` based API (see later) so the `.value()` means 'if this fails throw the cause of failure as a C++ exception':

3.1 Scatter write

```
// Works with either kind of file handle
void write_and_close(file_handle &&fh)
{
  // Set the maximum extent of the file to what is about to be written,
  // needed as mapped file handles do not auto extend their file if one
  // writes past the end of them.
  fh.truncate(12).value();

  // Buffers to write in a single shot
  const char a[] = "hel";
  const char b[] = "l";
  const char c[] = "lo w";
  const char d[] = "orld";

  // Perform gather write. For file_handle if max_buffers() >= 4,
  // this write will be atomic with respect to concurrent file reads
  // either appearing complete or not at all i.e. no torn reads.
  // For mapped file handle there can be no synchronisation and
  // reads and writes always tear with respect to concurrent
  // readers and writers, so additional synchronisation would be
  // needed e.g. fh.lock_file().
  fh.write(
    { // gather list, buffers use std::byte so we must cast from char
      { reinterpret_cast<const byte*>(a), sizeof(a) - 1 },
      { reinterpret_cast<const byte*>(b), sizeof(b) - 1 },
      { reinterpret_cast<const byte*>(c), sizeof(c) - 1 },
      { reinterpret_cast<const byte*>(d), sizeof(d) - 1 },
    }
                              // default deadline is infinite
  ).value();                  // If failed, throw a filesystem_error exception

  // Explicitly close the file rather than letting the destructor do it,
  // in case file close fails.
  fh.close().value();
}

// Open the file for write, creating if needed, write through caching
write_and_close(file(
  {},                                         // path_handle to base directory from which to lookup
```

```
                                            // leafname, in this case means current directory
  "hello",                                  // path_view to path fragment relative to base directory
  file_handle::mode::write,                 // write access please
  file_handle::creation::if_needed,         // create new file if needed
  file_handle::caching::reads_and_metadata  // writes block until they reach storage
                                            // default flags is none
).value());                                 // If failed, throw a filesystem_error exception

// Modify an existing file using memory maps
const path_handle& somewhere;
write_and_close(mapped_file(
  somewhere,                                // base directory from which to lookup leafname
  "hello2",                                 // path_view to path fragment relative to base directory
  file_handle::mode::write,                 // write access please
  file_handle::creation::open_existing      // only open an existing file, fail if doesn't exist
                                            // default caching is all
                                            // default flags is none
).value());                                 // If failed, throw a filesystem_error exception
```

3.2 Sparse stored array not consuming process private memory

```
// Make me a 1 trillion element sparsely allocated integer array!
// As a temporary inode, it does not appear on the filing system
// and no other process has access to it. It does not count towards
// the process' memory consumption, and therefore filling it will never
// create an out of memory situation. It does consume filing
// system storage space, and one can run out of that.
mapped_file_handle mfh = mapped_temp_inode().value();

// On an extents based filing system, doesn't actually allocate any physical
// storage but does map approximately 4Tb of all bits zero data into memory
mfh.truncate(1000000000000ULL * sizeof(int)).value();

// Create a typed view of the one trillion integers
span<int> one_trillion_int_array(reinterpret_cast<int*>(mfh.address()),
                                 1000000000000ULL);

// Write and read as you see fit, if you exceed physical RAM it'll be paged out
// Only the extents modified consume storage on the filing system, so this is
// quite efficient.
one_trillion_int_array[0] = 5;
one_trillion_int_array[999999999999ULL] = 6;

// Throw away the storage, all resources are released.
mfh.close().value();
```

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()`.

```
//! The behaviour of the handle: does it read, read and write, or atomic append?
enum class mode : unsigned char  // bit 0 set means writable
{
  unchanged = 0,
  none = 2,         //!< No ability to read or write anything, but can synchronise (SYNCHRONIZE or 0)

  attr_read = 4,    //!< Ability to read attributes (FILE_READ_ATTRIBUTES|SYNCHRONIZE or O_RDONLY)

  attr_write = 5,   //!< Ability to read and write attributes (FILE_READ_ATTRIBUTES|
                    //!< FILE_WRITE_ATTRIBUTES|SYNCHRONIZE or O_RDONLY)

  read = 6,         //!< Ability to read (READ_CONTROL|FILE_READ_DATA|FILE_READ_ATTRIBUTES|
                    //!< FILE_READ_EA|SYNCHRONISE or O_RDONLY)

  write = 7,        //!< Ability to read and write (READ_CONTROL|FILE_READ_DATA|FILE_READ_ATTRIBUTES|
                    //!< FILE_READ_EA|FILE_WRITE_DATA|FILE_WRITE_ATTRIBUTES|FILE_WRITE_EA|
                    //!< FILE_APPEND_DATA|SYNCHRONISE or O_RDWR)

  append = 9        //!< All mainstream OSs and CIFS guarantee this is atomic with respect to all
                    //!< other appenders (FILE_APPEND_DATA|SYNCHRONISE or O_APPEND)
};
```

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.

```
//! On opening, do we also create a new file or truncate an existing one?
enum class creation : unsigned char
{
  open_existing = 0,  //!< Filesystem entry must already exist (OPEN_EXISTING or 0)

  only_if_not_exist,  //!< Filesystem entry must NOT exist, and is atomically created by the
                      //!< success of this operation (CREATE_NEW or O_CREAT|O_EXCL)

  if_needed,          //!< If filesystem entry exists that is used, else one is created
                      //!< (OPEN_ALWAYS or O_CREAT)

  truncate_existing,  //!< Filesystem entry must already exist. It is atomically truncated on open,
                      //!< leaving creation date and unique identifier unmodified (TRUNCATE_EXISTING
                      //!< or O_TRUNC)

  always_new          //!< A newly created inode is already created, replacing any existing
                      //!< (CREATE_ALWAYS or <synthesised>)
};
```

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[1]. 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.

[1]The 'e' modifier for `fopen()` does not do what one would think, so I proposed a NB comment to WG14 so that C23 will be fixed going forth. POSIX has already been fixed.

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. Microsoft Windows has direct API support for this flag.

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[2]) 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 on unified cache virtual memory kernels[3].

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 shortly. Other kinds of i/o handles in LLFIO *may* implement userspace caching based on these flags, but neither `file_handle` nor `mapped_file_handle` do so.

```
//! What i/o on the handle may complete immediately due to kernel caching
enum class caching : unsigned char  // bit 0 set means safety fsyncs enabled
{
  unchanged = 0,
  none = 1,                 //!< No caching whatsoever, all reads and writes come from storage
                            //!< (i.e. <tt>O_DIRECT|O_SYNC</tt>). Align all i/o to 4Kb boundaries
                            //!< for this to work. <tt>disable_safety_barriers</tt> can be used
                            //!< here i.e. safety barriers are enabled by default here.

  only_metadata = 2,        //!< Cache reads and writes of metadata but avoid caching data
                            //!< (<tt>O_DIRECT</tt>), thus i/o here does not affect other cached data
                            //!< for other handles. Align all i/o to 4Kb boundaries for this to work.

  reads = 3,                //!< Cache reads only. Writes of data and metadata do not complete until
```

[2]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.

[3]Interestingly, some platforms do actually try to honour non-full kernel caching semantics on memory mapped files. Microsoft Windows, for example, will write out every single dirty page as soon as you dirty it if the backing file was opened with kernel caching of reads only. This obviously enough is pathologically slow.

```
                            //!< reaching storage (<tt>O_SYNC</tt>).
                            //!< <tt>disable_safety_barriers</tt> can be used here.

    reads_and_metadata = 5, //!< Cache reads and writes of metadata, but writes of data do not
                            //!< complete until reaching storage (<tt>O_DSYNC</tt>).
                            //!< <tt>disable_safety_barriers</tt> can be used here.

    all = 6,                //!< Cache reads and writes of data and metadata so they complete
                            //!< immediately, sending writes to storage at some point when the kernel
                            //!< decides (this is the default file system caching on a system).

    safety_barriers = 7,    //!< Cache reads and writes of data and metadata so they complete
                            //!< immediately, but issue safety barriers at certain points. See
                            //!< documentation for <tt>flag_disable_safety_barriers</tt>.

    temporary = 8           //!< Cache reads and writes of data and metadata so they complete
                            //!< immediately, only sending any updates to storage on last handle
                            //!< close in the system or if memory becomes tight as this file is
                            //!< expected to be temporary (Windows and FreeBSD only).
  };
```

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 (i) efficient or (ii) 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.

This author recently faced an interesting bug in the Lustre High Performance Compute (HPC) filesystem whereby if you wrote a lot of data to files with writeback caching (i.e. kernel caching is `all`), in certain corner cases it is possible to hang all Lustre clients across all nodes, which obviously is very bad. `caching::all` is what `iostream` does, so theoretically a similar problem could arise there too. We solved the problem using `caching::reads_and_metadata`, this causes back pressure to be applied to file writes according to network speed and Lustre capacity, thus pacing the write speed and avoiding writes accumulating in the local node's RAM.

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[4] 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

[4]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.

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.

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 perform 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 bitcopying, see [P1029] *SG14* *move* `=` *`bitcopies`*), cloneable[5], swappable, and closeable. Nothing else is provided.

6.2 Example of use of `path_handle`

```
// Open /home/ned/foo as a path handle
llfio::path_handle base("/home/ned/foo");
llfio::file_handle fileA(base, "fileA");
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`,

[5]C++ copy construction is disabled as duplicating a handle with the kernel is expensive. You can copy a handle instance using `.clone()`.

the path handle based lookup method demonstrated above could[6] 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:

```
namespace path_discovery
{
  //! \brief The default regex used to determine what temporary directories are backed by storage not
      memory.
  static constexpr const char storage_backed_regex[] =
  "btrfs|cifs|exfat|ext[2-4]|f2fs|hfs|apfs|jfs|lxfs|nfs[1-9]?|lustre|nilf2|ufs|vfat|xfs|zfs|msdosfs|
      newnfs|ntfs|smbfs|unionfs|fat|fat32|overlay2?";

  /*! \brief Returns a reference to an open handle to a verified temporary directory where files
  created are stored in a filesystem directory, usually under the current user's quota.

  This is implemented by 'verified_temporary_directories()' iterating all of the paths returned by
  and checking what file system is in use, comparing it to 'storage_backed_regex'.

  The handle is created during 'verified_temporary_directories()' and is statically cached thereafter.
  */
  const path_handle &storage_backed_temporary_files_directory() noexcept;
}
```

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`).

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.

[6] 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.

8 Walkthrough of `<handle type>::buffers_type`

Like in [P2586] *Standard Secure Networking*, LLFIO insists that scatter-gather buffers are layout compatible with those used by the host OS to avoid a repack, which in POSIX's case is a contiguous array of `struct iovec`. For Linux and most POSIX implementations, `struct iovec` is defined to be:

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

For Microsoft Windows, there are two host OS native scatter-gather buffer layouts. The first, used by sockets, is:

```
typedef struct _WSABUF {
  ULONG len;
  CHAR  *buf;
} WSABUF, *LPWSABUF;
```

The second requires uncached i/o to be in use, where all i/o is done in exact memory pages, and thus does not apply here.

By insisting upon layout compatibility, LLFIO guarantees that the compiler can optimise out the scatter gather buffer repacking. 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).

`<handle type>::buffers_type` and `<handle type>::const_buffers_type` are C++ types layout matching the host OS scatter-gather buffers. They are guaranteed to be a trivial type with standard layout (i.e. C compatible). They additionally implement:

- `.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, `buffers_type|const_buffers_type` 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

```
class file_handle
{
public:
  //! The path type used by this handle
  using path_type = filesystem::path;

  //! The file extent type used by this handle
  using extent_type = <implementation defined, but typically an unsigned integer 64 bits or more>;

  //! The memory extent type used by this handle
  using size_type = size_t;

  //! The scatter buffer type used by this handle.
  using buffer_type = <system iovec compatible>;

  //! The gather buffer type used by this handle.
  using const_buffer_type = <const system iovec compatible>;

  //! The scatter buffers type used by this handle.
  using buffers_type = like-span<buffer_type>;

  //! The gather scatter buffers type used by this handle.
  using const_buffers_type = like-span<const_buffer_type>;

  //! The device id type
  using dev_t = <system specific>;

  //! The inode id type
  using ino_t = <system specific>;

  //! The path view type used by this handle
  using path_view_type = path_view;

  //! The unique identifier type used by this handle
  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 reading, and a `const_buffers_type` and a `.write()` member function if it is capable of writing. For file handles, these are `span<>`'s 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. LLFIO defines this to an unsigned 128 bit integer type as that covers all the platforms it supports.

[P2586] *Standard Secure Networking* defines an additional `registered_buffer_kind` here, and so does the reference implementation of `file_handle` and `mapped_file_handle` for compatibility. Registered i/o buffers are by definition page size aligned and page size granularity, and so *could* be faster for file handles as they are ideal for DMA direct from userspace, but I am unaware of any OS kernel which does DMA direct from userspace for cached file i/o at the time of writing.

9.2 Stand-in result type

From R1 onwards, the APIs are `result`-based as per LEWG-I guidance.

[P1028] *SG14 `status_code` and standard `error` object* proposes a `std::result<T>`. To summarise that quickly for the purposes of this paper:

- A `result<T>` yields either a `T` or a `std::error` from P1028.
- `.has_value()` returns true if the result contains a `T`.
- `explicit operator bool()` returns true if the result contains a `T`.
- `.has_error()` returns true if the result contains a `std::error`.
- `.value()` returns `T&&` or `T&` if the result contains a `T`, else it calls `.error().throw_exception()`, which causes the `std::error` instance to throw a C++ exception appropriate to the error code domain of the code. For this library, that is always an exception type which inherits publicly from `std::filesystem_error`.
- `.error()` returns the `std::error&&` or `std::error&` if the result contains an error, else a `bad_result_access` exception is thrown.

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 and mapped file handle, the i/o result type is:

```
//! The i/o result type used by this handle. Guaranteed to be ‘TrivialType‘ apart from
//! construction, and ‘StandardLayoutType‘.
template <class T> struct io_result : result<T>
{
  constexpr io_result();

  //! Return the number of bytes transferred.
  constexpr size_type bytes_transferred() const noexcept;

  //! Return whether this i/o was incomplete.
  constexpr bool is_incomplete() const noexcept;
};
```

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

- For reads, `file_handle` modifies each buffer supplied to indicate the extent to which each was filled. The span of buffers returned indicates the total number of buffers modified. `mapped_file_handle` typically returns the input list truncated to one buffer, and sets that buffer to point into the map with a length equal or less to all the lengths of the input buffers added together.
- For writes, mapped file handle is the same as for file handle in that the buffers returned reflect the extent to which each buffer was drained, and how many buffers were modified. There is still a difference however: mapped file handles don't auto extend the length of their file if you write past the maximum extent of the file, whereas file handles do (the reason why is that there is no race free way to extend the length of a mapped file). If you write past the maximum extent of a file with a mapped file handle, you are returned partially drained buffers, which can only occur with file handle if an error occurred during the write.

Despite these differences in semantics, it is very possible to write code which works identically irrespective of whether the handle is a file or a mapped file handle – the custom DB at work will choose between the two implementations based on the characteristics of the file being opened, and the code which works with the handle generally doesn't need to know which implementation it is (with one exception: if reads return different pointers, it is assumed that is into a mapped file and a whole code path for allocating memory pages and reading into them is skipped, and the map is used directly).

It is thus very important to always use the buffers returned, and not assume anything about the buffers input, if you wish your code to work identically on both `file_handle` and `mapped_file_handle`. If you really do need to fill the specific memory pointed to by the buffers scattered, then in generic code you should loop over the input and output buffers, calling `memmove()` from output buffers to input buffers. If `file_handle` was your file handle implementation, `memmove()` will do nothing as the destination and source pointers will be identical.

This choice to modify buffers in-place 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, for most file i/o, one must 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 might issue all the buffer fills in parallel for improved performance.

As you will see below, there are convenience overloads in file and mapped file handle for `.read()` and `.write()` which take initialiser lists and return a bytes transferred count.

9.4 Constructors

```
//! Default constructor
constexpr file_handle();

//! Construct a handle from a supplied native handle
constexpr file_handle(native_handle_type h,
                      dev_t devid,
                      ino_t inode,
                      caching caching = caching::none,
                      flag flags = flag::none) noexcept;

//! No copy construction (use clone())
file_handle(const file_handle &) = delete;

//! No copy assignment
file_handle &operator=(const file_handle &) = delete;

//! Implicit move construction of file_handle permitted
constexpr file_handle(file_handle &&o) noexcept;

//! Explicit conversion from handle and io_handle permitted
explicit constexpr file_handle(handle &&o,
                               dev_t devid,
                               ino_t inode) noexcept;

//! Move assignment of file_handle permitted
file_handle &operator=(file_handle &&o) noexcept;

//! Swap with another instance
void swap(file_handle &o) noexcept;

//! Clone this handle (copy constructor is disabled to avoid accidental copying)
result<handle> clone() noexcept;

/*! Reopen this handle, optionally race free reopening the handle with different
access or caching.

Microsoft Windows provides a syscall for cloning an existing handle but with new
access. On POSIX, if not changing the mode, we change caching via ‘fcntl()‘, if
changing the mode we must loop calling ‘current_path()‘,
trying to open the path returned and making sure it is the same inode.

\errors Any of the values POSIX dup() or DuplicateHandle() can return.
\mallocs On POSIX if changing the mode, we must loop calling ‘current_path()‘ and
trying to open the path returned. Thus many allocations may occur.
*/
result<file_handle> reopen(mode mode_ = mode::unchanged,
                           caching caching_ = caching::unchanged,
                           deadline d = std::chrono::seconds(30)) const noexcept;

//! Immediately close the native handle type managed by this handle
result<void> close() noexcept;

//! Release the native handle type managed by this handle
native_handle_type release() noexcept;
```

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 what the Networking TS did.

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.

File handles are not implicitly copyable as duplication involves a kernel syscall. `.clone()`, which all LLFIO handles implement, provides explicit copying. File handle extends cloning via `.reopen()` 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. This is a very useful facility.

9.5 Free function constructors

9.5.1 Creating and opening files

```
/*! Create a file handle opening access to a file on path.
\param base Handle to a base location on the filing system. Pass ‘{}‘ to indicate that path will
be absolute.
\param path The path relative to base to open.
\param _mode How to open the file.
\param _creation How to create the file.
\param _caching How to ask the kernel to cache the file.
\param flags Any additional custom behaviours.

\errors Any of the values POSIX open() or CreateFile() can return.
*/
result<file_handle> file(const path_handle &base,
                         path_view_type path,
                         mode _mode = mode::read,
                         creation _creation = creation::open_existing,
                         caching _caching = caching::all,
                         flag flags = flag::none) noexcept;
```

This is the primary way of creating or opening a file, and returning an open handle to it.

9.5.2 Creating uniquely named files

```
/*! Create a file handle creating a uniquely named file on a path.
The file is opened exclusively with ‘creation::only_if_not_exist‘ so it
will never collide with nor overwrite any existing file. Note also
that caching defaults to temporary which hints to the OS to only
flush changes to physical storage as lately as possible.

\errors Any of the values POSIX open() or CreateFile() can return.
```

```
*/
result<file_handle> uniquely_named_file(const path_handle &dirpath,
                                        mode _mode = mode::write,
                                        caching _caching = caching::temporary,
                                        flag flags = flag::none) noexcept;
```

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 unique file entry 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.

9.5.3 Creating uniquely named temporary files

```
/*! Create a file handle creating the named file on some path which
the OS declares to be suitable for temporary files. Most OSs are
very lazy about flushing changes made to these temporary files.
Note the default flags are to have the newly created file deleted
on first handle close.
Note also that an empty name is equivalent to calling
‘uniquely_named_file(path_discovery::storage_backed_temporary_files_directory())‘
and the creation parameter is ignored.

\note If the temporary file you are creating is not going to have its
path sent to another process for usage, this is the WRONG function
to use. Use ‘temp_inode()‘ instead, it is far more secure.

\errors Any of the values POSIX open() or CreateFile() can return.
*/
result<file_handle> temp_file(path_view_type name = path_view_type(),
                              mode _mode = mode::write,
                              creation _creation = creation::if_needed,
                              caching _caching = caching::temporary,
                              flag flags = flag::unlink_on_first_close) noexcept;
```

A very common use case is to create a uniquely 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 on first close, 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.

9.5.4 Creating anonymous temporary inodes

```
/*! \em Securely create a file handle creating a temporary anonymous inode in
the filesystem referred to by \em dirpath. The inode created has
no name nor accessible path on the filing system and ceases to
exist as soon as the last handle is closed, making it ideal for use as
a temporary file where other processes do not need to have access
to its contents via some path on the filing system (a classic use case
is for backing shared memory maps).

```

```
\errors Any of the values POSIX open() or CreateFile() can return.
*/
result<file_handle> temp_inode(const path_handle &dirh
                                    = path_discovery::storage_backed_temporary_files_directory(),
                                mode _mode = mode::write,
                                flag flags = flag::none) noexcept;
```

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 virtual memory based 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 some other systems, it is possible to create a uniquely named file without the ability for anyone else (apart from root) to access it, and immediately unlink it. If a system cannot create a temporary inode without the possibility for concurrent actors to gain access to the contents of the newly created inode using a timing attack, then this call must fail.

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

```
/*! Returns the current path of the open handle as said by the operating system. Note
that you are NOT guaranteed that any path refreshed bears any resemblance to the original,
some operating systems will return some different path which still reaches the same inode
via some other route e.g. hardlinks, dereferenced symbolic links, etc. Windows and
Linux correctly track changes to the specific path the handle was opened with,
not getting confused by other hard links. MacOS nearly gets it right, but under some
circumstances e.g. renaming may switch to a different hard link's path which is almost
certainly a bug.

If LLFIO was not able to determine the current path for this open handle e.g. the inode
has been unlinked, it returns an empty path. Be aware that FreeBSD can return an empty
(deleted) path for file inodes no longer cached by the kernel path cache, LLFIO cannot
detect the difference. FreeBSD will also return any path leading to the inode if it is
hard linked. FreeBSD does implement path retrieval for directory inodes
correctly however, and see `algorithm::cached_parent_handle_adapter<T>` for a handle
adapter which makes use of that.

On Linux if `/proc` is not mounted, this call fails with an error. All APIs in LLFIO
which require the use of `current_path()` can be told to not use it e.g.
`flag::disable_safety_unlinks`.
It is up to you to detect if `current_path()` is not working, and to change how you
call LLFIO appropriately.

\warning This call is expensive, it always asks the kernel for the current path, and no
checking is done to ensure what the kernel returns is accurate or even sensible.
Be aware that despite these precautions, paths are unstable and **can change randomly at
any moment**. Most code written to use absolute file systems paths is **racy**, so don't
```

```
do it, use ‘path_handle‘ to fix a base location on the file system and work from that anchor
instead!

\mallocs At least one malloc for the ‘path_type‘, likely several more.

\sa ‘algorithm::cached_parent_handle_adapter<T>‘ which overrides this with an
implementation based on retrieving the current path of a cached handle to the parent
directory. On platforms with instability or failure to retrieve the correct current path
for regular files, the cached parent handle adapter works around the problem by
taking advantage of directory inodes not having the same instability problems on any
platform.
*/
result<path_type> current_path() const noexcept;
```

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 even under very heavy concurrent load, and plenty of commercial production code based on LLFIO has been absolutely reliable for years now. 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[7]. 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 committee members will disagree. I will say that this API is a hard requirement, it cannot be omitted without severely compromising the usefulness

[7]My thanks to the Mac OS kernel developers for recently fixing how their path retrieval API works with unlinked inodes.

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, usually by opening each DOS format path and comparing inode unique ids for equivalence.

9.6.2 Retrieving the maximum scatter-gather buffer quantity

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

Note that the actual number of buffers accepted for a read or a write may be significantly
lower than this system-defined limit, depending on available resources. The `read()` or `write()`
call will return the buffers accepted.

Note also that some OSs will error out if you supply more than this limit to `read()` or `write()`,
but other OSs do not. Some OSs guarantee that each i/o syscall has effects atomically visible or not
to other i/o, other OSs do not.

OS X does not implement scatter-gather file i/o syscalls.
Thus this function will always return `1` in that situation.

Microsoft Windows *may* implement scatter-gather file i/o under very limited circumstances.
Most of the time this function will return `1`.
*/
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.

Obviously enough for mapped file handles this function always returns zero, because read and write the implementation uses `memmove` on the map and therefore both reads and writes always tear.

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

```
/*! Obtain a handle to the path **currently** containing this handle’s file entry.

\warning This call is \b racy and can result in the wrong path handle being returned. Note that
unless ‘flag::disable_safety_unlinks‘ is set, this implementation opens a
‘path_handle‘ to the source containing directory, then checks if the file entry within has the
same inode as the open file handle. It will retry this matching until
success until the deadline given.

\mallocs Calls ‘current_path()‘ and thus is both expensive and calls malloc many times.

\sa ‘algorithm::cached_parent_handle_adapter<T>‘ which overrides this with a zero cost
implementation, thus making unlinking and relinking very considerably quicker.
*/
template<class Rep, class Period>
result<path_handle> parent_path_handle(const std::chrono::duration<Rep, Period> &duration
                                           = <implementation determined>) const noexcept;

//! \overload Convenience absolute based overload
template<class Clock, class Duration>
result<path_handle> parent_path_handle(const std::chrono::time_point<Clock, Duration> &timeout)
    noexcept;
```

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. The default duration timeout chosen by the reference library is thirty seconds, but any sensible value would do.

For inodes with exclusively binary identifiers, or for files in the root directory of their storage, the path handle returned is to their owning volume or device.

9.6.4 Retrieving the unique identifier of an open handle

```
//! The device id of the file when opened
dev_t st_dev() const noexcept;

//! The inode of the file when opened. When
//! combined with st_dev(), forms a unique identifer on this system
ino_t st_ino() const noexcept;

//! A unique identifier for this handle across the entire system. Can be used in hash tables etc.
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.

Some may not be aware that Microsoft Windows also guarantees there is a unique identifier of an inode within a system, same as POSIX does.

9.6.5 Retrieving the current maximum extent of an open handle

```
/*! Return the current maximum permitted extent of the file.

\errors Any of the values POSIX fstat() or GetFileInformationByHandleEx() can return.
*/
result<extent_type> maximum_extent() const noexcept;
```

Files are stored as a sequence 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.

Be aware that on any recent filing system, sparse file support allows the number returned by this call to be ludicrously high e.g. 18,446,744,073,709,551,614, even if the data stored by the file's content is zero bytes.

9.6.6 Retrieving the current valid extents of an open handle

```
//! \brief A pair of valid extents
struct extent_pair
{
  extent_type offset{(extent_type) -1};  //!< The offset of where the valid extent begins
  extent_type length{(extent_type) -1};  //!< The number of valid bytes in the valid extent

  constexpr extent_pair() {}
  constexpr extent_pair(extent_type _offset, extent_type _length)
      : offset(_offset)
      , length(_length)
  {
  }
};

/*! \brief Returns a list of currently valid extents for this open file. WARNING: racy!
\return A vector of pairs of extent offset + extent length representing the valid extents
in this file. Filing systems which do not support extents return a single extent matching
the length of the file rather than returning an error.
*/
result<std::vector<extent_pair>> extents() const noexcept;
```

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' at the offset of the hash modulus a Terabyte. 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. The API below `.clone_extents_to()` is extents-aware, and will try its best to replicate the sparseness of one file's contents to another.

9.6.7 Miscellaneous observers

```
//! True if the handle is valid (and usually open)
bool is_valid() const noexcept;

//! True if the handle is readable
bool is_readable() const noexcept;
//! True if the handle is writable
bool is_writable() const noexcept;
//! True if the handle is append only
bool is_append_only() const noexcept;

//! True if multiplexable
bool is_multiplexable() const noexcept;
//! True if nonblocking
bool is_nonblocking() const noexcept;
//! True if seekable
bool is_seekable() const noexcept;
//! True if requires aligned i/o
bool requires_aligned_io() const noexcept;
//! True if native handle is a valid kernel handle
bool is_kernel_handle() const noexcept;

//! True if a regular file or device
bool is_regular() const noexcept;
//! True if a directory
bool is_directory() const noexcept;
//! True if a symlink
bool is_symlink() const noexcept;
//! True if a pipe
bool is_pipe() const noexcept;
//! True if a socket
```

```
bool is_socket() const noexcept;
//! True if a multiplexer like BSD kqueues, Linux epoll or Windows IOCP
bool is_multiplexer() const noexcept;
//! True if a process
bool is_process() const noexcept;
//! True if a memory section
bool is_section() const noexcept;
//! True if a memory allocation
bool is_allocation() const noexcept;
//! True if a path or directory
bool is_path() const noexcept;
//! True if a TLS socket
bool is_tls_socket() const noexcept;
//! True if a HTTP socket
bool is_http_socket() const noexcept;

//! Kernel cache strategy used by this handle
caching kernel_caching() const noexcept;
//! True if the handle uses the kernel page cache for reads
bool are_reads_from_cache() const noexcept;
//! True if writes are safely on storage on completion
bool are_writes_durable() const noexcept;
//! True if issuing safety fsyncs is on
bool are_safety_barriers_issued() const noexcept;

//! The flags this handle was opened with
flag flags() const noexcept;
//! The native handle used by this handle
native_handle_type native_handle() const noexcept;
```

[P2586] *Standard Secure Networking*'s i/o classes have identical miscellaneous observers, which may explain the apparently irrelevant observers.

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

```
/*! Relinks the current path of this open handle to the new path specified. If ‘atomic_replace‘ is
true, the relink \b atomically and silently replaces any item at the new path specified. This
operation is both atomic and silent matching POSIX behaviour even on Microsoft Windows where
no Win32 API can match POSIX semantics.

\warning Some operating systems provide a race free syscall for renaming an open handle (Windows).
On all other operating systems this call is \b racy and can result in the wrong file entry being
relinked. Note that unless ‘flag::disable_safety_unlinks‘ is set, this implementation opens a
‘path_handle‘ to the source containing directory first, then checks before relinking that the item
about to be relinked has the same inode as the open file handle. It will retry this matching until
success until the deadline given. This should prevent most unmalicious accidental loss of data.

```

```
\param base Base for any relative path.
\param path The relative or absolute new path to relink to.
\param atomic_replace Atomically replace the destination if a file entry already is present there.
Choosing false for this will fail if a file entry is already present at the destination, and may
not be an atomic operation on some platforms (i.e. both the old and new names may be linked to the
same inode for a very short period of time). Windows and recent Linuxes are always atomic.
\param d The deadline by which the matching of the containing directory to the open handle's inode
must succeed, else 'errc::timed_out' will be returned.
\mallocs Except on platforms with race free syscalls for renaming open handles (Windows), calls
'current_path()' via 'parent_path_handle()' and thus is both expensive and calls malloc many times.
*/
template<class Rep, class Period>
result<void> relink(const path_handle &base,
                    path_view_type path,
                    bool atomic_replace = true,
                    const std::chrono::duration<Rep, Period> &duration
                        = <implementation determined>) noexcept;

//! \overload Convenience absolute based overload
template<class Clock, class Duration>
result<void> relink(const path_handle &base,
                    path_view_type path,
                    bool atomic_replace = true,
                    const std::chrono::time_point<Clock, Duration> &timeout) noexcept;
```

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

```
/*! Links the inode referred to by this open handle to the path specified. The current path
of this open handle is not changed, unless it has no current path due to being unlinked.

\warning Some operating systems provide a race free syscall for linking an open handle to a new
location (Linux, Windows).
On all other operating systems this call is \b racy and can result in the wrong inode being
linked. Note that unless 'flag::disable_safety_unlinks' is set, this implementation opens a
'path_handle' to the source containing directory first, then checks before linking that the item
about to be hard linked has the same inode as the open file handle. It will retry this matching
until success until the deadline given. This should prevent most unmalicious accidental loss of
data.

\param base Base for any relative path.
\param path The relative or absolute new path to hard link to.
\param d The deadline by which the matching of the containing directory to the open handle's inode
must succeed, else 'errc::timed_out' will be returned.
\mallocs Except on platforms with race free syscalls for renaming open handles (Windows), calls
'current_path()' via 'parent_path_handle()' and thus is both expensive and calls malloc many times.
```

```
*/
template<class Rep, class Period>
result<void> link(const path_handle &base,
                  path_view_type path,
                  const std::chrono::duration<Rep, Period> &duration
                      = <implementation determined>) noexcept;

//! \overload Convenience absolute based overload
template<class Clock, class Duration>
result<void> link(const path_handle &base,
                  path_view_type path,
                  const std::chrono::time_point<Clock, Duration> &timeout) noexcept;
```

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. As with all the other APIs, LLFIO calls the platform syscalls and whatever errors they return, LLFIO returns.

9.7.3 Extents cloning

```
/*! \brief Clones the extents referred to by ‘extent‘ to ‘dest‘ at ‘destoffset‘. This
is how you ought to copy file content, including within the same file. This is
fundamentally a racy call with respect to concurrent modification of the files.

Some of the filesystems on the major operating systems implement copy-on-write extent
reference counting, and thus can very cheaply link a "copy" of extents in one file into
another file (at the time of writing - Linux: XFS, btrfs, ocfs2, smbfs; Mac OS: APFS;
Windows: ReFS, CIFS). Upon first write into an extent, only then is a new copy formed for the
specific extents being modified. Note that extent cloning is usually only possible in
cluster sized amounts, so if the portion you clone is not so aligned, new extents
will be allocated for the spill into non-aligned portions. Obviously, cloning an entire
file in a single shot does not have that problem.

Networked filing systems typically can also implement remote extent copying, such that
extents can be copied between files entirely upon the remote server, and avoiding the
copy going over the network. This is usually far more efficient.

This implementation first enumerates the valid extents for the region requested, and
only clones extents which are reported as valid. It
then iterates the platform specific syscall to cause the extents to be cloned in
‘utils::page_allocator<T>‘ sized chunks (i.e. the next large page greater or equal
to 1Mb). Generally speaking, if the dedicated syscalls fail, the implementation falls
back to a user space emulation, unless ‘emulate_if_unsupported‘ is false.

If the region being cloned does not exist in the source file, the region is truncated
```

```
to what is available. If the destination file is not big enough to receive the cloned
region, it is extended. If the clone is occurring within the same inode, you should
ensure that the regions do not overlap, as cloning regions which overlap has platform-specific
semantics. If they do overlap, you should always set ‘force_copy_now‘ for portable
code.

\note The current implementation does not permit overlapping clones within the same
inode to differ by less than ‘utils::page_allocator<T>‘ sized chunks. It will fail
with an error code comparing equal to ‘errc::invalid_parameter‘.

If you really want the copy to happen now, and not later via copy-on-write, set
‘force_copy_now‘. Note that this forces ‘emulate_if_unsupported‘ to true.

If ‘dest‘ is not a ‘file_handle‘, ‘sendfile()‘ is used and the destination offset
and gaps in the source valid extents are ignored.
*/
result<extent_pair>
clone_extents_to(extent_pair extent,
                 byte_io_handle &dest,
                 byte_io_handle::extent_type destoffset,
                 deadline d = {},
                 bool force_copy_now = false,
                 bool emulate_if_unsupported = true) noexcept;

//! \overload Convenience overload for mirroring the whole contents
result<extent_pair>
clone_extents_to(byte_io_handle &dest,
                 deadline d = {},
                 bool force_copy_now = false,
                 bool emulate_if_unsupported = true) noexcept;
```

If you want to make a destination file – or socket – have the same contents as a source file, this function is the one to use. It is extents aware, and so will do a fair attempt at reproducing the sparse layout of any source file into the destination by punching holes in the destination where there are holes in the source. Unless you request `force_copy_now`, the function will attempt to use syscalls to perform the clone using ‘cheap copies’ i.e. the time taken is independent of the quantity of data cloned, which includes using any ‘send file’ socket API if the destination is a socket.

Unfortunately, some file systems don’t properly implement the cheap clone syscalls e.g. Lustre, which claims it does, but currently actually fails. If this occurs, you can have the API error out by setting `emulate_if_unsupported` to false, or fall back to manually copying the data (the default).

Implementers should be aware that correctly implementing this API whilst keeping it performant is quite challenging, and probably is one of the hardest parts of reimplementing LLFIO. It took me quite a lot of time to implement a correct algorithm, and even then there were quite a few months of corner case surprises once into production.

9.7.4 Unlinking

```
/*! Unlinks the current path of this open handle, causing its entry to immediately disappear
from the filing system. On Windows before Windows 10 1709 unless
‘flag::win_disable_unlink_emulation‘ is set, this behaviour is simulated by renaming the file
```

```
to something random and setting its delete-on-last-close flag. Note that Windows may prevent
the renaming of a file in use by another process, if so it will NOT be renamed.
After the next handle to that file closes, it will become permanently unopenable by anyone
else until the last handle is closed, whereupon the entry will be eventually removed by the
operating system.

\warning Some operating systems provide a race free syscall for unlinking an open handle (Windows).
On all other operating systems this call is \b racy and can result in the wrong file entry being
unlinked. Note that unless ‘flag::disable_safety_unlinks‘ is set, this implementation opens a
‘path_handle‘ to the containing directory first, then checks that the item about to be unlinked
has the same inode as the open file handle. It will retry this matching until success until the
deadline given. This should prevent most unmalicious accidental loss of data.

\param d The deadline by which the matching of the containing directory to the open handle’s inode
must succeed, else ‘errc::timed_out‘ will be returned.
\mallocs Except on platforms with race free syscalls for unlinking open handles (Windows), calls
‘current_path()‘ and thus is both expensive and calls malloc many times. On Windows, also calls
‘current_path()‘ if ‘flag::disable_safety_unlinks‘ is not set.
*/
template<class Rep, class Period>
result<void> unlink(const std::chrono::duration<Rep, Period> &duration
                        = <implementation determined>) noexcept;

//! \overload Convenience absolute based overload
template<class Clock, class Duration>
result<void> unlink(const std::chrono::time_point<Clock, Duration> &timeout) noexcept;
```

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.

I believe since R1 the Linux kernel has begun implementing direct syscall support for this call. I vaguely remember seeing mention of it on the FreeBSD kernel mailing list as well.

9.7.5 Truncating extents and setting a new maximum extent

```
/*! Resize the current maximum permitted extent of the file to the given extent, avoiding any
new allocation of physical storage where supported. Note that on extents based filing systems
this will succeed even if there is insufficient free space on the storage medium.

\return The bytes actually truncated to.
\param newsize The bytes to truncate the file to.
\errors Any of the values POSIX ftruncate() or SetFileInformationByHandle() can return.
*/
result<extent_type> truncate(extent_type newsize) noexcept;
```

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

```
/*! \brief Efficiently zero, and possibly deallocate, data on storage.

On most major operating systems and with recent filing systems which are "extents based", one can
deallocate the physical storage of a file, causing the space deallocated to appear all bits zero.
This call attempts to deallocate whole pages (usually 4Kb) entirely, and memset's any excess to all
bits zero. This call works on most Linux filing systems with a recent kernel, Microsoft Windows
with NTFS, and FreeBSD with ZFS. On other systems it simply writes zeros.

\return The bytes zeroed.
\param offset The offset to start zeroing from.
\param bytes The number of bytes to zero.
\param d An optional deadline by which the i/o must complete, else it is cancelled.
Note function may return significantly after this deadline if the i/o takes long to cancel.
\errors Any of the values POSIX write() can return, ‘errc::timed_out‘, ‘errc::operation_canceled‘.
‘errc::not_supported‘ may be returned if deadline i/o is not possible with this particular
handle configuration (e.g. writing to regular files on POSIX or writing to a non-overlapped
HANDLE on Windows).
\mallocs The default synchronous implementation in file_handle performs no memory allocation.
The asynchronous implementation in async_file_handle may perform one calloc and one free.
*/
template<class Rep, class Period>
result<extent_type> zero(extent_type offset,
                         extent_type bytes,
                            const std::chrono::duration<Rep, Period> &duration
                                = <implementation determined>) noexcept;

//! \overload Convenience absolute based overload
template<class Clock, class Duration>
result<extent_type> zero(extent_type offset,
                         extent_type bytes,
                         const std::chrono::time_point<Clock, Duration> &timeout) noexcept;
```

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 popular 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 sparse 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

```
//! The kinds of write reordering barrier which can be performed.
enum class barrier_kind
{
  nowait_data_only,  //!< Barrier data only, non-blocking. This is highly optimised on NV-DIMM
                     //!< storage, but consider using 'nvram_barrier()' for even better performance.
  wait_data_only,    //!< Barrier data only, block until it is done. This is highly optimised on
                     //!< NV-DIMM storage, but consider using 'nvram_barrier()' for even better
                         performance.
  nowait_all,        //!< Barrier data and the metadata to retrieve it, non-blocking.
  wait_all           //!< Barrier data and the metadata to retrieve it, block until it is done.
};

/*! \brief Issue a write reordering barrier such that writes preceding the barrier will reach
storage before writes after this barrier.

\warning **Assume that this call is a no-op**. It is not reliably implemented in many common
use cases, for example if your code is running inside a LXC container, or if the user has mounted
the filing system with non-default options. Instead open the handle with 'caching::reads' which
means that all writes form a strict sequential order not completing until acknowledged by the
storage device. Filing system can and do use different algorithms to give much better performance
with 'caching::reads', some (e.g. ZFS) spectacularly better.

\warning Let me repeat again: consider this call to be a **hint** to poke the kernel with a stick
to go start to do some work sooner rather than later. **It may be ignored entirely**.

```

```
\warning For portability, you can only assume that barriers write order for a single handle
instance. You cannot assume that barriers write order across multiple handles to the same inode,
or across processes.

\return The buffers barriered, which may not be the buffers input. The size of each scatter-gather
buffer is updated with the number of bytes of that buffer barriered.
\param reqs A scatter-gather and offset request for what range to barrier. May be ignored on
some platforms which always write barrier the entire file. Supplying a default initialised reqs
write barriers the entire file.
\param kind Which kind of write reordering barrier to perform.
\param d An optional deadline by which the i/o must complete, else it is cancelled.
Note function may return significantly after this deadline if the i/o takes long to cancel.
\errors Any of the values POSIX fdatasync() or Windows NtFlushBuffersFileEx() can return.
\mallocs None.
*/
io_result<const_buffers_type> barrier(io_request<const_buffers_type> reqs
                                          = io_request<const_buffers_type>(),
                                      barrier_kind kind = barrier_kind::nowait_data_only) noexcept;

//! \overload Convenience nonblocking overload, note result has a boolean test
io_result<const_buffers_type> try_barrier(io_request<const_buffers_type> reqs
                                              = io_request<const_buffers_type>(),
                                          barrier_kind kind = barrier_kind::nowait_data_only)
                                              noexcept;

//! \overload Convenience duration based overload
template<class Rep, class Period>
io_result<const_buffers_type> try_barrier_for(io_request<const_buffers_type> reqs
                                                  = io_request<const_buffers_type>(),
                                              barrier_kind kind = barrier_kind::nowait_data_only,
                                              const std::chrono::duration<Rep, Period> &duration)
                                                  noexcept;

//! \overload Convenience absolute based overload
template<class Clock, class Duration>
io_result<const_buffers_type> try_barrier_until(io_request<const_buffers_type> reqs
                                                    = io_request<const_buffers_type>(),
                                                barrier_kind kind = barrier_kind::nowait_data_only,
                                                const std::chrono::time_point<Clock, Duration> &
                                                    timeout) noexcept;
```

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 Advisory whole file locks

SG1 Concurrency feedback from the Belfast meeting recommended the replacement of byte range locks with whole file locks, in order to avoid byte range lock insanity issues on POSIX.

```
/*! \brief Locks the inode referred to by the open handle for exclusive access.

Note that this may, or may not, interact with the byte range lock extensions. See 'unique_file_lock'
for a RAII locker.
\errors Any of the values POSIX 'flock()' can return.
\mallocs The default synchronous implementation in 'file_handle' performs no memory allocation.
The asynchronous implementation in 'async_file_handle' performs one calloc and one free.
*/
result<void> lock_file() noexcept;

/*! \brief Tries to lock the inode referred to by the open handle for exclusive access,
returning 'false' if lock is currently unavailable.

Note that this may, or may not, interact with the byte range lock extensions. See 'unique_file_lock'
for a RAII locker.
\errors Any of the values POSIX 'flock()' can return.
\mallocs The default synchronous implementation in 'file_handle' performs no memory allocation.
The asynchronous implementation in 'async_file_handle' performs one calloc and one free.
*/
bool try_lock_file() noexcept;

/*! \brief Unlocks a previously acquired exclusive lock.
*/
void unlock_file() noexcept;


/*! \brief Locks the inode referred to by the open handle for shared access.

Note that this may, or may not, interact with the byte range lock extensions. See 'unique_file_lock'
for a RAII locker.
\errors Any of the values POSIX 'flock()' can return.
```

```
\mallocs The default synchronous implementation in ‘file_handle‘ performs no memory allocation.
The asynchronous implementation in ‘async_file_handle‘ performs one calloc and one free.
*/
result<void> lock_file_shared() noexcept;

/*! \brief Tries to lock the inode referred to by the open handle for shared access,
returning ‘false‘ if lock is currently unavailable.

Note that this may, or may not, interact with the byte range lock extensions. See ‘unique_file_lock‘
for a RAII locker.
\errors Any of the values POSIX ‘flock()‘ can return.
\mallocs The default synchronous implementation in ‘file_handle‘ performs no memory allocation.
The asynchronous implementation in ‘async_file_handle‘ performs one calloc and one free.
*/
bool try_lock_file_shared() noexcept;

/*! \brief Unlocks a previously acquired shared lock.
*/
void unlock_file_shared() noexcept;
```

One will note the lack of `_for()` and `_until()` editions. There is no syscall for support for implementing those on POSIX, so only Windows could implement them efficiently. I therefore leave them off for later standardisation.

The astute will have noticed that the above whole file lock API meets `SharedLockable`, except for the presence of `_file` in the API names. The difference is that the locks are not thread aware, they apply to each file handle instance instead.

The question for SG1 Concurrency to answer next is which of the following to do:

- Should we add awareness of `file_handle` to `std::shared_lock` which can then be the RAII lock manager for both mutexes and files?
- Or should we keep `std::shared_lock` exclusively for thread-focused locks, and add something like a `std::shared_file_lock` or similar which duplicates `std::shared_lock`, but for files?

I know from the Prague meeting that Olivier feels that these locks need to be kernel thread aware so they meet the `SharedLockable` concept, however that is a lot of added overhead and a lot of use cases don't need kernel thread awareness. In fact, in anything I've written for production to date, I've not yet *ever* needed these to be kernel thread aware because you generally `clone()` a handle instance per thread, and you don't share handle instances across threads if you are going to be using them for IPC with other processes.

This may seem counterintuitive, however one doesn't generally use whole file locks to synchronise within the same program because they are many orders of magnitude slower than a mutex, or indeed, anything else. Where they really shine is synchronising between processes, and whilst somewhat slow compared to a `std::mutex`, most of the time they're fast enough for that use case.

9.8 Extended attributes

```
/*! \brief Fill the supplied buffer with the names of all extended attributes set on this file
```

```
or directory, returning a span of path view components.

Note that this routine is a very thin wrap of ‘listxattr()‘ on POSIX and ‘NtQueryInformationFile()‘
on Windows. If the supplied buffer is too small, the syscall typically returns failure rather
than do a partial fill. Most implementations do not support more than 64Kb of extended attribute
information per inode so maybe 70Kb is a safe default (to account for the return value storage),
however properly written code will detect the buffer being too small and will auto-expand it until
success.

\note On Windows, this is the list of alternate streams on a file, NOT NTFS extended attributes.

\raceguarantees The list of extended attributes is fetched in a single syscall. This may be an
atomically consistent snapshot.
*/
result<span<path_view_component>> list_extended_attributes(span<byte> tofill) const noexcept;

/*! \brief Retrieve the value of an extended attribute set on this file or directory.

\note On Windows, this is the list of alternate streams on a file, NOT NTFS extended attributes.
*/
result<span<byte>> get_extended_attribute(span<byte> tofill, path_view_component name) const
    noexcept;

/*! \brief Sets the value of an extended attribute on this file or directory.

To prevent collision in a globally visible resource, there is a convention whereby you ought to
namespace the names of your values as ‘namespace.attribute‘
e.g. ‘appname.setting‘ to prevent unintentional collision with other programs. Obviously, do choose
a unique ‘appname‘ if there is any chance another program might use the same namespace name.

On POSIX, there are additional namespacing requirements: before your value name, you need to prefix
one of ‘user‘ or ‘system‘, so the actual name you might set would be ‘user.appname.propname‘.
Windows does not have the ‘user‘/‘system‘ prefix requirement, but it does no harm to do the exact
same on Windows as on POSIX.

The host OS and target filing system choose the limits on value size, and will fail accordingly.
Some impose a maximum of 64Kb for all names and values per inode, others have a 4Kb maximum
value size, there are lots of combinations. You are probably safest not setting many names,
and keep the values short.

\warning Extended attributes are ’brittle’ because they can get silently wiped at any moment.
Never store anything in extended attributes which cannot be recalculated if missing. The ideal
use case for extended attributes is as a cache of additional metadata about a file or
directory e.g. "I last checked this directory at timestamp X", or "the MD5 hash at last modified
timestamp X for this file was Y". Also remember that other
processes can and do arbitrarily modify extended attributes concurrent to you.

### Windows only

This API is implemented as file alternate data streams, rather than the Extended Attributes
API as accessed via ‘NtSetEaFile()‘ and ‘NtQueryEaFile()‘ (which actually modify the file
alternate data stream ‘::$EA‘ in any case).

The reason why is that ‘NtSetEaFile()‘ can only **append** new records to EA storage. It cannot
deallocate any existing EA records, if you try to do so you will get ‘STATUS_EA_CORRUPT_ERROR‘.
You can append setting the same name to a different value, which can include a null value which then
```

```
  appears as if the name is no longer there. But there is a cap of 64kB for the EA record, and
  once it is consumed, it is gone forever for that inode.

  Obviously that doesn’t map at all well onto POSIX extended attributes, where you can set the
  value of an attribute as frequently as you like. The closest equivalent on Windows is therefore
  file alternate data streams, even though the attribute’s value is then worked with as a whole
  proper file with all the attendant performance consequences.

  As a result, ‘name‘ must be a valid filename and not contain any characters not permitted in
  a filename. We use the NT API here, so the character restrictions are far fewer than for the
  Win32 API e.g. single character names do NOT cause misoperation like on Win32.
  */
  result<void> set_extended_attribute(path_view_component name, span<const byte> value) noexcept;

  /*! \brief Removes the extended attribute set on this file or directory.

  \note On Windows, this is the list of alternate streams on a file, NOT NTFS extended attributes.
  We do not prevent you trying to remove internal alternate streams, either.
  */
  result<void> remove_extended_attribute(path_view_component) noexcept;

  /*! \brief Copies the extended attributes from one entity to another, optionally replacing
  all the extended attributes on the destination.

  This convenience function is implemented using the APIs above, and therefore is racy with
  respect to concurrent users. If you specifiy an invalid source with
  ‘replace_all_local_attributes = true‘, then this is a convenient way to remove all extended
  attributes on the local inode.

  \note This function uses 130Kb of stack and cannot handle attribute values larger than 64Kb.
  */
  result<size_t> copy_extended_attributes(const fs_handle &src, bool replace_all_local_attributes =
      false) noexcept;
```

Extended attributes are one of the oldest things in filesystems, yet very little code I’ve seen uses them, which I find a shame as they’re awfully useful. If you ever need to cache something ephemeral about a file e.g. the SHA256 of its content, or how recently you checked it for correctness, or whether it was downloaded from an insecure location and needs to be deeply checked, this is the right API. If you’re dealing with hundreds of millions of files, you can turn your `O(N^2)` algorithms from hideously slow to practical by chopping up the count of files and processing them in batches, annotating via extended attributes what you’ve done and have yet to do. The host OS and its applications will also set extended attributes for various things e.g. on Mac OS, the type of the file is set in extended attributes; on Microsoft Windows, Explorer will set from where you file originated; on Linux Samba will store Windows-specific metadata about files, and SELinux will store ACLs.

As one of the the oldest things in filesystems, syscall support for these is very good across most platforms. Indeed, Microsoft Windows offers two mutually incompatible implementations of extended attributes, just for kicks.

9.9 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 forty 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!

```
/*! \brief Read data from the open handle.

\warning Depending on the implementation backend, **very** different buffers may be returned than
you supplied. You should **always** use the buffers returned and assume that they point to different
memory and that each buffer’s size will have changed.

\return The buffers read, which may not be the buffers input. The size of each scatter-gather
buffer returned is updated with the number of bytes of that buffer transferred, and the pointer
to the data may be \em completely different to what was submitted (e.g. it may point into a
memory map).
\param buffers A list of scatter buffers.
\param offset The offset from which to read.
\param duration An optional deadline by which the i/o must complete, else it is cancelled.
Note function may return significantly after this deadline if the i/o takes long to cancel.
\errors Any of the values POSIX read() can return, ‘errc::timed_out‘, ‘errc::operation_canceled‘.
‘errc::not_supported‘ may be returned if deadline i/o is not possible with this particular handle
configuration (e.g. reading from regular files on POSIX or reading from a non-overlapped HANDLE
on Windows).
\mallocs The default synchronous implementation in file_handle performs no memory allocation.
The asynchronous implementation in async_file_handle performs one calloc and one free.
*/
io_result<buffers_type> read(buffers_type buffers,
                             extent_type offset) noexcept;

//! Convenience initialiser list based overload for ‘read()‘
result<size_type> read(std::initializer_list<buffer_type> lst,
                       extent_type offset) noexcept;

//! \overload Convenience nonblocking overload, note result has a boolean test
io_result<buffers_type> try_read(buffers_type buffers,
                                 extent_type offset) noexcept;

io_result<size_type> try_read(std::initializer_list<buffer_type> lst,
                              extent_type offset) noexcept;

//! \overload Convenience duration based overload
template<class Rep, class Period>
io_result<buffers_type> try_read_for(buffers_type buffers,
                                     extent_type offset,
                                     const std::chrono::duration<Rep, Period> &duration) noexcept;

template<class Rep, class Period>
io_result<size_type> try_read_for(std::initializer_list<buffer_type> lst,
                                  extent_type offset,
                                  const std::chrono::duration<Rep, Period> &duration) noexcept;

```

```
//! \overload Convenience absolute based overload
template<class Rep, class Period>
io_result<buffers_type> try_read_until(buffers_type buffers,
                                       extent_type offset,
                                       const std::chrono::time_point<Clock, Duration> &timeout)
                                           noexcept;

template<class Rep, class Period>
io_result<size_type> try_read_until(std::initializer_list<buffer_type> lst,
                                    extent_type offset,
                                    const std::chrono::time_point<Clock, Duration> &timeout)
                                        noexcept;
```

As explained earlier, every handle type in LLFIO defines its single buffer type, its plural buffers type, and its i/o result type. For `file_handle`, that is a type which quacks like `span<byte>`; plural buffers is a `span<span<byte>>`; 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::read()` – 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 may be modified, which can be verbose 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 instead of modifying the buffer list in-place.

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 only (ext4 implements them for uncached i/o just fine).

9.10 Gather writing to a file offset

```
/*! \brief Write data to the open handle.

\warning Depending on the implementation backend, not all of the buffers input may be written.
For example, with a zeroed deadline, some backends may only consume as many buffers as the system
has available write slots for, thus for those backends this call is "non-blocking" in the sense
that it will return immediately even if it could not schedule a single buffer write. Another example
is that some implementations will not auto-extend the length of a file when a write exceeds the
maximum extent, you will need to issue a ‘truncate(newsize)‘ first.
```

```

\return The buffers written, which may not be the buffers input. The size of each scatter-gather
buffer returned is updated with the number of bytes of that buffer transferred.
\param buffers A list of gather buffers.
\param offset The offset to which to write.
\param duration An optional deadline by which the i/o must complete, else it is cancelled.
Note function may return significantly after this deadline if the i/o takes long to cancel.
\errors Any of the values POSIX write() can return, ‘errc::timed_out‘, ‘errc::operation_canceled‘.
‘errc::not_supported‘ may be returned if deadline i/o is not possible with this particular handle
configuration (e.g. writing to regular files on POSIX or writing to a non-overlapped HANDLE on
Windows).
\mallocs The default synchronous implementation in file_handle performs no memory allocation.
The asynchronous implementation in async_file_handle performs one calloc and one free.
*/
io_result<const_buffers_type> write(const_buffers_type buffers,
                                    extent_type offset) noexcept;

//! Convenience initialiser list based overload for ‘write()‘
result<size_type> write(std::initializer_list<const_buffer_type> lst,
                        extent_type offset) noexcept;

//! \overload Convenience nonblocking overload, note result has a boolean test
io_result<const_buffers_type> try_write(const_buffers_type buffers,
                                        extent_type offset) noexcept;

io_result<size_type> try_write(std::initializer_list<const_buffers_type> lst,
                               extent_type offset) noexcept;

//! \overload Convenience duration based overload
template<class Rep, class Period>
io_result<const_buffers_type> try_write_for(const_buffers_type buffers,
                                            extent_type offset,
                                            const std::chrono::duration<Rep, Period> &duration)
                                                noexcept;

template<class Rep, class Period>
io_result<size_type> try_write_for(std::initializer_list<const_buffers_type> lst,
                                   extent_type offset,
                                   const std::chrono::duration<Rep, Period> &duration) noexcept;

//! \overload Convenience absolute based overload
template<class Rep, class Period>
io_result<const_buffers_type> try_write_until(const_buffers_type buffers,
                                              extent_type offset,
                                              const std::chrono::time_point<Clock, Duration> &
                                                  timeout) noexcept;

template<class Rep, class Period>
io_result<size_type> try_write_until(std::initializer_list<const_buffers_type> lst,
                                     extent_type offset,
                                     const std::chrono::time_point<Clock, Duration> &timeout)
                                         noexcept;
```

There is very little to mention here after explaining `.read()`. The array of byte arrays is written to the offset specified, or to the current 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 Walkthrough of `mapped_file_handle`

Mapped file handle is a complete implementation of file handle. In other words, any code which can accept a `file_handle&`, can also accept a `mapped_file_handle&`. This requires code which uses file handle to be written correctly to handle both implementations, which is not hard to do. It may seem unwise to allow this substitution given some of the differences I am about to describe, however the ability to runtime-configure whether an uncached i/o file or (by definition cached i/o) mapped file handle is used external to an i/o routine is *enormously* useful. It lets you write your often complex i/o code exactly once, and it will work at maximum efficiency for both cached and uncached use cases, without needing templates.

10.0.1 Reads are zero copy

The most noticeable difference in everyday use between them is that reads do not fill the buffers passed, but rather the buffers returned by reads point into the map. This avoids memory copying. So long as code always uses the buffers returned rather than assuming the buffers input will be written to, idempotence is assured. If one really does need the specific buffers sent to `read()` to be written into, one should always loop over the buffers returned, performing `memmove()` from each output buffer to each input buffer. If the pointers are identical i.e. your file handle is not mapped, `memmove()` will do nothing. It should be noted that the proposed normative wording for `file_handle::read()` specifically says that the buffers supplied may not be filled, and internal buffers can be returned instead. Other file handle implementations e.g. a ZIP archive file reader, would also return internal buffers, so it is not just the mapped file handle which would do this.

Because mapped memory is not definable in any likely soon edition of the current C++ standard, we simply define that the buffer pointers returned by `read()` are valid up until the mapped file handle is closed, truncated, updated or reserved. We work around the problem that writes into a new page in a mapped file can trigger `SIGBUS` or `SIGSEGV` due to running out of disc space by requiring the `write()` implementation to trap such events, and return an error matching `errc::no_space_on_device`. This lets us avoid having to mention mapped memory at all. Mapped file handles are as if a file handle implementation which keeps internal buffers of the file content.

10.0.2 Address space reservation

When you create a mapped file handle, you get to choose a number of bytes of address space to *reserve* for the mapping. This is very like the capacity setting in a `std::vector`. The file content can grow very efficiently on some platforms up to that reservation, and without its address in memory changing.

If you need to grow the file past the initial reservation, or access content written by other processes beyond your reservation, then you can ask for a larger reservation. This will tear down the existing map, create a new contiguous reservation of address space, and map the file into it. Almost always the address of the map would change, and thus any read buffers returned become invalidated. If you truncate a mapped file larger than its reservation, the reservation will automatically be expanded to the new larger size, just as `std::vector::resize()` would do. If you truncate a mapped file smaller than its reservation, the reservation is not modified.

On some platforms (Linux, BSD, Mac OS), if a concurrent process extends the maximum extent of the mapped file, all maps of that file anywhere in the system are automatically updated, up to their address space reservations i.e. newly written extents just magically appear in all maps in the system. On Microsoft Windows, you need to explicitly call the `.update_map()` function to have the map updated to match the new maximum extent of the mapped file, however the very first kernel thread in any process in the system to do this will update the map for **all** maps of that file anywhere in the system. The typical idiom for portability is that if you are about to read or write beyond the maximum file extent but before the reservation, call `.update_map()` and check the maximum file extent again. The same code on non-Microsoft platforms is a no-op, so you don't need to `#ifdef` anything.

10.0.3 Zero length files are very inefficient

Note that zero length files can NOT be memory mapped. This is a platform limitation. If the mapped file handle refers to a zero length file, or you cause it to refer to a zero length file, the maps are destroyed. As soon as you cause the file to have non-zero length, the address space reservation is created and the file mapped into it. If you have file handle based code where you oscillate frequently between non-zero file length and zero file length, such code will perform extremely badly with mapped file handle. Relatedly, if you have some high performance i/o which must perform consistently, always make sure to truncate a mapped file to a non-zero length before the hot i/o code path begins, and make sure to never zero the length of a file in that hot i/o code path either.

10.0.4 Automatic atomic append not supported

Writing off the maximum file extent in file handle causes automatic extension of the maximum file extent. This is performed by the system on your behalf. In mapped file handle, writing off the maximum file extent simply truncates the write i.e. the buffers returned are partial. This is because extending the map is an expensive operation which you really don't want to do frequently, and besides only the OS can atomically extend a file without race conditions. Therefore, file and mapped file handle idempotent code will always `.truncate()` before `.write()` off the end of the file, possibly locking the file for exclusive access if there could be concurrent users. Efficient code will truncate upwards by a large amount and fill in the gap with writes, rather than truncating small amounts every single write.

10.0.5 No exposure of map-based implementation

Finally, the utility of mapped file handle on 32-bit platforms is limited, relative to 64-bit platforms. Mapped file handle always maps all of the file from its beginning up to the reservation you chose. LLFIO's `map_handle` and `section_handle` allow far finer grained mapping (indeed, `mapped_file_handle` is implemented by LLFIO as a naïve combination of `file_handle`, `section_handle` and `map_handle`), however they could not be standardised without the standard defining concepts for the C++ virtual machine like virtual memory, memory pages, and page faulting. Finer control of memory maps is thus a C++ 26 or later targeted feature.

10.1 Constructors

Note that these are only the constructors different from `file_handle`:

```
//! Explicit conversion from file_handle permitted
explicit constexpr mapped_file_handle(file_handle &&o) noexcept;

//! Explicit conversion from file_handle permitted, this overload also attempts to map the file
explicit mapped_file_handle(file_handle &&o, size_type reservation) noexcept;
```

These constructors allow adopting of an existing file handle, and specifying an address space reservation for the mapped file handle to use.

10.2 Free function constructors

```
/*! Create a memory mapped file handle opening access to a file on path.
\param reservation The number of bytes to reserve for later expansion when mapping.
Zero means reserve only the current file length.
\param base Handle to a base location on the filing system. Pass ‘{}‘ to indicate
that path will be absolute.
\param _path The path relative to base to open.
\param _mode How to open the file.
\param _creation How to create the file.
\param _caching How to ask the kernel to cache the file.
\param flags Any additional custom behaviours.
\param offset The offset into the backing file for the map.

Note that if the file is currently zero sized, no mapping occurs now, but
later when ‘truncate()‘ or ‘update_map()‘ is called.

\errors Any of the values which the constructors for ‘file_handle‘, ‘section_handle‘
and ‘map_handle‘ can return.
*/
result<mapped_file_handle> mapped_file(size_type reservation,
                                       const path_handle &base,
                                       path_view_type _path,
                                       mode _mode = mode::read,
                                       creation _creation = creation::open_existing,
                                       caching _caching = caching::all,
                                       flag flags = flag::none,
                                       extent_type offset = 0) noexcept;

//! \overload Convenience overload with reservation set to zero, which uses the
//! current maximum extent of the file
result<mapped_file_handle> mapped_file(const path_handle &base,
                                       path_view_type _path,
                                       mode _mode = mode::read,
                                       creation _creation = creation::open_existing,
                                       caching _caching = caching::all,
                                       flag flags = flag::none,
                                       extent_type offset = 0) noexcept;

/*! Create an mapped file handle creating a uniquely named file on a path.
The file is opened exclusively with ‘creation::only_if_not_exist‘ so it
will never collide with nor overwrite any existing file. Note also
that caching defaults to temporary which hints to the OS to only
flush changes to physical storage as lately as possible.

\errors Any of the values POSIX open() or CreateFile() can return.
*/
result<mapped_file_handle> mapped_uniquely_named_file(size_type reservation,
                                                       const path_handle &dirpath,
                                                       mode _mode = mode::write,
                                                       caching _caching = caching::temporary,
                                                       flag flags = flag::none) noexcept;

/*! Create a mapped file handle creating the named file on some path which
the OS declares to be suitable for temporary files. Most OSs are
very lazy about flushing changes made to these temporary files.
Note the default flags are to have the newly created file deleted
on first handle close.
Note also that an empty name is equivalent to calling
‘mapped_random_file(path_discovery::storage_backed_temporary_files_directory())‘
and the creation parameter is ignored.

\note If the temporary file you are creating is not going to have its
path sent to another process for usage, this is the WRONG function
to use. Use ‘temp_inode()‘ instead, it is far more secure.

\errors Any of the values POSIX open() or CreateFile() can return.
*/
result<mapped_file_handle> mapped_temp_file(size_type reservation,
                                             path_view_type name = path_view_type(),
                                            mode _mode = mode::write,
                                            creation _creation = creation::if_needed,
                                            caching _caching = caching::temporary,
                                            flag flags = flag::unlink_on_first_close) noexcept;

/*! \em Securely create a mapped file handle creating a temporary anonymous inode in
the filesystem referred to by \em dirpath. The inode created has
no name nor accessible path on the filing system and ceases to
```

```
exist as soon as the last handle is closed, making it ideal for use as
a temporary file where other processes do not need to have access
to its contents via some path on the filing system (a classic use case
is for backing shared memory maps).

\errors Any of the values POSIX open() or CreateFile() can return.
*/
result<mapped_file_handle> mapped_temp_inode(size_type reservation = 0,
                                             const path_handle &dir = path_discovery::
                                                 storage_backed_temporary_files_directory(),
                                             mode _mode = mode::write,
                                             flag flags = flag::none) noexcept;
```

These mapped file handle constructors exactly match those for file handles, apart from an added reservation parameter.

10.3 Observers and Operations

Note that these are only the observers different from `file_handle`:

10.3.1 Accessing the map directly

```
//! The address in memory where this mapped file resides
volatile byte *address() const noexcept;

//! The offset into the backing file from which this mapped file begins
extent_type starting_offset() const noexcept;

//! The page size used by the map, in bytes.
size_type page_size() const noexcept;
```

A mapped file would be mapped somewhere in memory, with some granularity of page size, which might be larger than the system page size for improved efficiency with very high bandwidth storage devices (2Mb is common on Intel platforms with DAX mounted storage). Because the contents of a mapped file may change at any time, the pointer returned is `volatile`, which may be cast off by end user code if they know that the file will not be modified by others.

10.3.2 Capacity and Reservation

```
//! The address space (to be) reserved for future expansion of this file.
size_type capacity() const noexcept;

/*! \brief Reserve a new amount of address space for mapping future expansion of this file.
\param reservation The number of bytes of virtual address space to reserve. Zero means reserve
the current length of the underlying file.

Note that this is an expensive call, and ‘address()‘ may return a different value afterwards.
This call will fail if the underlying file has zero length.
*/
```

```
result<size_type> reserve(size_type reservation = 0) noexcept;
```

As mentioned earlier, you can define a reservation when creating a mapped file handle, or change the reservation later. This works very similarly to `std::vector` in that contiguous address space is allocated immediately for the reservation, and the portion of that address space which is used for the map is adjusted as the file content grows (or shrinks).

Passing in a reservation of zero bytes means to use the file's current maximum extent. This is perfect for files whose maximum extent does not change, but will be very inefficient for files whose maximum extent does change.

10.3.3 Updating the map

```
//! Return the current maximum permitted extent of the file.
result<extent_type> maximum_extent() const noexcept;

/*! \brief Resize the current maximum permitted extent of the mapped file to the given extent,
avoiding any new allocation of physical storage where supported, and mapping or unmapping any
new pages up to the reservation to reflect the new maximum extent. If the new size exceeds the
reservation, ‘reserve()‘ will be called to increase the reservation.

Note that on extents based filing systems
this will succeed even if there is insufficient free space on the storage medium. Only when
pages are written to will the lack of sufficient free space be realised, resulting in an
operating system specific exception.

\note On Microsoft Windows you cannot shrink a file with a section handle open on it in any
process in the system. We therefore *always* destroy the internal map and section before
truncating, and then recreate the map and section afterwards if the new size is not zero.
‘address()‘ therefore may change.
You will need to ensure all other users of the same file close their section and
map handles before any process can shrink the underlying file.

\return The bytes actually truncated to.
\param newsize The bytes to truncate the file to. Zero causes the maps to be closed before
truncation.
*/
result<extent_type> truncate(extent_type newsize) noexcept;

//! The maximum extent of the underlying file
result<extent_type> underlying_file_maximum_extent() const noexcept;

/*! \brief Efficiently update the mapping to match that of the underlying file,
returning the size of the underlying file.

This call is often considerably less heavyweight than ‘truncate(newsize)‘, and should be used where
possible.

If the internal section and map handle are invalid, they are restored unless the underlying file is
zero length.
*/
result<extent_type> update_map() noexcept;
```

The first two functions `.maximum_extent()` and `.truncate()` are the same as those for `file_handle`, but they have different implementations. The maximum extent returned by `mapped_file_handle` is not live, like it is with `file_handle`, but rather returns the current length of the mapped data. To update the value returned by `.maximum_extent()` to the current maximum extent of the file, one can either truncate the file to a new maximum extent to set a new value, or call `.update_map()` to read the current maximum extent, and adjust the map to match. As mentioned earlier in the discussion, `.update_map()` does almost no work on Linux, BSD and Mac OS, but does have a fair bit of first-caller overhead on Windows. On Microsoft Windows only, shrinking the file will cause a new address for the file to be chosen, because Windows does not permit the shrinking of a file which has open maps of its content, so there is no choice but to tear down all maps, shrink the file, and create new maps.

11 Conclusion

All of the above will probably seem like overkill for modern file i/o in the C++ standard. Other modern i/o proposals for C++ 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, once mastered what you get in return is *fine grained control*, which C++ has historically advertised as a key feature of itself.

Furthermore, if this approach towards low level i/o 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 `https://2misses.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 in 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. Those systems process Terabytes of data per day across multiple concurrent AWS nodes working on a mixture of local and networked (Lustre) file systems. It is hard to imagine such a solution being achievable with current standard library facilities, it just doesn't offer sufficient control.

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 standardised modern 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 ASIO. 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.

12 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
https://wg21.link/P1028

[P1029] Douglas, Niall
SG14 move = `bitcopies`
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

[P2300] Dominiak, Michal; Evtushenko, Georgy; Baker, Lewis; Teodorescu, Lucian Radu; Howes, Lee; Shoop, Kirk; Garland, Michael; Niebler, Eric; Adelstein Lelbach, Bryce,
`std::execution`
https://wg21.link/P2300

[P2586] Douglas, Niall
Standard Secure Networking
https://wg21.link/P2586

[Outcome] *(Boost.)Outcome*
https://ned14.github.io/outcome/

[POSIXext] *The Open Group Technical Standard, 2006, Extended API Set Part 2*
https://pubs.opengroup.org/onlinepubs/9699939699/toc.pdf