DGGGGR0 draft 2: Low level file i/o library

Document #: DGGGGR0 draft 2
Date: 2018-04-15
Project: Programming Language C++
Library Evolution Working Group
Reply-to: Niall Douglas
<s_sourceforge@nedprod.com>

A proposal for a low level file i/o library very thinly wrapping kernel syscalls into a portable standard library API, preserving all of the time and space complexities of the host platform.

See [PXXXX] *Data persistence and algorithms (`iostreams` v2) study group* for some of the interesting things one can build with this library.

A reference implementation of the proposed library with reference API documentation can be found at `https://ned14.github.io/afio/`. It works well on FreeBSD, MacOS, Linux and Microsoft Windows on ARM, AArch64, x64 and x86.

Changes since draft 1:

- Added FAQ entry on why bother at all with this pointless library?
- Added examples of use

Changes since draft 0 (paper including proposal for study group):

- Added detail about constructing a file handle.
- Added flags detail to `section_handle`.
- Added Frequently Asked Questions section.
- Broke off the 'big picture overview' into separate paper PXXXX.

Contents

1 Introduction

Why does the C++ standard need a low level file i/o library above and beyond needing one to build out an iostreams v2?

1.1 Latency to storage has become more important than it was

For a long time now, kernels have kept a cache of recently accessed filesystem data in order to improve read latencies, but also to buffer writes in order to reorder those writes into strides suitable for efficiently making use of a spinning hard drive's actuators. A randomly placed 4Kb i/o to main memory takes about 5 microseconds, whereas the same i/o to a hard drive takes at best 9,000 microseconds amortised. One could afford a few extra memory copies of an i/o without noticing a difference. Thus the standard library's `iostreams` does not worry too much about the multiple memory copies (in the whole system between the C++ code and the hard drive) that all the major

STL implementations make per i/o[1].

The rise of SSD storage has changed things. Now a SATA connected flash drive takes maybe 800 microseconds for that 4Kb i/o @ 99%[2], and random access is as fast as sequential access, so that is no longer an amortised latency figure hiding large individual i/o latency variance. Furthermore, flash based SSDs are highly concurrent, they can service between 16 and 32 concurrent random 4Kb i/o's (queue depth, QD) in almost the same time as a single random 4Kb i/o. These two differences profoundly transform how to write algorithms which work well on a filesystem, but it also has an important consequence for C++:

$$\frac{800\ microseconds}{32} = 25\ microseconds\ per\ 4Kb\ i/o\ amortised\ @\ 99\%.$$

On a SATA connected flash SSD with QD32, every unnecessary memory copy increases i/o cost by a minimum of 20%!

Achieving sustained QD32 is rare however – one needs to be performing large sequential blocks of i/o of at least 32 x 4Kb = 128Kb to have any chance of sustaining QD32, and for large sequential block i/o, latency is usually unimportant for most users[3].

However, just recently NVMe rather than SATA connected flash drives have become available to the mass market. These perform that random 4Kb i/o in just 300 microseconds @ 99%. At QD4, which is much more common than QD32, every unnecessary memory copy in the whole system increases i/o cost by 6%. If you are using `<iostream>` on a recent MacBook Pro (which has a high end NVMe flash SSD), perhaps 10% of your i/o cost is due to your choosing `<iostream>`. In my opinion, that is unacceptable in the C++ standard going forward.

And the march of technological progress will make things even worse soon. Intel's NVMe Optane drives using X-Point non-volatile memory will do that 4Kb i/o in just 35 microseconds @ 99% and 13 microseconds @ 50%, **and at QD1**. Every unnecessary memory copy in the whole system increases i/o cost by 14-38%. Next year NV-DIMMs will be standardised, at which point your non-volatile storage will do a 4Kb i/o in **8 microseconds**. Every unnecessary memory copy is now adding **60%** to i/o costs.

If C++ is to achieve the direction laid out in [P0939] *Direction for ISO C++*, in my opinion it needs a data persistence implementation which enables zero memory copies throughout the whole system. One will soon no longer be able to get away with anything less.

[1] All the major STL implementations implement `std::ofstream::write()` via the C function `fwrite()`. Because of buffering, `fwrite()` often calls `write()` multiple times. Each is an unavoidable memory copy into the kernel page cache, plus kernel transition. Eventually the dirty page in the kernel page cache will reach its age deadline, and be flushed to storage.

[2] The 99% means that 99% of i/o latencies will be below the given figure. All latency numbers in this section come from empirical testing by me on hardware devices. They differ significantly from manufacturer figures. Device manufacturers tend to quote the latency of the device without intervening filesystem or user space transition. All latency values quoted in this paper include intervening software systems, and are what a user space process can realistically expect to achieve.

[3] But not all. A past consulting client of mine had a problem whereby their application was applying real-time filters to *uncompressed* 8k video at a high frame rate. The CPU demands were not the problem, it was the storage subsystem: to get smooth video added an unacceptable amount of latency to the real-time video stream for their customers. This is exactly the sort of problem domain C++ ought to excel at.

1.2 The immature standard library support for file i/o leads to a lot of inefficient and buggy code and/or reinvention of the wheel

Memory mapped files, especially on 64 bit architectures, are usually a good reasonable default choice for most i/o to non-networked drives. They usually have superb sequential and random i/o performance, and usually cause no more than one memory copy in the whole system. Yet using them from standard C++ is not as trivial as one would imagine. Even with the Boost C++ Libraries to hand, there are two main mechanisms for mapping files into memory, and the plethora of questions about various corner case use issues on Stack Overflow would suggest that neither is entirely obvious to people. They are certainly not 'fire and forget', like a `std::ofstream` would be.

One area where a lot of people get stuck is how to efficiently *append* to a memory mapped file. Most developers – probably even most of the WG21 experts reading this paper right now – would suggest making the file much bigger and coordinate between your processes at what offset one 'appends' new data. They would suggest this because there is a widespread, *and completely inaccurate*, belief that memory maps are fixed size, and you must tear them down and recreate bigger ones in order to expand a map.

In fact, all the major platforms let you reserve address space for future expansion of a memory map. Indeed, often they will auto-expand your memory map into that reservation if the maximum extent of the backing file is increased, or they provide a super fast syscall for poking the kernel to expand maps of that file across the system. So, in fact, appending to memory mapped files without costly teardown and recreation of maps is fully supported by kernels, yet judging from Stack Overflow posts, very few realise this[4].

A standard library supplied implementation of a 'fire and forget' memory mapped file primitive object would help address these sorts of problem. The proposed low level file i/o library proposes a suite of polymorphic objects which can perform i/o. Code written to use them need not consider their implementation, thus allowing initiating code to choose whichever implementation is most suitable. Virtual function overrides then choose an optimised implementation, and the code need not worry itself about implementation details. Appends, for example, 'just work'.

2 Examples of use

A surprising number of people wanted examples of usage before any further discussion of the proposed library design. I therefore supply many such use examples, and my thanks to std-proposals for suggesting which.

I make the following caveats in the following use examples:

- These use examples are for the reference library written in C++ 14, not the proposed approximate C++ 23 library which assumes that something approximating [P0709] *Zero-overhead deterministic exceptions* have been added to the language. Specifically, this means that the `.value()` at the end of each call would not be there any more, and because `std::byte` is now in the language, we wouldn't have to keep reinterpret casting between `byte` and `char`.

[4] `https://stackoverflow.com/questions/4460507/appending-to-a-memory-mapped-file`

- This is a very low level library offering absolute maximum performance. It is correspondingly less convenient to use. Specifically, no single buffer overloads, no integration with STL containers, no serialisation/deserialisation, no dynamic memory allocation, no (traditional) exception throws. All these convenience APIs would be in a later standardised serialisation layer built on top of this layer.

- There is also no file length. Files do not have length. They have a maximum extent property. This property refers to the maximum possible extent offset which you will encounter when reading the extents which constitute the file's storage. It is extremely important to understand this difference: files, especially ones built using the planned generic filesystem algorithms template library, may regularly have a maximum extent in the Petabytes range, but store only a few Kb of extents. Algorithms and programs which treat the maximum extent as a length will perform *extremely* poorly in this situation.

  This is why we *truncate* files, we do not resize files, because we are truncating those extents exceeding the new maximum extent. We can also truncate to a later maximum extent.

2.1 Read an entire file into a vector assuming a single valid extent:

For brevity, the initial examples are lazy code which will suffer from pathologically poor performance on files with a large maximum extent. Later examples account for allocated extents.

```
namespace afio = AFIO_V2_NAMESPACE;

// Open the file for read
afio::file_handle fh = afio::file(  //
  {},       // path_handle to base directory
  "foo"     // path_view to path fragment relative to base directory
            // default mode is read only
            // default creation is open existing
            // default caching is all
            // default flags is none
).value();  // If failed, throw a filesystem_error exception

// Make a vector sized the current maximum extent of the file
std::vector<afio::byte> buffer(fh.maximum_extent().value());

// Synchronous scatter read from file
afio::file_handle::buffers_type filled = afio::read(
  fh,                                 // handle to read from
  0,                                  // offset
  {{ buffer.data(), buffer.size() }}  // Single scatter buffer of the vector
                                      // default deadline is infinite
).value();                            // If failed, throw a filesystem_error exception

// In case of racy truncation of file by third party to new length, adjust buffer to
// bytes actually read
buffer.resize(filled[0].len);
```

2.2 Write multiple gather buffers to a file:

```
namespace afio = AFIO_V2_NAMESPACE;

// Open the file for write, creating if needed, don't cache reads nor writes
afio::file_handle fh = afio::file(  //
  {},                                         // path_handle to base directory
  "hello",                                    // path_view to path fragment relative to base directory
  afio::file_handle::mode::write,             // write access please
  afio::file_handle::creation::if_needed,     // create new file if needed
  afio::file_handle::caching::only_metadata   // cache neither reads nor writes of data on this handle
                                              // default flags is none
).value();                                    // If failed, throw a filesystem_error exception

// Empty file
fh.truncate(0).value();

// Perform gather write
const char a[] = "hel";
const char b[] = "l";
const char c[] = "lo w";
const char d[] = "orld";

fh.write(0,                // offset
  {                        // gather list, buffers use std::byte
    { reinterpret_cast<const afio::byte *>(a), sizeof(a) - 1 },
    { reinterpret_cast<const afio::byte *>(b), sizeof(b) - 1 },
    { reinterpret_cast<const afio::byte *>(c), sizeof(c) - 1 },
    { reinterpret_cast<const afio::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
fh.close().value();
```

2.3 Map a file into memory and search it for a string (1):

```
namespace afio = AFIO_V2_NAMESPACE;

// Open the mapped file for read
afio::mapped_file_handle mh = afio::mapped_file(  //
  {},       // path_handle to base directory
  "foo"     // path_view to path fragment relative to base directory
            // default mode is read only
            // default creation is open existing
            // default caching is all
            // default flags is none
).value();  // If failed, throw a filesystem_error exception

auto length = mh.maximum_extent().value();

// Find my text
for (char *p = reinterpret_cast<char *>(mh.address());
     (p = (char *)memchr(p, 'h', reinterpret_cast<char *>(mh.address()) + length - p));
     p++)
{
  if (strcmp(p, "hello"))
  {
    std::cout << "Happy days!" << std::endl;
  }
}
```

2.4 Map a file into memory and search it for a string (2):

The preceding example used the wrap of other facilities into a convenience type `mapped_file_handle`. For more control and customisation, it can also be done by hand:

```
namespace afio = AFIO_V2_NAMESPACE;

// Open the file for read
afio::file_handle rfh = afio::file(  //
  {},       // path_handle to base directory
  "foo"     // path_view to path fragment relative to base directory
            // default mode is read only
            // default creation is open existing
            // default caching is all
            // default flags is none
).value();  // If failed, throw a filesystem_error exception

// Open the same file for atomic append
afio::file_handle afh = afio::file(  //
  {},                               // path_handle to base directory
  "foo",                            // path_view to path fragment relative to base directory
  afio::file_handle::mode::append   // open for atomic append
                                    // default creation is open existing
                                    // default caching is all
                                    // default flags is none
).value();                          // If failed, throw a filesystem_error exception

// Create a section for the file of exactly the current maximum extent of the file
afio::section_handle sh = afio::section(rfh).value();

// Map the end of the file into memory with a 1Mb address reservation
afio::map_handle mh = afio::map(sh, 1024 * 1024, sh.length().value() & ~4095).value();

// Append stuff to append only handle
afio::write(afh,
  0,                                                      // offset is ignored for atomic append only handles
  {{ reinterpret_cast<const afio::byte *>("hello"), 6 }}  // single gather buffer
                                                          // default deadline is infinite
).value();

// Poke map to update itself into its reservation if necessary to match its backing
// file, bringing the just appended text into the map. A no-op on many platforms.
size_t length = mh.update_map().value();

```

```
// Find my appended text
for (char *p = reinterpret_cast<char *>(mh.address());
     (p = (char *) memchr(p, ’h’, reinterpret_cast<char *>(mh.address()) + length - p));
     p++)
{
  if (strcmp(p, "hello"))
  {
    std::cout << "Happy days!" << std::endl;
  }
}
```

2.5 Sparsely stored arrays:

A neat use case is for sparsely allocated huge arrays. One can allocate up to 127Tb of address space on most 64 bit architectures.

```
namespace afio = AFIO_V2_NAMESPACE;

// Make me a 1 trillion element sparsely allocated integer array!
afio::mapped_file_handle mfh = afio::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
(void) mfh.truncate(1000000000000ULL * sizeof(int));

// Create a typed view of the one trillion integers
afio::algorithm::mapped_span<int> one_trillion_int_array(mfh);

// Write and read as you see fit, if you exceed physical RAM it’ll be paged out
one_trillion_int_array[0] = 5;
one_trillion_int_array[999999999999ULL] = 6;
```

2.6 Resumable i/o with Coroutines:

```
namespace afio = AFIO_V2_NAMESPACE;

// Create an asynchronous file handle
afio::io_service service;
afio::async_file_handle fh =
  afio::async_file(service, {}, "testfile.txt",
    afio::async_file_handle::mode::write,
    afio::async_file_handle::creation::if_needed).value();

// Resize it to 1024 bytes
truncate(fh, 1024).value();

// Begin to asynchronously write "hello world" into the file at offset 0,
// suspending execution of this coroutine until completion and then resuming
// execution. Requires the Coroutines TS.
alignas(4096) char buffer[] = "hello world";
co_await co_write(fh, 0, { { reinterpret_cast<afio::byte *>(buffer), sizeof(buffer) } }).value();
```

2.7 Read all valid extents of a file using asynchronous file i/o:

```
namespace afio = AFIO_V2_NAMESPACE;

// Create an i/o service to complete the async file i/o
afio::io_service service;

// Open the file for read
afio::async_file_handle fh = afio::async_file(  //
  service,  // The i/o service to complete i/o to
  {},       // path_handle to base directory
  "foo"     // path_view to path fragment relative to base directory
            // default mode is read only
            // default creation is open existing
            // default caching is all
            // default flags is none
).value();  // If failed, throw a filesystem_error exception

// Get the valid extents of the file.
const std::vector<
  std::pair<afio::file_handle::extent_type, afio::file_handle::extent_type>
> valid_extents = fh.extents().value();

// Schedule asynchronous reads for every valid extent
std::vector<std::pair<std::vector<afio::byte>, afio::async_file_handle::io_state_ptr>> buffers(valid_extents.size());
for (size_t n = 0; n < valid_extents.size(); n++)
{
  // Set up the scatter buffer
  buffers[n].first.resize(valid_extents[n].second);
  for(;;)
  {
    afio::async_file_handle::buffer_type scatter_req{ buffers[n].first.data(), buffers[n].first.size() };  // buffer to fill
    auto ret = afio::async_read( //
      fh,                                 // handle to read from
      { { scatter_req } , 0 },            // The scatter request buffers
      [](                                                                            // The completion handler
        afio::async_file_handle *,                                                   // The parent handle
        afio::async_file_handle::io_result<afio::async_file_handle::buffers_type> &  // Result of the i/o
        ) { /* do nothing */ }
      // default deadline is infinite
    );
    // Was the operation successful?
    if (ret)
    {
      // Retain the handle to the outstanding i/o
      buffers[n].second = std::move(ret).value();
      break;
    }
    if (ret.error() == std::errc::resource_unavailable_try_again)
    {
      // Many async file i/o implementations have limited total system concurrency
      std::this_thread::yield();
      continue;
    }
    // Otherwise, throw a filesystem_error exception
    ret.value();
  }
}

// Pump i/o completion until no work remains
while (service.run().value())
{
  // run() returns per completion handler dispatched if work remains
  // It blocks until some i/o completes (there is a polling and deadline based overload)
  // If no work remains, it returns false
}

// Gather the completions of all i/o scheduled for success and errors
for (auto &i : buffers)
{
  // Did the read succeed?
  if (i.second->result.read)
  {
    // Then adjust the buffer size to that actually read
    i.first.resize(i.second->result.read.value().size());
  }
  else
  {
    // Throw the cause of failure as an exception
    i.second->result.read.value();
  }
}
```

3 Impact on the Standard

The proposed low level file i/o library is potentially a pure-library solution with dependencies on:

1. `std::filesystem`
2. P0057 *C++ Extensions for Coroutines* https://wg21.link/P0057. Status: Approved for C++ 20.
3. P0122 *span: bounds-safe views for sequences of objects* https://wg21.link/P0122. Status: Approved for C++ 20.
4. P0734 *Concepts* https://wg21.link/P0734. Approved for C++ 20.
5. PAAAA *SG14 span<T> colouring* https://wg21.link/PAAAA.

   This proposes a set of attributes for colouring spans with extra metadata e.g. this span is aligned to a DMAable page boundary, this span is on persistent storage (please flush write buffers/use non-temporal stores) etc
6. PBBBB *SG14 status_ code and error* https://wg21.link/PBBBB.

This proposes a `<system_error>` v2 which fixes a number of problems which have emerged in the use of `<system_error>` as hindsight has emerged.

7. PDDDD *Filesystem path views* `https://wg21.link/PDDDD`.

   This proposes an always UTF-8 string view of a filesystem path which can use a polymorphic source of data e.g. ASCII, UTF-8, UTF-16.

There is an optional dependency on two core language enhancements:

1. [P0709] *Zero-overhead deterministic exceptions* `https://wg21.link/P0709`.

   This proposes that the C++ language implements the lightweight throwing of error/status codes similar to that implemented by Boost.Outcome[1].

2. PEEEE *SG14 Relaxing TriviallyCopyable into Relocatable and TriviallyRelocatable* `https://wg21.link/PEEEE`.

   This proposes two new categories of types which enables more aggressive optimisation:

   - `Relocatable`

     This is a type whose address in memory can be arbitrarily changed without performing a move nor copy. This enables constant time capacity expansion, instead of linear time e.g. a `vector` of `Relocatable` types could use `mremap()` instead of having to allocate new memory, move the contents of the vector, deallocate the old memory.

   - `TriviallyRelocatable`

     This is a type whose move constructor and move assignment is implemented such that the destination is as-if a `memcpy` of the source, and that the source is left as-if constexpr default constructed. Such types can be optimised as aggressively as `TriviallyCopyable` types currently can be, despite potentially having a non-trivial destructor. Lots of types currently in the standard could benefit from this e.g. `std::unique_ptr`, `std::shared_ptr`.

     All the `handle` refinements proposed in this library deliberately meet `TriviallyRelocatable` and thus if this proposal were accepted, could be transported using CPU registers.

There *may* be a dependency on [P0443] *A Unified Executors Proposal for C++* depending on design decisions not yet taken (see below).

The low level file i/o library generally works with `span<T>` or `span<span<T>>`, and thus should automatically work well with Ranges.

4 Proposed Design

The design is very straightforward and intuitive, if you are familiar with low level i/o. We do not innovate in this proposed design. It is more, or less, a straight thin wrap of a subset of the POSIX file i/o specification, as it was standardised in the POSIX.1-2008 specification (ten years ago was chosen as it has wide implementation conformance).

There is a fundamental type called `native_handle_type` which is a simple, *unmanaged* union storage of one of a POSIX file descriptor, or a Windows `HANDLE`. Any other platform-specific resource identifier types would be added here.

`native_handle_type` contains *disposition* about the identifier, specifically what kind it is, what rights it has, is it seekable, does it require aligned i/o, must it be spoken to in overlapped and so on. It can be made invalid i.e. it has a formal invalid state. It is all-constexpr.

At the base of the inheritance hierarchy is the polymorphic `class handle`. It manages a `native_handle_type`, which can be released from its handle if wished. When the handle is destructed, the `native_handle_type` inside the instance is closed.

`class handle` is a move-only type. It does provide a polymorphic clone member function which will duplicate the handle. The reason that the C++ copy constructor is disabled is because duplicating handles is expensive, and unintentionally doing so would be bad.

Apart from releasing, cloning and closing, the only other thing one can do with a handle is to retrieve its current path on the filesystem. It is very important to understand that this is **not** the path it was opened with (if the user wants that, they can cache it themselves). Rather it is what the kernel *says* is the current path for this inode right now[5]. This can be useful to know, as other processes can arbitrarily change the path of lots of open files simply by changing the name of a directory further up the hierarchy. In fact, handle has entirely trivial storage as it stores nothing which is allocated from memory, it can thus be constexpr constructed and is trivially relocatable[6].

Handle defines many types and bitfields used by its refinements:

- `mode`

  This selects what kind of i/o we wish to do with a handle. One of none, attribute read, attribute write, read, write, (atomic) append.

- `creation`

  This selects what opening a handle ought to do if the path specified already exists or doesn't exist. One of open existing, only if not exist, if needed, (atomic) truncate.

- `caching`

  This selects what kind of caching (buffering) the kernel ought to perform for this handle:

  – No caching whatsoever, and additionally `fsync()` the file and any other resources[7] at certain key moments to ensure recovery after sudden power loss (immediately after creation, immediately after maximum extent change, immediately after close). On many, but not all, platforms this is direct DMA to the device from user space which comes with a list of special use requirements (see later in paper).

[5] A standard API for this is not present in POSIX.1-2008, but proprietary APIs are available on all the major platforms and most of the minor ones, including embedded operating systems. For those few systems without kernel support, we provide a templated adapter for all handle types which caches the path for you.

[6] This is a concept which doesn't exist in the language yet, see [PEEEE] for its proposal paper.

[7] On Linux ext4, one must also sync the parent directory as well as the inode to ensure complete recovery after power loss.

  - Cache only metadata. On many, but not all, platforms this is direct DMA to the device from user space.
  - Cache only reads, and with `fsync()` at key moments described above. Writes block until they and the metadata to retrieve them after power loss fully reach storage.
  - Cache reads and metadata, and `fsync()` at key moments described above. Writes block until they fully reach storage.
  - Cache reads, writes, and metadata (the default). Writes are enqueued and written to storage at some later point asynchronously.
  - Cache reads, writes, and metadata, and `fsync()` at key moments described above.
  - Avoid writing to storage as much as possible. Useful for temporary files.

  For those not familiar with data synchronisation outside of `fsync()`, explicitly disabling some or all of kernel caching at handle open results in much better performance than following every write with a `fsync()`. Indeed, in some filing systems like ZFS, a special fast non-volatile device is used to complete an uncached write immediately, which is synced later to slow non-volatile storage.

- `flags`

  This selects various bespoke behaviours and semantics:

  - `unlink_on_close`

    Causes the entry in the filesystem to disappear on first close by any process in the system.

    Microsoft Windows partially implements this in its kernel, and significantly changes how it caches data based on the setting of this flag.

  - `disable_safety_fsyncs`

    Disables the safety `fsync()`'s for the modes listed above.

  - `disable_safety_unlinks`

    Do not compare inode and device with that of the open file descriptor before unlinking it.

  - `disable_prefetching`

    Most kernels prefetch data into the kernel cache after an i/o. For truly random i/o workloads, this flag ought to be set.

  - `maximum_prefetching`

    If we are copying a file's contents using caching i/o, this flag ought to be set.

  - `win_disable_unlink_emulation`

    On Microsoft Windows, POSIX unlink semantics are emulated by renaming on unlink the file entry to something very random such that it cannot be found[8,9]. Setting this flag disables this emulation.

  - `win_disable_sparse_file_creation`

    Microsoft's NTFS file system was designed in the 1980s back when extents-based filing systems were not common. It was later upgraded to an extents-based implementation capable of working with sparse files. Due to backwards compatibility, during file creation one must *opt-in* to using extents-based storage. That setting remains attached to that file for the remainder of its life, which could theoretically break some programs. The proposed library always opts in to extents based storage by default for newly created files to match semantics with almost every modern filing system elsewhere. This flag disables that default opt-in.

4.1 Class hierarchy inheriting from `handle`

Inheriting from `class handle` are these refinements of handle:

- `io_handle`

  I/O handle adds types and member functions for scatter-gather synchronous i/o to a seekable handle[10].

  All i/o is optionally deadline based, with a choice of interval or absolute timeout. Deadline i/o for files only works if the most derived implementation is `async_file_handle` as these synchronous calls are implemented using an asynchronous implementation which can be cancelled.

  I/O handle also adds member functions for mutually excluding part, or all of, the resource represented by the handle from any other process in the system. These are always *advisory* not mandatory exclusions i.e. they require all processes to cooperate by checking for locks before an i/o.

  Inheriting from `io_handle` are these refinements of i/o handle:

  - `file_handle`

    File handle is the simple, unfussy thin wrap of the platform's file read and write facilities. All i/o is always performed via the appropriate syscall. This passes through any POSIX

[8] Due to VMS legacy compatibility, NT implements file deletion by marking a file entry as deleted which prevents it being opened for access thenceforth. It does not remove the file entry until some arbitrary time (usually milliseconds) after the last open handle to it in the system has closed. This confounds code written to expect POSIX semantics whereby unlinking a file causes it to immediately disappear from the filesystem. This workaround of renaming the file to something very random simulates, incompletely, POSIX semantics on Microsoft Windows, sufficiently so at least that most filesystem algorithms 'just work'.

[9] I have been told by Microsoft that the next version of Windows 10 implement opt-in POSIX unlink semantics, so on newer Windows we can avoid rename-to-random tricks.

[10] Non-seekable handles are valid, but that would start to overlap the Networking TS. For various technical reasons, asynchronous socket and pipe i/o cannot portably use the same i/o service implementation as asynchronous file i/o, this is why this proposed library is orthogonal to the Networking TS.

read-write atomicity and sequential consistency guarantees which may be implemented by the platform.

File handles provide the following additional static member functions:

- For creating and opening a named file using a `path_handle` instance as the base (a default constructed `path_handle` instance requires the path view to refer to an absolute path).
- For creating a cryptographically randomly named file at a location specified by a `path_handle` instance. This is useful for creating a temporary file which once fully written to, will be atomically renamed to replace an existing file.
- For creating a temporary file in one of the temporary file locations found during path discovery (see `path_discovery` below), counted against user quota or system RAM quota.
- For securely creating an anonymous temporary inode at a location specified by a `path_handle` instance. These are always unnamed, always inaccessible inodes which do not survive process exit. These are used especially by generic template algorithms to implement novel STL containers like vectors with constant, rather than linear, capacity expansion times.

File handles provide the following additional polymorphic member functions:

- For getting and setting the maximum file extent (not 'the length', though many people get confused on this).
- For issuing a write reordering barrier which can be optionally applied to a subset of extents in the file, optionally with blocking until preceding writes reach storage, and optionally with an additional flush of inode metadata which indicates current maximum extent, timestamps etc.
- For enumerating the valid extents in the file. Modern extents-based filing systems (pretty much all in common use today except for FAT) only store the extents written to, so a 1Tb maximum extent file might only have 4Kb of extents allocated within it. Colloquially known as 'sparse files'.
- For deallocating a valid extent in the file. Colloquially known as 'hole punching'.
- For unlinking the hard link currently referred to by the open handle.
- For relinking the hard link currently referred to by the open handle to another path, optionally atomically replacing any item currently at that path.
- For creating a new hard link to the inode referred to by the open handle at a new path location.

Note that one can instance any refinement of `file_handle` implementation and pass it to functions as if it were a true `file_handle`. Under the bonnet, scatter-gather synchronous i/o is implemented as whatever is the most optimal for that implementation type e.g. for `mapped_file_handle` scatter-gather synchronous i/o is implemented with `memcpy()`.

Inheriting from `file_handle` are these refinements of file handle:

- `async_file_handle`

  The async file handle can behave in every way as if a synchronous file handle i.e. the member functions inherited from `io_handle` behave as if synchronous, though unlike in other implementations, they can observe timeouts.

  It adds member functions for scatter-gather asynchronous i/o taking a completion callback (`async_read()`, `async_write()`). Instantiating an async file handle requires the user to supply an instance of `io_service` to issue callback completions against, this must be pumped for completion dispatch very similarly to the `io_service` in the Networking TS.

  Async file handle also provides member functions for coroutinised i/o (`co_read()`, `co_write()`) whereby the calling coroutine is suspended until the i/o completes, whereupon it is resumed.

- `mapped_file_handle`

  The mapped file handle is the most highly performing file handle implementation in terms of i/o, but comes with significantly higher cost construction, extension and destruction and with severe usability limits on 32 bit architectures. It also loses any POSIX read-write atomicity and sequential consistency guarantees which may be implemented by the platform on the other types of handle.

  It always maps the whole file into memory, extending the map as needed into an *address reservation*. Unless you are opening and closing files frequently, or the files you are working with are much smaller than the system page size, or you are on a 32 bit architecture, this is an excellent default choice for most users giving maximum zero whole system memory copy performance on all devices apart from network attached storage devices.

– `map_handle`

Map handle is a region of shared or private memory mapped from a backing `section_handle`, or unmapped private memory backed by the swap file, or reserved address space. Within the committed (i.e. allocated) part of that region, i/o can be performed, or more usefully, the region can be accessed directly as memory.

Added member functions include the ability to commit (allocate) sub-regions of reserved address space, or to decommit (deallocate) previously allocated sub-regions.

It comes with a comprehensive set of static member functions which can be applied to any memory in a process e.g. ‘please kick the contents of this memory page out to backing storage’, ‘please unset the dirty bit of this memory page (i.e. don’t flush its contents to storage until the next modification)’, or ‘please asynchronously ready this range of memory for access (i.e. prefault it)’ and so on.

`mapped_file_handle` and many other classes use this class as an internal implementation primitive for all forms of mapped and unmapped and reserved memory.

- `path_handle`

  Path handles refer to some base location on the filesystem from which path lookup begins. The inode opened may change its path arbitrarily and at any time without affecting the paths which use an open path handle as their base. This handle is, therefore, the foundation of the race free filesystem which the proposed library implements.

  Many platforms implement the creation of these handles as an especially lightweight operation, hence they are standalone from `directory_handle`.

  Inheriting from `path_handle` are these refinements of path handle:

  - `directory_handle`

    Directory handles refer to inodes which list other inodes. The main added member function is to enumerate that list of other inodes into a user supplied array (`span`) of `directory_entry`. One can open existing directories, create new directories, create randomly named new directories, and in your choice of path including temporary paths found during path discovery. One can of course also unlink and relink directories.

- `section_handle`

  Section handles refer to a section of shared or private memory. They may be backed by a user supplied `file_handle`, or by an anonymous inode in one of the path categories returned by `path_discovery`, or by some other source of shared memory. They are particularly useful for when you need some temporary storage (counted against either the RAM quota or the current user's quota) which will be thrown away at process end.

  Section handles have a length which can be queried and changed. It may be less than, but cannot exceed, the maximum extent of any backing file.

  Section handles have additional flags in addition to those inherited from `handle`. Section handle flags are reused by `map_handle`:

  - `none`: This memory region is reserved address space.
  - `read`: This memory region can be read.
  - `write`: This memory region can be written.
  - `cow`: This memory region is copy-on-write (i.e. when you first write, the kernel makes you a process-local copy of the page).
  - `execute`: This memory region can contain code which the CPU will execute.
  - `nocommit`: Don't immediately allocate resources for this section/memory region upon construction. Most kernels allocate space for unbacked sections against the system memory + swap files, and will refuse new allocations once some limit is reached. Setting this flag causes unbacked sections to allocate system resources 'as you go' i.e. as you explicitly commit pages using the appropriate member functions of `map_handle`.
  - `prefault`: Prefault, as if by reading every page, any views of memory upon creation. This eliminates first-page-access latencies where on first access, the page is faulted into

existence.

  - `executable`: This section represents an executable binary.
  - `singleton`: A single instance of this section is to be shared by all processes using the same backing file. This means that when one process changes the section's length, all other processes are instantly updated (with appropriate updates of maps of the section) at the same time, which can be considerably more efficient.
  - `barrier_on_close`: Maps of this section, if writable, issue a blocking `barrier()` when destructed, blocking until data (not metadata) reaches physical storage.

- `symlink_handle`

  Symlink handles refer to inodes which contain a relative or absolute path. Added member functions can read and write that stored path.

4.2 Miscellaneous and utility classes and functions

There are also some utility classes:

- `deadline`

  A deadline is a standard layout and trivially copyable type which specifies either an interval or absolute deadline. Deadlines can construct from any arbitrary `std::chrono::duration<>` or `std::chrono::time_point<>`. The advantage to this object is halving the number of polymorphic function overloads required.

- `directory_entry`

  A `path_view` and `stat_t` combination. Filled by `directory_handle`'s enumeration function. Note that it has standard layout and is trivially copyable.

- `io_service`

  A completion handler dispatcher used by `async_file_handle`. Looks very similar to the Networking TS's `io_service`, but must be distinct as asynchronous file i/o cannot be portably implemented using the same i/o service as pipe and socket i/o.

- `path_discovery`

  Path discovery generally runs once per process and it interrogates the platform to discover suitable paths for (i) storage backed temporary files (counted against the current user's quota) and (ii) memory backed temporary files (counted against available RAM). Path discovery does not trust the platform specific APIs, and it tries creating a file in each of the directories reported by the platform to find out which are valid. This is slow, so the results are statically cached.

- `path_view`

Path views are covered in detail in [PDDDD], but in essence they are a polymorphic view of a string. The view is presented by default in UTF-8, but its underlying storage could be anything (usually UTF-8 or UTF-16 depending on platform).

The choice of polymorphic underlying storage will be controversial, however I can assure readers that it is the correct design choice given the exigencies. Path views have different design exigencies to `std::filesystem::path`, and therefore should have a different design. The proposed low level file i/o library almost exclusively uses path views throughout, which can avoid copying up to 64Kb of path per syscall, plus avoids memory allocation and makes direct use of string literals.

- `stat_t`

  Almost certainly WG21 will want the name needs to be changed to avoid conflict with the platform `stat_t`, but I haven't personally found it to be an issue in practice. This is a C++-ified `struct stat_t`, it uses `std::filesystem` constants and data types instead of the platform-specific ones.

- `statfs_t`

  Similarly, almost certainly WG21 will want the name needs to be changed to avoid conflict with the platform `statfs_t`, but I haven't personally found it to be an issue in practice. This is a C++-ified `struct statfs_t`, it uses `std::filesystem` constants and data types instead of the platform-specific ones. Unusually for types in the proposed library, this one is not trivially copyable as it contains two `std::string`'s and a `std::filesystem::path` for the `f_fstypename`, `f_mntfromname` and `f_mntonname` members.

There are some minor utility functions as well which are not described in detail for now. They fetch things like the TLB page size entries for this machine, have the kernel return single TLB entry allocations of varying sizes either via a C malloc type API or via a special STL allocator, ask the kernel to fill a buffer with cryptographically strong random data, fast to-hex and from-hex routines and so on.

4.3 Generic filesystem algorithms and template classes already reference implemented

These are some generic algorithms and template classes which act as primitives for more complex filesystem algorithms. It should be stressed that all of the below are 100% header only code, and use **no** platform-specific APIs. They are implemented **exclusively** using the public APIs in the proposed low level file i/o library. This may give an idea of the expressive power to build useful and interesting filesystem algorithms using the proposed design.

- `shared_fs_mutex`

  This is an abstract base class for a family of shared filing system mutexs i.e. a suite of algorithms for excluding other processes and threads from execution using the filesystem as the interprocess communication mechanism.

Unlike memory-based mutexes already in the standard library, in the lock operation these mutexes take a sequence of *entities* upon which to take a shared or exclusive lock. An entity is a 63 bit number (the top bit stores whether it is exclusive or not)[11].

The reason that these mutexes are list-of-entities based is because it is very common to lock more than one thing concurrently on the filing system, whereas with memory-based mutexes that is the exception rather than the norm. For example, if you were updating file number 2 and file number 10 in a list of files at the same time, you would concurrently lock entities 2 and 10. If you were implementing a content addressable database like a git store, you'd use the last 63 bits of the git SHA as the entity, and so on.

Each of the implementations has varying benefits and tradeoffs, including the ability to lock many entities in the same time as one entity. The appropriate choice depends on use case, and to an extent, the platform upon which the code is running.

– `shared_fs_mutex::atomic_append`

  This implementation uses an atomically appended shared file as the IPC mechanism. Advantages include invariance to number of entities locked at a time, ability to sleep the CPU and compatibility with all forms of storage except NFS. Disadvantages include an intolerance to one of the using processes experiencing sudden process exit during lock hold, and filling all available free space on filing systems which are not extents based (i.e. incapable of 'hole punching').

– `shared_fs_mutex::byte_ranges`

  This implementation uses the byte range locks feature of your platform as the IPC mechanism. Advantages include ability to sleep the CPU and automatic handling of sudden process using during lock hold. Disadvantages include wildly differing performance and scalability between platforms, lack of thread compatibility with POSIX implementations other than recent Linux, ability to crash NFS in the kernel due to overload.

– `shared_fs_mutex::lock_files`

  This implementation uses exclusively created lock files as the IPC mechanism. Advantages include simplicity and wide compatibility without corner case quirks on some platforms. Disadvantages include an inability to sleep the CPU, and an intolerance to one of the using processes experiencing sudden process exit during lock hold.

– `shared_fs_mutex::memory_map`

  This implementation uses a shared memory region as the IPC mechanism. Advantages include blazing performance to the extent of making your mouse pointer stutter. Disadvantages include inability to use networked storage, inability to sleep the CPU, and an intolerance to one of the using processes experiencing sudden process exit during lock hold.

– `shared_fs_mutex::safe_byte_ranges`

[11] This design choice works around the problem that on some platforms, byte range locks are *signed* values, and attempting to take a lock on a top bit set extent will thus always fail.

This implementation – on POSIX only – wraps the byte range locks on the platform with a thread locking layer such that individual threads do not overwrite the locks of other threads within the same process, as is required by the POSIX standard for byte range locks. On other platforms, this is a typedef to `shared_fs_mutex::byte_ranges`.

- `cached_parent_handle_adapter<T>`

  Ordinarily, handles do not store any reference to their parent inode. They provide a member function which will obtain a such a handle by fetching the current path of the inode and looping the check to see if it has a leaf with the same inode and device number as the handle. This, obviously enough, is expensive to call.

  For use cases where a lot of race free sibling and parent operations occur, one can instantiate any of the handle types using this adapter. It overrides some of the virtual functions to use a cached parent inode implementation instead. These parent inode handles are kept in a global registry, and are reference counted to minimise duplication. This very considerably improves the performance of race free sibling and parent operations, at the cost of increasing the use of file descriptors, plus synchronising all threads on accessing the global registry.

  There is an additional use case, and that is where the platform does not implement file inode path discovery reliably, which can afflict some older editions of some kernels [12].

- `mapped_span<T>`

  A mapped span is simply a coloured `span<T>` of a `map_handle`'s region. It implies a `reinterpret_cast<T>` of the `map_handle`'s `char` mapped memory.

  Coloured spans is covered by [PAAAA]. In essence, it is a `span<T>` with attributes. In this specific case, the span is coloured with attributes to say that it is (i) aligned to one of the system page sizes (ii) is a multiple of one of the system page sizes (iii) this is a region which will be DMAed in the future (and so please use non-temporal stores where appropriate) and (iv) whether the 'full' `barrier()` or 'light' `barrier()` is sufficient to synchronise writes to backing storage.

  Mapped spans allow one to easily adapt sparse storage into a sparsely stored array:

```
namespace afio = AFIO_V2_NAMESPACE;

// Make me a 1 trillion element sparsely allocated integer array!
afio::mapped_file_handle mfh = afio::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));

// Create a typed span of the one trillion integers
afio::algorithm::mapped_span<int> one_trillion_int_array(mfh);

```

[12] At the time of writing, OS X's path fetching API returns one of the paths for any hard link to the inode, randomly. This is almost certainly a bug. FreeBSD does not reliably provide path fetching for file inodes, but does for directory inodes. From examination of the kernel source, this ought to be easy to fix. In both cases, fetching the path of a directory inode is reliable, and thus via this adapter works around these platform-specific quirks and bugs.

```
// Write and read as you see fit, if you exceed physical RAM it'll be paged out
one_trillion_int_array[0] = 5;
one_trillion_int_array[999999999999ULL] = 6;
```

Virtual memory based kernels have been able to do this sort of stuff for years, but making use of it, especially portably, was tedious and error prone. The above shows how much easier this sort of programming becomes.

4.4 Generic filesystem algorithms yet to be reference implemented

- Persistent page allocator which is interruption safe, concurrency safe, lock free. This is effectively a persistent linked-list implementation of allocated and non-allocated regions within the file.
- The aforementioned B+ tree implementation[2] which is interruption safe, concurrency safe, lock free.
- Persistent vector which is interruption safe, concurrency safe, lock free.
- Coroutine generators for valid, or all, file extents.
- Compare two directory enumerations for differences (Ranges based).
- B+-tree friendly[13] directory hierarchy deletion algorithm.
- B+-tree friendly directory hierarchy copy algorithm.
- B+-tree friendly directory hierarchy update (two and three way) algorithm.

4.5 Functionality whose design is blocked on undecided features at WG21

Most of the just listed items are tricky to implement until compilers and standard libraries implement Coroutines and Ranges without quality of implementation problems, hence why they have not been reference implemented yet.

Some, however, are blocked on WG21. In particular, the directory algorithms need to be available to multiple kernel threads as the filesystem tends to scale linearly with CPU cores if poked at right i.e. the algorithm would sometimes choose to execute a list of operations in parallel knowing that this filing system will scale for this operation, but execute another list sequentially knowing that that will scale better, and this would need to be dynamically determined inside the algorithm's execution tree.

In theory, the [P0443] Executors proposal should fit the bill. Perhaps it is my ignorance of the proposed design, but it seems too 'heavy' for the kind of micro-operations that the directory algorithms would do. In particular, it appears to require allocating memory for every task executed, and the future-based completion notification mechanism which implies another memory allocation,

[13] By 'B+-tree friendly', I mean that the algorithm orders its operations to avoid the filesystem's B+-tree rebalancing frequently, as a linear algorithm which almost everybody writes without thinking will do. This can improve performance by around 20% on the major filing systems.

plus atomic reference counting, is hardly lightweight. An alternative is something based on the Parallelism TS, but that would suffer from being too much a ‘one size fits all’ approach.

My ideal solution would in fact be completely agnostic to the concurrency mechanism employed, so the user decides. But designing such an implementation with so many as yet undecided design choices at WG21 is hard, and it will get easier if I simply kick the can down the road. Which is what I have done until now.

If WG21 agrees to set up a Study Group in this topic, that would be an excellent place to bounce around some ideas on how best to implement these remaining generic filesystem algorithms.

4.6 Filesystem functionality deliberately omitted from this proposal

The eagle eyed will have spotted entire tracts of the filesystem have been omitted from this initial proposal:

- Permissions

  Standardising this is a ton of extra work best pushed, in my opinion, into a later standardisation effort.

- Extended attributes

  These probably could be standardised without much effort, but I am also unsure of the demand from the user base. Despite almost universal support in file systems nowadays, they are not exactly commonly used outside of MacOS.

- Directory change monitoring

  This is surprisingly hard to implement correctly. Imagine writing an implementation which scales up to 10M item directories and never misrepresents a change? The demands on handling race conditions correctly are very detailed and tricky to get right in a performant and portable way. I would like the change delta algorithms decided upon before tackling this one.

5 Design decisions, guidelines and rationale

The design decisions are as follows, in priority:

5.1 Race free filesystem

As anyone familiar with programming the filesystem is aware, it is riddled with race conditions because most code is designed assuming that the filesystem will not be changed by third parties during a sequence of operations. Yet, not only can the filesystem permute at any time, it is also a bountiful source of unintended data loss and security exploits via Time-of-check-Time-of-use (TOCTOU) failures.

As an example, imagine the following sequence of code which creates an anonymous inode to temporarily hold data which will be thrown away on the close of the file descriptor, perhaps to pass to a child process or something:

```
int fd = ::open("/home/ned/db/foo", O_RDWR|O_CREAT|O_EXCL, S_IWUSR);
::unlink("/home/ned/db/foo");
::write(fd, child_data, ...);
```

Imagine that privileged code is executing that code. Now witness this:

```
int fd = ::open("/home/ned/db/foo",
  O_RDWR|O_CREAT|O_EXCL, S_IWUSR);


::unlink("/home/ned/db/foo");  // oh dear!
```

```


::rename("/home/ned/db", "/home/ned/db.prev");
::symlink("/etc", "/home/ned/db");
```

We have just seen unintended data loss where `/etc/foo` is unlinked instead of the programmer intended `/home/ned/db/foo`.

Here is another common race on the filesystem:

```
int storefd = ::open("/home/ned/db/store",
  O_RDWR);


int indexfd = ::open("/home/ned/db/index",
  O_RDWR);
```

```


::rename("/home/ned/db", "/home/ned/db.prev");
::rename("/home/ned/db.other", "/home/ned/db")
    ;
```

Now the index opened is not the correct index file for the store file. Misoperation and potential data corruption is likely.

POSIX.1-2008, and every major operating system currently in use, fixes this via a *race free* filesystem API. Here are safe implementations:

```
int dirh = ::open("/home/ned/db", O_RDONLY|O_DIRECTORY);
int fd = ::openat(dirh, "foo", O_RDWR|O_CREAT|O_EXCL, S_IWUSR);
::unlinkat(dirh, "foo", 0);
```

```
int dirh = ::open("/home/ned/db", O_RDONLY|O_DIRECTORY);
int storefd = ::openat(dirh, "store", O_RDWR);
int indexfd = ::openat(dirh, "index", O_RDWR);
```

The proposed low level file i/o library considers race free filesystem to be sufficiently important that it is enabled by default i.e. it is always on unless you explicitly ask for it to be off. The natural question will be 'How expensive is this design choice?'.

These are figures for the reference library implementation running on various operating systems and filing systems. They were performed with a fully warm cache i.e. entirely from kernel memory without accessing the device. They therefore represent a **worst case** overhead.

	FreeBSD ZFS	Linux ext4	Win10 NTFS
Delete File:	6.2%	11.6%	0%

The extra cost on POSIX for deletion is due to opening the inode's parent directory, checking that a leaf item with the same name as the file to be unlinked has the same inode and device as that of the open handle, and if so then unlinking the leaf in that directory. This algorithm makes file deletion impervious to concurrent third party changes in the path, up to the containing directory, during the deletion operation. A similar algorithm is used for renames, and added overhead is typically around 10%.

One will surely note that overhead on Microsoft Windows is zero. The is because the NT kernel provides much more extensive a race free filesystem API than POSIX does. In particular, it provides a by-open-file-handle API for deletion and renaming so one need not implement any additional work to achieve race freedom.

I appreciate that the choice to make race free filesystem opt-out rather than opt-in will be a controversial one on the committee, not least due to implementation concerns on the less major kernels[14]. However it is my belief that correctness trumps performance for the default case, and for those users who want the fastest possible filesystem performance, race free filesystem can be disabled per object in the constructor.

5.2 No (direct) support for kernel threads

For those coming from a Networking TS/ASIO background, the choice to not support kernel threads in the proposed library's `io_service` seems stunning. I know this from the more than one bug report filed against the reference library implementation over the years: there is a widespread belief that asynchronous i/o *ought* to support kernel threads. I therefore need to explain why the proposed library does not.

Kernels implement synchronous i/o by enqueuing a request to the hardware on the same CPU as the thread which initiated the i/o. When the hardware completes the i/o, it raises an interrupt on that CPU, and the kernel resumes the thread. Asynchronous i/o is little different, except that the initiating thread continues immediately and is not suspended until the hardware raises an interrupt, rather usually some form of signal or notification is posted to the initiating thread for collection later when that thread is ready to execute completions.

There is a second variant of asynchronous i/o however, whereby the interrupt is directed to the next currently idle CPU within a pool of threads attached to completing the i/o. It is this second variant which ASIO uses, whereas this proposed library uses the first variant.

Threaded asynchronous i/o was without doubt much faster fifteen years ago, back when ASIO was designed. But kernels have improved greatly since then, sufficiently so that *asynchronous i/o is usually slower than synchronous i/o* because the kernel must do more work to schedule an asynchronous i/o (specifically, it must almost always allocate some memory). In addition, kernel threads damage locality of CPU cache utilisation which has become much more important on today's

[14] See the description of `cached_parent_handle_adapter<T>` above. However I believe that kernel maintainers are highly amenable to adding a syscall to unlink-by-fd or relink-by-fd, they just need to be given a business case for it. It certainly is trivially easy to implement in any of the kernel sources I have investigated.

CPUs than those of fifteen years ago. If you're accessing any memory other than the i/o, increasingly you want that hot in the CPU cache of the kernel thread implementing the i/o completion, a good bet for which usually is the kernel thread which initiated that i/o.

Many latency-sensitive users of ASIO therefore end up running an `io_service` instance per CPU pinned kernel thread in order to better control cache locality. However, ASIO still is employing, under the bonnet, all the machinery to support multiple threads, and that may make it run less efficiently than it might otherwise. I will not say more about ASIO on this, but for file i/o this author benchmarked a wide variety of file i/o patterns using blocking i/o, alertable i/o (complete i/o to the initiating thread) and IOCP (complete i/o to the next idle CPU) on Microsoft Windows, and found that alertable i/o bested IOCP in every way for file i/o. Latency variance was an order of magnitude lower. No mutexs were required. Implementation was considerably more simple. This is why the proposed low level file i/o library does not support multiple kernel threads.

Two points are important to understand however. The first is that it is straightforward to build a pool of threads running file i/o services using this proposed library, and to distribute i/o work across them, if that is what you want. The second is that C++ Coroutines work very well with this library, you simply write code such as:

```
namespace afio = AFIO_V2_NAMESPACE;

// Create an asynchronous file handle
afio::io_service service;
auto fh = afio::async_file(service, {}, "testfile.txt",
                           afio::async_file_handle::mode::write,
                           afio::async_file_handle::creation::if_needed).value();

// Resize it to 1024 bytes
truncate(fh, 1024).value();

...

// Begin to asynchronously write "hello world" into the file at offset 0,
// suspending execution of this coroutine until completion and then resuming
// execution. Requires the Coroutines TS.
alignas(4096) char buffer[] = "hello world";
co_await co_write(fh, {{{buffer, sizeof(buffer)}}, 0}).value();
```

This works exactly as one would expect: coroutines initiate i/o which suspends the coroutine until it completes. In the meantime, other coroutines on the same kernel thread execute if they are ready to be resumed.

5.3 Asynchronous file i/o is much less important than synchronous file i/o

A theme running throughout this proposal paper is that asynchronous file i/o is usually not worth the extra CPU cost on recent kernels of the major operating systems, and hence the proposed low level file i/o library mostly speaks of synchronous, not asynchronous, file i/o. I have noticed that some on WG21 are very keen on bringing complex coroutinised asynchronous i/o frameworks similar to WinRT to the C++ standard soon – indeed, these papers from me are being submitted now in order to preempt misguided papers from those others.

Firstly, of the major operating systems, the only one to actually implement asynchronous file i/o on buffered (cached) files is Microsoft Windows. Linux, FreeBSD, and MacOS all use userspace or kernel threadpools to *emulate* asynchronous i/o, if the handle is buffered. And with Microsoft Windows it definitely is an order of magnitude latency variance penalty to use IOCP to complete asynchronous i/o, only alertable i/o has a reasonable variance, and that still is markedly worse than straight synchronous i/o. So let me be clear on this: *empirical testing suggests that you are almost always worse off employing asynchronous i/o on buffered files on all platforms* because asynchronous i/o is always more work for the CPU to complete, and half the time, even with random i/o, intelligent prefetching by the kernel page cache will be able to complete the i/o immediately in any case, making the asynchronous ceremony a waste of CPU time.

All of the major operating systems do implement true asynchronous file i/o if the handle is *unbuffered*, though rarely by the POSIX asynchronous file i/o API as it scales poorly to queue depth. Unbuffered i/o generally requires all i/o to be performed on native device sector alignment boundaries and multiples: 4Kb is a widely portable choice with today's storage devices. It is therefore unsuitable for WinRT-style general purpose coroutinised asynchronous i/o frameworks. It is quite hard for the typical developer to write an unbuffered filesystem algorithm which significantly outperforms buffered i/o[15]. Countless millions of hours by the best filesystem engineers in the world have been invested on tuning buffered i/o to perform excellently under a very wide range of use cases.

There are important use cases for unbuffered i/o, especially in the bulk copying of files to avoid evicting the current kernel page cache. This proposed library hence has excellent support for unbuffered i/o. There are few cases where asynchronous file i/o makes sense over synchronous i/o. If you really need to multiplex i/o, you are far better off using a userspace pool of threads doing synchronous i/o, ideally to memory maps to get true zero copy. It parallelises much better, scales much better, does both consistently across all the major platforms, and makes great use of the kernel page cache.

All this could be seen as an argument against this proposed library supplying asynchronous i/o at all. I would not oppose that conclusion, it is a rational one based on empirical fact. However I can also see that there is a quality of implementation opportunity in here: an implementation could use a pool of Executors to emulate asynchronous i/o in order for coroutinised i/o to be written to multiplex i/o on a single kernel thread. There is also a desirable impact on discipline: even if one never standardises the asynchronous file i/o part of this proposal, having it in here forces one to make a better designed, more extensible, more customisable library in my opinion.

6 Technical specifications

It is proposed that due to its size, complexity and relative independence from other parts of the C++ standard library, that the low level file i/o library be formulated in a Technical Specification under a new Persistent Data and Algorithms study group [PXXXX].

[15] This is not to say that one should not *manage* the kernel page cache by proactively evicting and hinting pages where it makes sense to do so. The proposed library has a comprehensive suite of static member functions for doing this.

7 Frequently asked questions

7.1 Why bother with a low level file i/o library when calling the kernel syscalls directly is perfectly fine?

1. This low level file i/o library defines *a common language* of basic operations across platforms. In other words, it chooses a common denominator across 99% of platforms out there. If you append to a memory mapped file, that'll do the platform-specific magic on all supported platforms.
2. This low level file i/o library only consumes and produces trivially copyable, trivially relocatable and standard layout objects. Empirical testing has found that the optimiser will eliminate this low level library almost always, inlining the platform specific syscall directly. So, it is *no worse* in any way over calling the platform syscalls directly, except that this library API is portable.
3. Where trivial to do so, we encode domain specific knowledge about platform specific quirks. For example, `fsync()` on MacOS does not do a blocking write barrier, so our `barrier()` function calls the appropriate magic `fcntl()` on MacOS only where the `barrier()` is requested to block until completion.
4. This low level file i/o library is a bunch of primitives which can be readily combined together to build filesystem algorithms whose implementation code is much cleaner looking and easier to rationalise about than using syscalls directly.
5. We can provide deep integration with C++ language features in a way which platform specific syscalls cannot. Ranges, Coroutines and Generators are the obvious examples, but we also make a ton of use of `span<T>`, so all code which understands `span<T>` – or more likely the `std::begin()` and `std::end()` overloads it provides – automagically works with no extra boilerplate needed.

7.2 The filesystem has a reputation for being riddled with unpredictable semantics and behaviours. How can it be possible to usefully standardise anything in such a world?

That is a very good question. This proposal *passes through*, for the most part, whatever the platform syscalls do. If, for example, `read()` and `write()` implement the POSIX file i/o atomicity guarantees, then:

1. A write syscall's effects will either be wholly visible to concurrent reads, or not at all (i.e. no 'torn writes').
2. Reads of a file offset *acquire* that offset, writes to a file offset *release* that offset. Acquire and release have the same meaning as for atomic acquire and release, so they enforce a sequential ordering based on overlapping regions[16].

[16] Many, if not most, filing systems actually implement a RW mutex per inode so their guarantees are rather stronger than POSIX requirements. One should not rely on this in portable code however!

These are very useful guarantees for implementing lock free filesystem algorithms, and are a major reason to use `read()` and `write()` instead memory maps because one can forego using any additional locking. Major platform support for the POSIX read/write atomicity guarantees is pretty good in recent years[17]:

	FreeBSD ZFS	Linux ext4	Win10 NTFS
Buffered i/o	Scatter-gather	No	Per buffer
Unbuffered i/o	Scatter-gather	Scatter-gather	Scatter-gather

As another example, `fsync()` commonly has no effect on the current configured platform[18], and thus `io_handle::barrier()` may do nothing useful. Moreover, code cannot tell if `fsync()` is working or not. Writing portable code which works correctly would therefore seem impossible, but in truth this is a *configuration* issue. Software cannot be expected to predict things outside its control. All it can do is call the correct syscalls at the correct times to prevent unsafe write reordering, and proceed as if those syscalls are working correctly.

So how would I propose that one writes up the pre and post conditions for the functions in this library? I propose to mandate certain minimum behaviour guarantees, and to explicitly list those behaviours which we allow to float. For example, consider the gather write function:

```
io_result<const_buffers_type> io_handle::write(io_request<const_buffers_type> reqs,
                                               deadline d = deadline()) noexcept;
```

[`io_result<T>` is like an Expected which can return either a `T` or an error code of a form similar to `std::error_code`.]

We can write what is required of a conforming implementation:

1. If the handle is not open, or not open for writing, the function will fail by returning an error code which compares equal to `std::errc::bad_file_descriptor`, `std::errc::permission_denied`, or `std::errc::operation_not_permitted`. [The ambiguity is due to platform-specific differences between the major platforms, we never remap error codes returns by syscalls]

2. If the handle implementation does not implement deadline i/o, the function will fail by returning an error code which compares equal to `std::errc::not_supported`.

3. If not opened for append (`handle::mode::append`), an attempt shall be made to write each of the regions of memory specified by the i/o request, consecutively, to the offset within the open file specified by the i/o request's offset member.

4. If opened for append, the open file's maximum extent shall be atomically increased by the size necessary to write all of the gather buffers list, followed by an attempt to write each of the regions of memory specified by the i/o request into this newly allocated extent.

[17] Scatter-gather atomicity means that the entire of a scatter-gather buffer sequence is treated as an atomic unit. Per buffer atomicity means that atomicity is per scatter-gather buffer only.

[18] `fsync()` having no effect is surprisingly common in the real world. First, POSIX permits it to be a no-op, so that is what it is in many cases. Secondly, 'it makes software go slow', so there is huge incentive to partially or wholly disable it. For example, MacOS makes it into a non-blocking write reordering barrier, which is decidedly out of spec. Most LXC containers make it into a no-op to prevent containers implementing a denial of service attack on the other containers.

5. If the time taken to write the regions exceeds the deadline, the remaining write may be cancelled and the function may fail by returning an error code which compares equal to `std::errc::timed_out`.
6. Upon at least partial success (defined by at least one buffer written), the buffers returned shall be updated to reflect the memory regions actually written. These may have different addresses, as well as sizes, to the buffers input.
7. If the system currently does not have the resources to write even one of the buffers supplied, an error code comparing equal to `std::errc::resource_unavailable_try_again` will be returned. If at least part of one buffer is written, this will be considered a partial success and the buffers returned will have their sizes updated to indicate actual data written.
8. If any failure is returned by any of the syscalls called by the implementation which cannot be easily handled at the time, these shall be returned unmodified and unmapped in the error code.

We can write what we knowingly leave up to implementations to define:

- It is implementation defined whether cancellation of i/o completes at any point close to the deadline specified, or at all.
- It is implementation defined whether the system has sufficient resources to issue any more than one of the gather buffers in this operation (some implementations have very limited gather buffer limits which depend on system load, these limits can vary from call to call).
- It is implementation defined whether other processes doing a read of offsets overlapping those being updated will see torn writes (i.e. the write in the process of being applied).
- It is implementation defined whether writes are made visible to other processes, or placed on storage, in the order the program issues them.
- It is implementation defined whether a write which exceeds the file's maximum extent may cause the automatic increase of the file's maximum extent, and whether the change in the file's maximum extent will be propagated to other users of the file[19].
- It is implementation defined whether gather buffers shall be issued to the system one by one, or as a group/batch.
- It is implementation defined whether the current file pointer for the underlying handle shall be affected by the write.

There are some gotchas in the above though. For example, NFS has no wire format method of indicating atomic appends, so append-only files do not append atomically (Samba gets this right). So saying that the function will do atomic appends is clearly not possible for some platform configurations unless the standard library always takes a byte range lock at the end of all files opened for append only on a NFS mount.

[19] Some major platforms do not present changes to maximum extent to other processes until the metadata for the change has reached storage, yet reading off the end of the apparent maximum extent will succeed. The time required for the maximum extent value to propagate to all processes can take some milliseconds sometimes, and can confound algorithms.

And one might think that wise, until one considers what happens if you have a situation where some processes are atomic appending locally and others are atomic appending via NFS. Now for correctness, *all* atomic append files need to take a byte range lock, which rather defeats the purpose of the proposed library exposing kernel support for atomic appends.

My own personal viewpoint on this is to simply consider NFS to have a bug, and to not consider it further. We mandate required semantics in the ISO specification, same as the POSIX specification does. We accept that implementations will have a quality of implementation choice which they themselves can make decisions on based on the platforms they support, and their user bases.

7.3 Why do you consider race free filesystem so important as to impact performance for all code by default, when nobody else is making such claims?

Firstly, performance is only impacted if the host platform does not support direct syscall implementations for all the race free operations exposed by the proposed low level file i/o library, and the missing functionality must be emulated from user space. At least one major platform provides a full set (Microsoft Windows), and I have an enhancement ticket open for Linux[20] to implement the missing support. If WG21 forms the proposed study group, you can be assured that I will try to bang the drum with the OS vendors to add the missing support to their syscalls, indeed I may just go submit a kernel patch to Linux myself (or persuade a Study Group member to do it).

I strongly take the opinion that correctness must precede performance, and as the filesystem is free to be concurrently permuted at any time by third parties, a correct implementation **requires** program code to be as impervious as possible to filesystem race conditions.

I appreciate that many do not share this opinion. A great many ran ext3 as their Linux filing system when it was demonstrably incorrect in a number of important behaviours[21]. Such users preferred maximum performance to losing data occasionally, and I don't mind any individual choosing that for their individual needs.

But international engineering standards must be more conservative. Choices made here affect everybody, including users where data loss must be avoided at all costs. Defaulting to race free filesystem is the safest choice. Without defaulting to race free filesystem, code written using this low level file i/o library would be much less secure, more prone to surprising behaviour, and end users of C++ code exposed to a higher risk of loss of their data.

8 Acknowledgements

Thanks to Nicol Bolas, Bengt Gustaffson and Chris Jefferson for their feedback.

[20] `https://bugzilla.kernel.org/show_bug.cgi?id=93441`

[21] Feel fear after reading `http://danluu.com/file-consistency/`.

9 References

[P0443] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown,
A Unified Executors Proposal for C++
http://wg21.link/P0443

[P0709] Herb Sutter,
Zero-overhead deterministic exceptions
https://wg21.link/P0709

[P0939] B. Dawes, H. Hinnant, B. Stroustrup, D. Vandevoorde, M. Wong,
Direction for ISO C++
http://wg21.link/P0939

[PAAAA] Douglas, Niall
SG14 span<T> colouring
https://wg21.link/PAAAA

[PBBBB] Douglas, Niall
SG14 status_ code and error
https://wg21.link/PBBBB

[PDDDD] Douglas, Niall
Filesystem path views
https://wg21.link/PDDDD

[PEEEE] Douglas, Niall
SG14 Relaxing TriviallyCopyable into Relocatable and TriviallyRelocatable
https://wg21.link/PEEEE

[PXXXX] Douglas, Niall
Data persistence and algorithms (`iostreams` v2) study group
https://wg21.link/PXXXX

[1] *Boost.Outcome*
Douglas, Niall and others
https://ned14.github.io/outcome/

[2] Deukyeon Hwang and Wook-Hee Kim, UNIST; Youjip Won, Hanyang University; Beomseok Nam, UNIST
Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree
Proceedings of the 16th USENIX Conference on File and Storage Technologies (2018)
https://www.usenix.org/system/files/conference/fast18/fast18-hwang.pdf