D2052R0: Making modern C++ i/o a consistent API experience from bottom to top

Document #:	D2052R0
Date:	2020-01-12
Project:	WG21 Programming Language C++ Library Evolution Working Group Library Evolution Working Group Incubator SG1 Concurrency SG4 Networking

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

A proposed marriage of [P1031] *Low level file i/o* with [P1341] *Unifying Asynchronous APIs in the C++ standard library* to create a consistent i/o API experience from the very lowest levels, right up to the high level i/o API experience in the Networking TS [N4771].

Contents

1 Introduction

The 2019 Cologne meeting highlighted that multiple approaches and design philosophies to asynchronicity and i/o were beginning to 'butt heads' against one another. The 2019 Belfast meeting found unexpected committee support for the low level file i/o proposal [P1883] *`file_handle` and `mapped_file_handle`*, with much feedback being received that many would like to see a similar low level API treatment of non-file i/o. It was noted again during the second discussion of [P1750] *A Proposal to Add Process Management to the C++ Standard Library* that most of P1750 deals with i/o rather than processes, because nowhere else in the standard deals with the pipe i/o required to speak stdin, stdout and stderr to child processes.

Straight after the Belfast meeting I had an email discussion with all the major modern i/o players on WG21, during which I proposed building a consistent low level i/o API framework capable of solving everybody's use cases, while also being compatible with Sender-Receiver, and specifically designed so a Networking TS implementation could be wholly constructed from it. If successful, we would have a path forwards via which all modern i/o in the future C++ standard could be made a consistent experience for C++ developers, whilst retaining flexibility of composure and genericity, but also with low level i/o's strong guarantees of propagating the i/o determinism characteristics of the underlying platform.

Due to limits on free time, the prototype only implements `pipe_handle`. Due to surprise at the difficulty of matching ASIO's performance, which required several complete redesigns to achieve which were time-costly, by the time this paper was written the prototype only works on Microsoft Windows, and only in single threaded configuration. The recommendations made by this paper ought to be coloured appropriately.

2 Background and Context

As C++ moves into its fourth decade, there is a widespread yearning for its i/o facilities to be modernised. The considerable optional improvements to i/o specified in POSIX.2008 were made mandatory in POSIX.2017, however due to how long it took for the Filesystem TS to be standardised, i/o in standard C++ remains a 1990s-era experience.

The decision to standardise ASIO as the Networking TS in 2014 began an onslaught of standardisation work on the many prerequisites necessary to codify what 'work' is, and what 'execute' must mean [P0443]. This took longer than anyone expected, and it meant that the Networking TS moved target from C++ 17, then to 20, and most recently to 23.

Concurrent to these developments was the development of Coroutines which shipped with C++ 20, which is a language feature built into the core of C++ itself. As a language not library feature, Coroutines gets special 'magical' treatment by the compiler and runtime in various use cases and

contexts. Some felt that any Networking i/o proposal not built entirely around the same design language and design philosophy as Coroutines must be by definition suboptimal, even maybe the wrong choice going forward. Others felt that an ISO standards committee specifically exists to standardise existing practice, and that already far too many fundamental refactorings of ASIO had been performed into order to mutate it into something that WG21 would accept as the Networking i/o layer.

As the specifics for Executors and Networking as intended for C++ 23 have firmed up, the divisions between philosophical and ideological opinions on what should, and should not, became more apparent. This led to some particularly heated conversations within WG21 committee meetings, never mind elsewhere in the C++ userbase and ecosystem.

Thankfully, this is engineering rather than religion or politics, and a great deal of highly contentious debate can be solved to the satisfaction of all via demonstrating proof in code. The challenging problem at hand was whether it was possible to build an i/o framework which was a consistent API experience from lowest to highest levels, whilst retaining performance, determinism, and not being a pain to use.

3 Design principles

The goals I set out in the post-Belfast discussion for the extension of LLFIO into asynchronous non-file i/o were as follows, in order:

1. **Low level i/o should be an optimal building block experience on top of which to write high level abstractions**

   That means adhering to the LLFIO API design principles, which are: (i) strong ABI boundaries such that all platform-specific code lives behind an ABI and away from the translation unit, and header files are interfaces not implementation (i.e. very low build impact) (ii) all the determinism and performance supplied by the OS and platform are to be delivered undamaged to the API user (i.e. no dynamic memory allocation, no copying memory, no mutexs, no hidden waits) (iii) it should always be possible to max out your hardware from a single kernel thread, firstly in terms of minimum latency, secondly in terms of maximum throughput, *in that order*.

2. **A spinlooped polling API design should be the topmost asynchronous i/o API layer**

   In general, low latency can be exchanged for increased throughput, but throughput cannot be exchanged for lower latency. On that basis, a reactor-based API design – which is fundamentally about favouring throughput over latency – is the wrong choice for the topmost asynchronous i/o API layer. A reactor-based API ought to come *after* a spinlooped polling API layer.

   I appreciate that for those who have only ever used ASIO in its default out of the box configuration, it is hard to imagine an asynchronous i/o API which doesn't have an i/o service dispatching i/o completions to whichever next kernel thread becomes idle.

However for an important subset of Networking users (e.g. many who work in Finance), the word 'thread' is as anathema as the word 'malloc' is to audio programmers. Threads mean synchronisation, and synchronisation means loss of determinism. Furthermore completing i/o on the 'wrong' kernel thread can mean loss of cache warmth, also damaging determinism. Such low latency networking users run a custom build of ASIO usually with `ASIO_DISABLE_THREADS` defined, with a separate i/o service instance per kernel thread, and with Mellanox's `https://github.com/Mellanox/libvma` redirecting BSD socket APIs into non-kernel RDMA for extra low latency.

My goal was to directly support ultra-low-latency single-kernel-thread usermode RDMA i/o as the principle API design for low level asynchronous i/o, with failover to a reactor API layer. For some high end devices, looping `.poll()` over a (small) array of pending i/o would be all that you need do for minimum possible latency i/o. There would also be a second tier of API suitable for large arrays of pending i/o which efficiently as-if calls `.poll()` for only those handles which have signalled, but otherwise is as fully deterministic as it is possible to be. And finally, there would be a third tier of API which if no pending i/o has been signalled, will sleep the kernel thread until any pending i/o signals.

For those who don't care about latency and only care about throughput, they can exclusively use the third tier of API, and they would get what out-of-the-box ASIO currently provides.

3. **Match or exceed ASIO's performance**

   There is no point proposing a low level asynchronous i/o layer which is slower than ASIO, otherwise end users pay for what they don't use, and you cannot construct a useful implementation of ASIO on top.

4. **It must be possible to implement the Networking TS entirely using the low level framework**

   Given the design constraints already outlined, directly using the low level asynchronous i/o framework will be laden with footguns and unanticipated surprise, never mind much tedious verbosity. Strict discipline, and considerable domain experience, is required to write correct, high performance, low level i/o code – just as it is for someone writing lock free code using atomics.

   Conversely, the C++ userbase has extensive experience with ASIO; StackOverflow and many books are laden with answers and design patterns for how best to write high throughput networking code using ASIO; and the uninitiated programmer can get up to speed with writing correct, high throughput networking in C++ quickly given all this documentation already available.

   I **strongly believe** that we need to standardise ASIO **as is**. To do anything else will not be warmly received by the C++ userbase for the simple reason that *ASIO works*, and it works well without too much effort for its solution domain of high throughput networking i/o.

   This does not mean however that ASIO ought to be standardised as a black box which performs magic internally. If it can be standardised as a convenient set of high level wrappers abstracting

away the footguns, mandatory discipline and verbosity tedium which doing low level i/o directly must always force upon the developer, I find that the ideal outcome for standardising Networking i/o, personally speaking.

4 Walkthrough of the proposed low level i/o API design

4.1 Simple blocking i/o (extract from P1031 *Low level file i/o*)

Many thought twenty years ago that simple blocking i/o would never be used in high performance i/o code, however as the added overhead of implementing asynchronous i/o over the i/o itself increases due to hardware improvements, for use cases where the i/o is highly likely to be implemented as `memcpy()` (e.g. all file i/o served from the kernel cache, all RDMA on shared memory regions with a current local oplock etc), simple blocking i/o often beats all alternatives. As file and fabric i/o is very likely to be served from local RAM cache for most patterns of i/o, [P1031] *Low level file i/o* places simple blocking i/o at its heart.

P1031 specifies `.read()` and `.write()` as 'block until there is any data' operations. To be specific, if there is one or more bytes which can be i/o'd right now, perform that i/o and return the buffers partially or wholly filled/drained. If, and only if, no i/o can be performed right now, then do not return until some i/o has been performed. Both operations take a *deadline*, which is a relative or absolute timeout by which some i/o must occur, so forward progress can be guaranteed up to whatever the platform provides. A zero relative timeout means non-blocking i/o.

Under P1031, each handle type defines its own `buffer_type` and `buffers_type`, and they can be anything. For [P1883] *`file_handle` and `mapped_file_handle`*, and indeed the `pipe_handle` as implemented by our prototype, they are types concept-requirement matching those of `span<byte>` and `span<span<byte>>` respectively, but with the guarantee that layout matches that of the platform-specific scatter-gather structure type (e.g. `struct iovec` on POSIX). This enables scatter-gather buffer lists to be handled straight to the OS, without conversion nor repacking.

Under P1031, each handle type also defines its own `io_request` type. This can also be anything. For [P1883] *`file_handle` and `mapped_file_handle`*, and indeed the `pipe_handle` implemented by our prototype, it is:

```
// A scatter-gather list of buffers, and file extent offset at which to perform the i/o.
template <class T> struct io_request
{
  T buffers{};
  extent_type offset{(extent_type) -1};

  constexpr io_request();
  explicit constexpr io_request(T _buffers);
  constexpr io_request(T _buffers, extent_type _offset);
};
```

One might be surprised that a non-seekable handle type such as `pipe_handle` still takes an extent offset. This is because an append-only `file_handle` can be constructed where writes are always atomically appended to the file, and for which the offset request is ignored. Attempting reads at

offsets exceeding the current maximum file extent also have various platform-specific semantics, which includes ignoring the requested offset. The concept of the offset being ignored in various situations is therefore already well formalised. Additionally, code written for a non-seekable handle type can use an append-only `file_handle` and work correctly, and indeed for a number of algorithms, it is very useful for have unbounded buffer length pipes where writes never block[1]. There are also debugging and logging use cases. In short, it was felt that, on balance, retaining an identical API for pipe handle i/o to file i/o was worth it, and my personal opinion on that would probably also extend the same offset-based i/o API to low level socket i/o as well.

Finally, under P1031 each handle type also defines its own `io_result` type which extends the `result<T>` from [P1028] *SG14 `status_code` and standard `error` object*, which allows handle types to implement free-upon-result-destruction patterns, and so on. A `result` returns either `T` for success, or `error` from P1028. If you observe the value on an errored result, result invokes `error().throw_exception()` on your behalf to throw the C++ exception defined by the `error`'s domain. Here is the full API:

```
/*! \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 reqs A scatter-gather and offset request.
\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.
*/
io_result<buffers_type> read(io_request<buffers_type> reqs) noexcept;

//! \overload Convenience nonblocking based overload for 'read()'
io_result<buffers_type> try_read(io_request<buffers_type> reqs) noexcept;

//! \overload Convenience duration based overload for 'read()'
template<class Rep, class Period>
io_result<buffers_type> try_read_for(io_request<buffers_type> reqs,
                                     const std::chrono::duration<Rep, Period> &duration) noexcept;

//! \overload Convenience absolute based overload for 'read()'
template<class Clock, class Duration>
io_result<buffers_type> try_read_until(io_request<buffers_type> reqs,
                                       const std::chrono::time_point<Clock, Duration> &timeout)
                                           noexcept;
```

The write API is so similar as to be not worth spelling out (if you really want to know, see [P1883]

[1] `file_handle::zero()` deallocates written extents in a file, so when combined with a temporary inode, you can easily create a true unbounded buffer length pipe.

file_handle and mapped_file_handle). But before moving on, here is an example of use of the simple blocking API:

```
namespace llfio = LLFIO_V2_NAMESPACE;

// Open the file for read
llfio::file_handle fh = llfio::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<llfio::byte> buffer(fh.maximum_extent().value());

// Make a scatter buffer to indicate what to fill
llfio::file_handle::buffer_type b{ buffer.data(), buffer.size() };

// Make an i/o request object
llfio::file_handle::buffer_types req{
    {b},  // sequence of buffers to fill
    0     // offset within file to start reading from
};

// Synchronous scatter read from file
llfio::file_handle::io_result<buffers_type> buffersread = llfio::read(
    fh,     // handle to read from
    req     // i/o request to perform
).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(buffersread.bytes_transferred());
```

4.2 Simple 'blocking' coroutine i/o

When writing code within C++ Coroutines, often one wishes to suspend execution of the coroutine until an i/o completes, resuming execution thereafter.

This looks almost identical to the simple blocking i/o code under P1031:

```
namespace llfio = LLFIO_V2_NAMESPACE;

llfio::pipe_handle::buffer_type buffer;
for(;;)
{
    // This will never return if the coroutine gets cancelled during suspension
    llfio::pipe_handle::io_result<llfio::pipe_handle::buffers_type> r
      = co_await read_pipe.co_read({    // use co_read() instead of read()
        {buffer},                       // otherwise *identical* arguments to read()
        0
```

```
      });
    if(!r)
    {
        co_return r.error();
    }
    BOOST_CHECK(r.value().size() == 1);
    BOOST_CHECK(r.value()[0].size() == sizeof(_buffer));
  }
}
```

The overwhelmingly most common point of feedback is 'I really hate the `co_` prefix. We need that to go away in general, not become worse than it already is'.

And I empathise, I truly do. However, the `co_` naming ship has sailed in C++ 20, and that can no longer be undone.

The only difference between `.co_read()`, `.co_write()` and `.co_barrier()` over their non-`.co_` equivalents is that they return an `io_awaitable<async_read>`, `io_awaitable<async_write>` or `io_awaitable<async_barrier>` respectively. Each takes identical input parameters, so the only way to disambiguate them from their non-awaitable equivalents would be via function naming in any case. And a `co_` prefix is as good as any, and it is consistent with the `co_` prefixed keywords.

> [*Note:* For those unfamiliar with Sender-Receiver, you may wish to study the next section before re-reading this section. – end note]

The `io_awaitable<T>` type is actually a Receiver wrapper around an i/o Sender (from [P1341] *Unifying Asynchronous APIs in the C++ standard library*) which tells C++ Coroutines how to do the i/o. The awaitable's Receiver definition is trivially simple:

```
template <class ResultType>
struct io_awaitable_receiver_type
{
    ResultType *result{nullptr};
    coroutine_handle<> coro;

    void set_value(ResultType &&res)
    {
        result = &res;
        if(coro)
        {
            coro.resume();
        }
    }
    void set_done() {}
};
```

In fact, the guts of the whole i/o awaitable type is sufficiently short it can be listed here too:

```
bool await_ready() noexcept
{
    // If i/o not started yet, start it
    if(!_state->started())
    {
        _state->start();
```

```
    }
    // If i/o already completed, we are ready
    if(_state->completed())
    {
        return true;
    }
    // Poll i/o for completion, if it completes we are ready
    if(_state->poll() == io_status_type::completed)
    {
        return true;
    }
    return false;
}
_result_type await_resume()
{
    assert(this->_receiver.result != nullptr);
    return std::move(_state->storage.ret);
}
void await_suspend(coroutine_handle<> coro)
{
    _lock_guard g(this->lock);
    if(_state->completed())
    {
        coro.resume();
        return;
    }
    _state->_receiver.coro = coro;
}
```

Walking through a use instance, `.co_read()` which returns an `io_awaitable<async_read>` initialises an `async_read` Sender, and passes it to `io_awaitable` which then Connects it with the i/o awaitable's Receiver implementation. We return the Connected, but not Started, i/o state transported by the i/o awaitable returned from `.co_read()`.

As per how C++ Coroutines works, when calling `co_await` on an awaitable type, `.await_ready()` is first asked if the awaitable is ready. In the above implementation, if this is the first time `.await_ready()` has been called, we Start the previously Connected i/o state. This begins the i/o, which may complete immediately (e.g. it was served from kernel cache, so no blocking needed). If it does complete immediately, we return true to indicate that no coroutine suspension is required. The coroutine would therefore proceed as optimally as is possible.

You may notice there are two ways of checking for i/o completion. The first, `_state->completed()`, is a simple read of a flag saying whether the i/o is completed. The second, `_state->poll()` involves a virtual function call (hence the early check) which calls some handle-specific i/o code to poll the i/o for completion. It is required that `.poll()` be deterministic and to do as little work as is possible to check for the i/o to be completed, for some definition of 'little work' (e.g. a short fixed length spin loop checking a hardware register might be acceptable).

As per C++ Coroutines, if `.await_ready()` returns false, `.await_suspend()` is called to suspend the coroutine's execution. Other code will now execute. At some point, someone else would call the connected i/o state's `.poll()` function to poll the i/o, or call a function which figures out which connected i/o states are ready to have their `.poll()` function called. If the i/o has now

completed, the Receiver's `.set_value()` will be called, which resumes any suspended coroutine. Upon resumption, the coroutine's `co_await` operator will call `.await_resume()` to fetch the result of the i/o, which will be the scatter-gather buffers filled or drained, same as for simple blocking i/o.

Most of why the above simple 'blocking' coroutine i/o is so simple to implement is thanks to Sender-Receiver, which is next.

4.3 Low level Sender-Receiver i/o

Low level Sender-Receiver i/o is just a small step upwards from what we've already seen. I should caveat that the Sender-Receiver design presented here is not exactly the one from P1341R0. Firstly, I have incorporated my best understanding of the upcoming changes to Sender-Receiver which will be proposed by Lewis, Kirk and Eric and others in papers in this mailing, and because we were all so time pressed before the mailing deadline this year due to Christmas, we have not coordinated paper content as much as would be usual (sorry!). Secondly, I had to make some changes to the Sender-Receiver lifetime model (covered shortly) which Lewis thinks solved by overly simplistic means, and he has an alternative mechanism to better achieve the same ends which I haven't seen yet.

Still, none of that is as important as making the general point that Sender-Receiver has evolved since the last meeting, and we all of us are all basically on the same page.

Under Sender-Receiver, to begin say a read i/o, we firstly construct an `async_read()` sender. This takes identical arguments, and has identical form, to simple blocking `read()`:

```
async_read::async_read(handle &, io_request<buffers_type>);

async_read::try_async_read(handle &, io_request<buffers_type>);

template<class Rep, class Period>
async_read::try_async_read_for(handle &,
                               io_request<buffers_type>,
                               const std::chrono::duration<Rep, Period> &);

template<class Clock, class Duration>
async_read::try_async_read_until(handle &,
                                 io_request<buffers_type>,
                                 const std::chrono::time_point<Clock, Duration> &);
```

`async_read` is the thread-unsafe form of this i/o sender. If you want a thread-safe form, you use `atomic_async_read` instead. The latter requests atomic synchronisation for the connected state to ensure that multiple kernel threads concurrently using the connected i/o state will synchronise. If you are building a Networking TS on top of low level Sender-Receiver i/o, `atomic_async_read` is what you must use within your implementation, as the completion handlers which the Receiver would invoke could traverse kernel threads.

We next Connect this Sender instance with a Receiver instance, which must implement these member functions:

```
// Called when the i/o has completed with the i/o result
```

```
void Receiver::set_value(io_result<buffers_type> &&);

// Called when the i/o state can be disposed of
void Receiver::set_done();
```

(Due to using `io_result` to return failure deterministically, and because we can never throw exceptions, we do not use `Receiver::set_error()`)

In the reference implementation, Connecting an i/o sender and a Receiver yields an `io_operation_connection<Sender, Receiver>` object. This provides a `.start()` member function which implements the Start operation. You can relocate in memory a connected i/o state up until you perform Start upon it. Thereafter you cannot relocate it in memory until its `.set_done()` is called.

4.3.1 Polling and awaiting

Once you have Started a connected i/o state, you must periodically call `io_operation_connection<Sender, Receiver>::poll()` upon it to (a) check if the i/o it began has completed and (b) to cause its Receiver to be invoked if the i/o did just complete.

`io_operation_connection<Sender, Receiver>::poll()` is guaranteed to be a deterministic call apart from whatever the invocation of the Receiver might do, and whilst efficient for small numbers of pending i/o, it becomes inefficient when the number of pending i/o becomes large. Therefore every `handle` has an associated `io_multiplexer` instance into which every asynchronous i/o is collated. One can as-if call `io_operation_connection<Sender, Receiver>::poll()` on all pending i/o known to that `io_multiplexer` instance, up to specified limits, by calling the multiplexer's `.complete_io()` member function:

```
// As-if call the i/o part of `.poll()` on up to `max_items` pending i/o, returning how many called.
result<int> io_multiplexer::complete_io(int max_items = -1) noexcept;

/* As-if call the i/o part of `.poll()` on up to `max_items` pending i/o, not exceeding the specified
maximum duration or time point.
*/
template<class Rep, class Period>
result<int> io_multiplexer::complete_io_within(
                            const std::chrono::duration<Rep, Period> &duration,
                            int max_items = -1) noexcept;
template<class Clock, class Duration>
result<int> io_multiplexer::complete_io_within(
                            const std::chrono::time_point<Clock, Duration> &timeout,
                            int max_items = -1) noexcept;
```

`.complete_io()` is not guaranteed to be deterministic, but it is guaranteed to be non-blocking, apart from any blocking that the Receiver invocation might do. Most platforms will implement it with `O(1)` complexity, but others could have `O(total pending i/o)` complexity. It may complete one, or many pending i/o before it returns, the number depending on what is efficient on the local platform, and obviously enough on the number of ready pending i/o. To aid bounded execution times, one can bound i/o completions to within numerical or time constraints – this can help ensure that `.complete_io()` returns in less time than it might otherwise. If there is no ready pending

i/o right now, a negative number is returned. If there is no pending i/o at all right now, zero is returned.

`.complete_io()` only checks for i/o completion based on i/o being successful or unsuccessful. It does not perform any checking for i/o timeouts. If you never place timeouts other than infinite or zero on scheduled i/o, then you never need to call this function:

```
// As-if call the timeout part of '.poll()' on up to 'max_items' pending i/o, returning how many
// called.
result<int> io_multiplexer::timeout_io(int max_items = -1) noexcept;

// Time bounded editions
template<class Rep, class Period>
result<int> io_multiplexer::timeout_io_within(
                            const std::chrono::duration<Rep, Period> &duration,
                            int max_items = -1) noexcept;
template<class Clock, class Duration>
result<int> io_multiplexer::timeout_io_within(
                            const std::chrono::time_point<Clock, Duration> &timeout,
                            int max_items = -1) noexcept;
```

`.timeout_io()` is very similar to `.complete_io()` in terms of behaviour, however it is fundamentally an `O(total pending i/o)` complexity operation as an ordered list of all i/o with timeouts must be constructed in order to figure out the priority queue for completing i/o with `errc::timed_out`. This list is cached within the i/o multiplexer, and only recently added pending i/o is merged into those lists. However this call is unavoidably more expensive than `.complete_io()`, which is why you only need to call it if you use timeouts in your i/o.

4.3.2 The i/o multiplexer

The i/o multiplexer provides a facility for posting invocables from any kernel thread. This is always thread safe, even if the i/o multiplexer implementation itself is not thread safe (see later). There is a corresponding drain function for posted invocables:

```
// Will invoke 'f()' later. Always threadsafe.
template <class U>
requires(std::is_invocable_v<U>)
result<void> io_multiplexer::post(U &&f);

// Invoke up to 'max_items' previously posted items
result<int> io_multiplexer::invoke_posted_items(int max_items = -1) noexcept;

// Time bounded editions
template<class Rep, class Period>
result<int> io_multiplexer::invoke_posted_items_within(
                            const std::chrono::duration<Rep, Period> &duration,
                            int max_items = -1) noexcept;
template<class Clock, class Duration>
result<int> io_multiplexer::invoke_posted_items_within(
                            const std::chrono::time_point<Clock, Duration> &timeout,
                            int max_items = -1) noexcept;
```

Unless you call `io_multiplexer::run()`, it is on user code to drain the posted items queue, if they wish.

The final facility is `io_multiplexer::run()`, which roughly corresponds with ASIO's `io_context::poll()` (I couldn't name the function `.poll()`, as it would be confusing with the i/o connected state's `.poll()`). `io_multiplexer::run()` looks boringly similar to the previous functions, but note that instead of time bounding total execution time, we now time limit waits upon new work. Hence instead of using the `_within()` nomenclature, we use the idiomatic `try_`, `try_X_for` and `try_X_until` nomenclature:

```
/* As-if call all of `.poll()` on up to `max_items` pending i/o, and `.invoke_posted_items()`,
returning how many called/invoked.
*/
result<int> io_multiplexer::run(int max_items = -1) noexcept;

// Non blocking edition
result<int> io_multiplexer::try_run(int max_items = -1) noexcept;

// Timeout editions
template<class Rep, class Period>
result<int> io_multiplexer::try_run_for(
                            const std::chrono::duration<Rep, Period> &duration,
                            int max_items = -1) noexcept;
template<class Clock, class Duration>
result<int> io_multiplexer::try_run_until(
                            const std::chrono::time_point<Clock, Duration> &timeout,
                            int max_items = -1) noexcept;
```

`io_multiplexer::run()` essentially combines `.complete_io()`, `.timeout_io()`, and `.invoke_posted_items()` in a loop. If any of those three return a positive number, at the end of the loop `run()` returns with the number of items processed. If there is no pending i/o to wait upon, and no posted items, `run()` immediately returns with zero. If there is pending i/o but no ready i/o, the calling thread is put to sleep until a pending i/o signals, or somebody posts a callable to be invoked from another kernel thread, or any timeout specified elapses. This 'wait-until' semantic differs from the 'work-until' semantic in the previous functions, because this function is always non-deterministic, with potentially unknown lengths of thread sleep possible.

4.3.3 Example of use of Sender-Receiver low level i/o

Believe it or not, I just described *all* of the essentials of an implementation of low level asynchronous i/o. This proposal is quite compact. Just a few pages of normative wording in addition to non-asynchronous low level i/o, would describe it all.

Using low level asynchronous i/o is very straightforward, though verbose to write, as it is for all low level i/o. Here is how to begin many asynchronous read i/o's:

```
namespace llfio = LLFIO_V2_NAMESPACE;

// Assume there are many read pipes, and each has a buffer to be filled
std::array<std::pair<llfio::pipe_handle, llfio::pipe_handle::buffer_type>, N> pipes;

```

```
// Define the Receiver for a pipe read
struct pipe_read_receiver
{
  size_t idx{0};

  void set_value(llfio::pipe_handle::buffers_type &&filled)
  {
    std::cout << "Pipe " << idx << " read " << filled.bytes_transferred() << " bytes." << std::endl;
  }
  void set_done() { /*do nothing*/ }
};

// Connect asynchronous reads of all read pipes to their receivers
std::array<llfio::io_operation_connection<llfio::async_read, pipe_read_receiver>, N> connections;
for(size_t n = 0; n < N; n++)
{
  connections[n] = llfio::connect(
    llfio::async_read(                                  // sender
      pipes[n].first,                                           // handle to read
      {                                                         // i/o request object
        { pipes[n].second },                                    // scatter buffer to fill
        0                                                       // offset within device to read from
      }),
    pipe_read_receiver{n}                               // receiver
  );
}

// Begin asynchronously reading all the pipes
for(size_t n = 0; n < N; n++)
{
  connections[n].start();
}
```

Something not obvious in the above is:

- No dynamic memory allocations can ever be performed.
- No C++ exceptions can ever be thrown.
- No locks can ever be taken.
- No waits can ever occur.
- The compiler can see all the memory that can ever be used by the asynchronous i/o from the base of the call tree. This enables various optimisations e.g. coroutine frame allocation can perform a single allocation for all state.

The above code is therefore *fully deterministic*, if your OS pipe handle implementation can begin async read i/o's deterministically.

You now have a choice of how to implement i/o polling. Here is a simple spinlooped polling based implementation:

```
// Loop polling all the pipes until all have read something, or failed to read something
bool done;
do
```

```
{
  done = true;
  for(auto &i : connections)
  {
    if(llfio::update_status::completed != i.poll())
    {
      done = false;
    }
  }
} while(!done);
```

If all the pipes being read have at least one byte in their buffers, the above code is also fully deterministic and wait free, if your OS pipe handle implementation can drain a non-empty buffer deterministically.

If you don't mind losing determinism, the above simple spinlooped polling based implementation can be implemented more efficiently using:

```
// Loop efficiently polling all the pipes until all have read something, or failed to read something
size_t done = 0;
while(done < N)
{
  int completed = pipes[0].first.multiplexer()->complete_io().value();
  if(completed > 0)
  {
    done += completed;
  }
}
```

This is more efficient because `complete_io()` asks the OS which of the read i/o's scheduled have completed, and calls the receivers for those completed i/o's only. This avoids having to linearly iterate polling each of the whole array of scheduled read i/o's, which would become inefficient as `N` grows large.

`complete_io()` never blocks, so the CPU will spin at 100% in the loop. If you would like the thread to be slept until a read i/o completes, just replace `complete_io()` with `run()`:

```
// Sleep until all the pipes until all have read something, or failed to read something
size_t done = 0;
while(done < N)
{
  int completed = pipes[0].first.multiplexer()->run().value();
  if(completed > 0)
  {
    done += completed;
  }
}
```

The above code is all you need to do asynchronous low level i/o. Some relevant implementation detail not mentioned for brevity is that unless you explicitly set the i/o multiplexer for a `handle`, it will set itself on first async i/o to use the current thread locally set i/o multiplexer. If one of those hasn't been retrieved for this kernel thread before, `io_multiplexer::best_available(1)` is called to retrieve the best available i/o multiplexer implementation for this platform which supports

no more than one thread. Thus the above code works because the programmer knows that the thread-local i/o multiplexer is being used, and that no other i/o has been scheduled by this kernel thread to that i/o multiplexer (otherwise the completed i/o counts returned by `complete_io()` and `run()` would refer to other scheduled i/o that just our pipes, and confound the logic).

4.4 Summary

I have achieved all I set out to do before Christmas in this prototype:

1. Use a spinloop-polling-based Sender-Receiver API design throughout, rather than a Proactor/Reactor API design.
2. Enforce a strict ABI boundary behind which all platform-specific stuff can be encapsulated (i.e. don't leak system headers into the interface), but without getting in the way of the compiler being able to statically merge dynamic memory allocations.
3. Match or exceed ASIO's performance despite all the above constraints.
4. It must be possible to build a backwards compatible Networking TS/ASIO wrapper on top of all this.

In short: I have proven it is possible to standardise a single consistent i/o API experience from simple blocking i/o right through to ultra high performance i/o. And it's all thanks to Sender-Receiver.

It is believed by this author that the above API contains all the necessary primitive operations to implement the current Networking TS exclusively using low level i/o facilities. It is possible that some of ASIO's proactor facilities (e.g. strands) could possibly have a less optimal implementation than is possible with strictly employing the above facilities alone. However, personally speaking, I would implement strands as a C++ Coroutine in C++ 20, whose suspension and resumption may occur across kernel threads, and I believe such a strand implementation would be optimal with the above facilities.

I should stress that I don't wish to hold up the Networking TS from C++ 23 based on these proposals. I would simply not standardise the parts (e.g. the socket classes) which would be standardised as low level socket i/o classes in C++ 26. Whilst this may seem highly problematic, both ASIO and the Networking TS allow user defined socket i/o classes, so standard library implementations simply ship their own locally defined stand-in socket i/o classes, or else ship experimental low level socket i/o classes based on P1031. Indeed, if standard library implementers were feeling keen, they could ship their Networking implementation written using experimental low level i/o as this paper outlines, and send implementation feedback to WG21.

The point I wish to make is that we just need to not standardise in the normative wording of C++ 23 the stuff that we can't row back upon in C++ 26, and we can press ahead with standardising Networking now.

5 Implementation notes from developing this prototype

5.1 Background

Back when I started this prototyping in the run up towards the Christmas vacation break, I *thought* that I understood Sender-Receiver. I had read Corentin's excellent blog post at https://cor3ntin.github.io/posts/executors/. I had watched Eric's many video presentations at many conferences on universal asynchronicity in C++. I had been a follower of Lewis' https://github.com/lewissbaker/cppcoro for many many years. And of course I had been keeping up with reading my WG21 mailings, so I had read all the relevant papers.

However it turns out that I really **did not** understand Sender-Receiver. Not one bit. And I would like to take this opportunity to publicly thank principally Eric Niebler for gently holding my hand over many private email exchanges as I laboriously thrashed around in confusion until, thanks to his excellent tutelage, one day the lightbulb just switched on, and it all made sense.

It didn't *just* make sense though. It was far, far better than that. The LLFIO reference library has long had an async file i/o implementation based on a similar design to ASIO. Its performance, quite frankly, sucked; implementation was a pain; and I generally steered anybody interested away from async file i/o, which is generally slower than blocking file i/o except for uncached i/o.

Between Belfast and Christmas break in 2019, I had built a new async i/o framework around the deterministic polling-based design principles I described earlier in this paper. It was very complex. It was buggy. Lewis very kindly sent a long, long list of entirely valid feedback which basically said (though he never did) 'this is crap'. Then the Sender-Receiver lightbulb switched on, and suddenly it all became so clear as to what should be the design: clean, elegant lines of efficiency, code reuse and abstraction. My only (big!) concern was whether this stuff could be implemented with a hard ABI boundary separating the platform-specific code away from the template-stuff (as hitherto, all the Sender-Receiver stuff did not use concrete types, and it was not known at that time whether it could work with concrete types over a formal ABI boundary), without losing a lot of performance, and retaining the magical ability of the compiler to collapse and merge Coroutine frame allocation because it can foresee all static memory allocations from the base of a call tree. As by that stage Christmas vacation was about to begin, I wished everybody a Merry Christmas, and I set to work throwing away everything and trying again.

The message I want to communicate to anyone reading is that if you don't like Sender-Receiver for various reasons, then in my opinion you don't really understand it yet. Sender-Receiver is an *architecture*, a way of laying out, structuring, and grouping your code into reusable parts upon which the compiler has magical powers to optimise particularly well, thanks to taking advantage of how Coroutines were standardised. Sender-Receiver is *genius* in my opinion. It's so damn simple people can't see just how game changing it is: it makes possible fully deterministic, ultra high performance, extensible, composable, asynchronous *standard* i/o. That's **huge**. No other contemporary systems programming language would have that: not Rust, not Go, not even Erlang.

Before I move on, I'd like the publicly thank all those who brought Sender-Receiver to C++, which was a group effort, but Kirk obviously enough is the visionary who invented Sender-Receiver, and Lewis did much of the building out of, and tying in, that bright spark of innovation into Coroutines,

Connect-Start, etc and all the technical donkey work of building out an opinion in favour of Sender-Receiver. Eric has worked his ass off doing all the liaising, persuading, reaching out, coordination, and basically non-technical human management stuff which nobody tends to notice happening behind the scenes to bring technical innovations into the standard.

Persuading people to adopt complex niche stuff is hard, but persuading people to adopt new fundamental vocabulary requires particularly deep effort, for which very few will thank you because it was disruptive. I thank you all for your service.

5.2 Performance numbers

The benchmark chosen was deliberately designed to illuminate i/o framework implementation inefficiencies. There is a single pipe, with separate read and write handles to it. A pool of threads concurrently tries to write to the pipe using blocking i/o, in order to always keep its buffer full. On the read side, the framework under test reads a single byte from the pipe, and its completion handler schedules the next read of a single byte from the pipe. The idea is that we spend as much time within the i/o framework as is possible for a single kernel thread, and thus get an idea of its overhead relative to others. All testing was carried out on a laptop with four core Intel i7-8565U CPU boosting up to 4.6Ghz running Microsoft Windows 10, using Visual Studio 2019 16.4 as the compiler.

My first Sender-Receiver design was clean and simple, but it was far slower than ASIO in the first benchmark I ran. This first attempt was about half the performance of ASIO, which was a big surprise at the time.

Upon diagnosis, a large portion of the cause was due to how Sender-Receiver was specified before I began my efforts (and for which Sender-Receiver ended up needing to be changed). To understand this properly, I shall have to delve into quite a bit of implementation detail.

All non-blocking i/o in Microsoft Windows is done via the two-pointer-sized `IO_STATUS_BLOCK` structure:

```
typedef struct _IO_STATUS_BLOCK {
  union {
    NTSTATUS Status;
    PVOID    Pointer;
  } DUMMYUNIONNAME;
  ULONG_PTR Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

There is always exactly one of these structures per pending i/o (most reading will better recognise the Win32 formulation of this NT kernel structure, `struct OVERLAPPED`). The kernel will (potentially asynchronously) write into this structure when the i/o completes either with success or failure, so it cannot move in memory during the i/o.

The Sender-Receiver Connected state similarly cannot be moved in memory between Start and when either the Receiver's `.set_value()`, `.set_error()` or `.set_done()` is called (these correspond to asynchronous operation success, failure and abort respectively). My initial design therefore placed the `IO_STATUS_BLOCK` structure within the connected i/o state structure, which made sense.

Microsoft Windows has three mechanisms by which you can poll/force changes to the `IO_STATUS_BLOCK` structure:

1. Poll the `HANDLE` using a (possibly zero-timeout) wait-until-handle-signals instruction which returns when some i/o completes on that handle somewhere. One then checks this structure to see if this specific i/o completed.

   This is how `io_operation_connection<Sender, Receiver>::poll()` is implemented on Microsoft Windows: it performs a single non-blocking kernel wait on the handle, and it invokes the Receiver for the i/o if its `IO_STATUS_BLOCK` structure changes its status from `STATUS_IO_PENDING`. Similarly, this is how simple blocking i/o on non-blocking handles is implemented in LLFIO, one simply loops the wait-until-handle-signals syscall and checking the `IO_STATUS_BLOCK` structure until any deadline is reached.

2. Specify a per-i/o user supplied callback function to be invoked when the i/o completes at the next alertable wait point. In Microsoft Windows, thread sleep points can be specified as alertable or not, and if so they will wake as soon as any callback is posted to that thread, and drain the per-thread queue by invoking all the callbacks before the thread sleep point returns. This is often called 'alertable i/o', or 'APC i/o' (APC is an Asynchronous Procedure Callback).

3. Associate the `HANDLE` with a completion port, whereby notifications of i/o status changes in associated handles are enqueued to the completion port. The queue of accumulated notifications can be drained by one or many kernel threads. This is often called IOCP, shorthand for i/o completion ports. ASIO uses IOCP as its implementation on Microsoft Windows.

One can see easily that for the first mechanism, you only need to keep around the `IO_STATUS_BLOCK` structure until its status changes, and for the second mechanism you only need to keep it around until its callback function gets invoked. However, there is a big implementation wrinkle in the third IOCP mechanism: the `IO_STATUS_BLOCK` structure gets updated *for a second time* when somebody drains the completion notification for that i/o. In other words, you absolutely must keep around the `IO_STATUS_BLOCK` structure until IOCP is done with it, else you get memory corruption.

That in turn means that under the original Sender-Receiver specification, for IOCP registered handles, you can only call `.set_value()` and `.set_error()` and `.set_done()` **after** some thread has reaped the associated completion from IOCP. Specifically, you cannot call it *early* immediately when the i/o has completed, because under the original Sender-Receiver design, after the Receiver is invoked its state can be destructed. So you must wait.

This explains the performance difference of my first Sender-Receiver design against ASIO. In our benchmark against ASIO, we read a single byte from a single pipe handle using `ReadFile()`. We must next drain the IOCP queue using `GetQueuedCompletionStatus()`, and then invoke the Receiver now that IOCP is done with the i/o.

ASIO however invokes the completion handler immediately after `ReadFile()`, and does not wait to drain the IOCP queue first. This lets it begin the next i/o much sooner, and thus ASIO performs multiple `ReadFile()` syscalls and completion handler invocations for each drain of the IOCP queue via the `GetQueuedCompletionStatus()` syscall.

ASIO achieves this via a trick: each `OVERLAPPED` structure is kept in its own dynamically allocated

memory allocation. Smart pointers to each `OVERLAPPED` structure track completion handler invocation state. If an i/o completes before its IOCP notification is drained, its completion handler is called immediately, and will not be called again when the IOCP notification is reaped. Similarly, if IOCP is the first time ASIO learns of i/o completion, the completion handler is called at that point instead.

In this design, there can be a significant lag between completion handler invocation and draining the IOCP queue. ASIO handles this by dynamically allocating more `OVERLAPPED` structures as needed (it recycles structures dispensed with by IOCP).

5.3 Matching low-level Sender-Receiver i/o with ASIO's performance

Most reading this paper will not consider it important that potential dynamic memory allocation is unavoidable in the hot i/o path in ASIO's design. They would suggest using a custom allocator with ASIO, or 'preheating' ASIO. They would consider i/o to be inevitably non-deterministic, and thus striving for determinism in i/o a fool's errand.

However for me and those thinking like me who tend to hang around on SG14 Low Latency, I find that attitude symptomatic of developers conditioned on 'throughput-maximising' designs where you must work with the lowest common denominator in i/o (e.g. speaking the kind of network protocols required by the fundamentally non-deterministic internet). If you control your i/o implementation however, you can do much, *much* better.

In my line of work I really care about worst case latencies. Not average latencies, latencies @ 99%, or even latencies at 99.9%. I'm talking about latencies @ 99.9999%. Warm cached file i/o is almost perfectly deterministic on Linux, and high end consumer SSDs e.g. my Samsung 970 Pro i/o is deterministic 99% of the time. Enterprise grade SSDs can be much better again e.g. Samsung's Z-SSD, or Intel's Optane, which guarantee up to six sigmas of bounded i/o latencies. Lots of people pay a lot of money for storage hardware which can deliver 99.9999% determinism, and then find that most i/o software can't preserve determinism.

As one of my design goals is to retain determinism in i/o which is very precious to many use cases, and which hardware and kernel engineers have spent much effort in ensuring, replicating ASIO's trick which relies on potential non-determinism in the hot path wasn't possible. So instead I changed the Sender-Receiver lifecycle:

- Start begins the operation on the Connected state.
- The Receiver's `.set_value()` and `.set_error()` *completes* the operation.
- The Receiver's `.set_done()` is called when it is safe to dispose of or recycle the Connected state.

Note that `Receiver::set_done()` is now **always** called for every Started operation, and you cannot relocate in memory a Started state until `.set_done()` is invoked.

This change enables `.set_value()` to be called as soon as the i/o completes, which in turn closed most of the performance gap with ASIO. In fact, with that and a few other optimisations, LLFIO's

reference implementation gets within 2% of ASIO for the single byte reading benchmark if LLFIO speaks IOCP in the same way as ASIO does.

However thanks to this being a greenfield design, I was able to incorporate native support for 'Win8 IOCP' which can be opted into using `SetFileCompletionNotificationModes()`. This does not enqueue IOCP notifications for i/o which completes immediately. This enabled the reference implementation upon which this paper has been based to exceed ASIO's performance by 23% for the single byte single pipe reader benchmark.

I believe there is further fat to trim if one does not insist on a strict ABI boundary between platform-specific implementation code and everything else – we spend a measurable amount of time marshalling and repacking data to go through a vtable indirection, which wouldn't be necessary if say we had an intelligent enough Modules implementation to hand. I certainly believe that the 2% like-with-like performance gap from ASIO can be closed to zero.

5.4 Consequences of extending i/o state lifetime past completion

It may initially seem unwieldy for i/o states to hang around for arbitrarily long periods of time after i/o completion. How might code manage this situation without resorting to dynamic memory allocation? In other words, aren't we simply forcing end users to repeat the implementation trick ASIO used?

The answer is yes. But if you want this sort of implementation detail to be taken care of on your behalf, use ASIO!

If your use case requires absolute guaranteed bounded worst case i/o latencies, then that comes with a price attached: a considerable added headache in managing i/o states. But I personally think that an acceptable tradeoff, and I think most of my colleagues on SG14 Low Latency would agree.

6 What next?

If WG21 likes the general direction of this proposal, the first step would be to remove from the normative wording in the next edition of the Networking TS any specification of the basic socket classes, and leave them open for standardisation after C++ 23.

The second step would be to consider separating the socket-support parts of the Networking TS such as IP addresses, endpoints, and DNS resolution into an entirely orthogonal and standalone section, or indeed standalone paper. Or, indeed choose to provide non-normative notes for the specification of these, rather than normative wording in C++ 23.

The third step would be to consider separating the buffer sequence adapter parts of the Networking TS into an entirely orthogonal and standalone section, or indeed standalone paper. In my opinion, the buffer sequence infrastructure is useful for all i/o, and ought to be decoupled from Networking.

The fourth step would be to ensure that all the `async_*()` completion-token based free functions do not conflict with the identically named `async_*` Sender types. I believe Lewis and Eric have a proposed mechanism for ensuring this. So long as their proposed mechanism is compatible with a

strict ABI boundary i.e. the Sender types are concrete for each category of handle types, I would have no objection.

Stripping away all these orthogonal parts would considerably reduce the number of pages in the present Networking TS draft, and I believe would aid its more rapid standardisation into C++ 23 by helping everybody concentrate on the specific value which standardising ASIO adds to the standard C++ library: a suite of well tested, widely understand, high level concurrency and i/o abstractions.

7 Acknowledgements

Once again my especial thanks to Eric Niebler and Lewis Baker for all the personal help they gave me.

Thanks are also due to useful feedback and conversations with Corentin, Klemens, Elias and Jeff.

Finally the prototype, and this paper, would not have been possible without significant forbearance from my family during Christmas vacation, who put up with me battering away on the laptop whilst watching movies, and other passive forms of Christmas family entertainment.

My especial thanks go to my wife Megan, who enabled me to tag team childcare each day across the break in order to get this work done before the Prague mailing deadline.

8 References

[N4771] Wakely, Jonathan
Working Draft, C++ Extensions for Networking
https://wg21.link/N4771

[P0443] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, David Hollman, Lee Howes, Kirk Shoop, Eric Niebler
A Unified Executors Proposal for C++
https://wg21.link/P0443

[P1028] Douglas, Niall
SG14 `status_code` and standard `error` object
https://wg21.link/P1028

[P1031] Douglas, Niall
Low level file i/o
https://wg21.link/P1031

[P1341] Baker, Lewis
Unifying Asynchronous APIs in the C++ standard library https://wg21.link/P1341

[P1750] Klemans Morgenstern, Jeff Garland, Elias Kosunen, Fatih Bakir
A Proposal to Add Process Management to the C++ Standard Library https://wg21.link/P1750

[P1792] Kohlhoff, Christopher
Simplifying and generalising Sender/Receiver for asynchronous operations https://wg21.link/P1792

[P1883] Douglas, Niall
`file_handle` *and* `mapped_file_handle`
https://wg21.link/P1883

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