P1030R3: `std::filesystem::path_view`

Document #:	P1030R3
Date:	2019-09-26
Project:	Programming Language C++
	Library Evolution Working Group
Reply-to:	Niall Douglas
	<s_sourceforge@nedprod.com>

A proposal for a `std::filesystem::path_view`, a non-owning view of explicitly unencoded or encoded character sequences in the format of a local filesystem path, or a view of a binary key.

A mostly-conforming reference implementation of the proposed path view can be found at `https://github.com/ned14/llfio/blob/master/include/llfio/v2.0/path_view.hpp`. It has been found to work well on recent editions of GCC, clang and Microsoft Visual Studio, on x86, x64, ARM and AArch64. It has been in production use for several years now.

Changes since R2 due to LEWG and SG16 Unicode feedback:

- A new `path_view_component` prevents Ranges getting confused when iterating a path view.
- `char` source has been restored, it is the narrow system encoding.
- `byte` input has had its specification strengthened.
- Peeking off the end of input has been removed, now construction supplies whether input is zero terminated or not.
- `c_str` can now generate many renditions of the path view via template parameter, and the relationship to the filesystem native encoding has been weakened.
- Relative comparison operator overloads have been removed, as comparison is very expensive, and anyone using path views in say a `std::map` should always define a custom comparator (which is more efficient).
- Equality comparisons are now identity-based instead of lexicographic.
- After many, many exchanges with SG16 about how best to tame the evil of comparing filesystem paths, I have come up with a whole new way of doing path view comparison which hopefully ticks everybody's boxes.
- Some asked for visitation of the source data, added.
- Default stack internal buffer size has been reduced to 1Kb characters.

Contents

1 Introduction

In the current C++ standard, the canonical way for supplying filesystem paths to C++ functions which consume file system paths is `std::filesystem::path`. This wraps up `std::filesystem::path::string_type` (= `std::basic_string<Char>`) with a platform specific choice of `Char` (currently Microsoft Windows uses `Char` = `wchar_t`, everything else uses `Char` = `char`) with iterators and member functions which parse the string according to the path delimiters for that platform. For example `std::filesystem::path` on Microsoft Windows might parse this string:

`C:\Windows\System32\notepad.exe`

into:

- `root_name()` = "`C:`"
- `root_directory()` = "`\`"
- `root_path()` = "`C:\`"
- `relative_path()` = "`Windows\System32\notepad.exe`"
- `parent_path()` = "`C:\Windows\System32`"
- `filename()` = "`notepad.exe`"
- `stem()` = "`notepad`"
- `extension()` = "`.exe`"
- `*begin()` = "`C:`"

- *++`begin()` = "`/`" (note the forward, not backward, slash. This is considered to be the name of the root directory)
- *++++`begin()` = "`Windows`"
- *++++++`begin()` = "`System32`" (note no intervening slash)

For every one of these decompositions, a new `path` is returned, which means a new underlying `std::basic_string<Char>`, which means a new memory allocation. In code which performs a lot of path traversal and decomposition, these memory allocations, and the copying of fragments of path around, can start to add up. For example, in [P1031] *Low level file i/o library*, a directory enumeration costs around 250 *nanoseconds* per entry amortised. Each path construction might cost that again. Therefore, for each item enumerated, one *halves* the directory enumeration performance solely due to the choice of `path`, which is why P1031 uses `path_view` instead, and thus can enumerate four million directory items per second, which makes handling ten million item plus directories tractable.

There is also a negative effect on CPU caches of copying around path strings. Paths are increasingly reaching hundred of bytes, as anyone running into the 260 path character limit on Microsoft Windows can testify[1]. Every time one copies a path, one is evicting potentially useful data from the CPU caches, which need not be evicted if one did not copy paths.

Enter thus the proposed `std::filesystem::path_view`, which is a lightweight reference to part, or all of, a source of filesystem path data. It provides most of the same member functions as `std::filesystem::path`, operating by constant and often constexpr reference upon some character source which is in the format of the local platform's file system path, or a generic path, same as with `std::filesystem::path`. It is intended that for most functions currently accepting a `std::filesystem::path`, they can now accept a `std::filesystem::path_view` instead with minor to none refactoring of implementation.

2 Impact on the Standard

The proposed library is a pure-library solution.

3 Proposed Design

Much of the proposed path view is unsurprising, with a large subset of `std::filesystem::path`'s observers and modifiers replicated (apart from `path`'s mutating functions, which here are non-mutating and return new views instead). Constexpr abounds, and the path view is trivially copyable and is thus suitable for passing around by value.

WG21 feedback suggested that iteration of path views ought to not return another path view, so iteration returns `path_view_component` instead. I appreciate that this is a large divergence from filesystem path, however feedback suggests that filesystem path is deficient in this regard.

[1] You can now build your Windows application with this limit removed for your program.

Path views represent a user unknown polymorphic view of characters or bytes. The proposed supported path source encodings are:

1. `char`, the narrow native system encoding.
2. `wchar_t`, the wide native system encoding.
3. `char8_t`, UTF-8 encoding.
4. `char16_t`, UTF-16 encoding.
5. `byte`, raw encoded or unencoded bytes. This can mean 'passthrough' for some consumers of path views, or may take on some other meaning depending on consumer.

3.1 `path_view_component`

Path view components look very much like path views, but do not offer path component iteration, nor any of the path interpretation member functions based upon the filesystem path separator.

```
class path_view_component
{
public:
  //! True if path views can be constructed from this character type.
  //! i.e. is one of 'char', 'wchar_t', 'char8_t', 'char16_t'
  template <class Char> static constexpr bool is_source_chartype_acceptable;

  //! True if path views can be constructed from this source.
  //! i.e. 'is_source_chartype_acceptable', or is 'byte'
  template <class Char> static constexpr bool is_source_acceptable;

  //! The default internal buffer size used by 'c_str'.
  static constexpr size_t default_internal_buffer_size = 1024;  // 2Kb for wchar_t, 1Kb for char

public:
  path_view_component() = default;
  path_view_component(const path_view_component &) = default;
  path_view_component(path_view_component &&) = default;
  path_view_component &operator=(const path_view_component &) = default;
  path_view_component &operator=(path_view_component &&) = default;
  ~path_view_component() = default;

  //! True if empty
  [[nodiscard]] constexpr bool empty() const noexcept;
  constexpr bool has_stem() const noexcept;
  constexpr bool has_extension() const noexcept;

  //! Returns the size of the view in characters.
  constexpr size_t native_size() const noexcept;

  //! Swap the view with another
  constexpr void swap(path_view_component &o) noexcept;

  // True if the view contains any of the characters '*', '?', (POSIX only: '[' or ']').
  constexpr bool contains_glob() const noexcept;

```

```
//! Returns a view of the filename without any file extension
constexpr path_view_component stem() const noexcept;

//! Returns a view of the file extension part of this view
constexpr path_view_component extension() const noexcept;

//! Return the path view as a path. Allocates and copies memory!
filesystem::path path() const;

/*! Compares the two path views for equivalence or ordering using `T`
as the destination encoding, if necessary.

If the source encodings of the two path views are compatible, a
lexicographical comparison is performed. If they are incompatible,
either or both views are converted to the destination encoding
using `c_str<T, Delete, _internal_buffer_size>`, and then a
lexicographical comparison is performed.

This can, for obvious reasons, be expensive. It can also throw
exceptions, as `c_str` does.

If the destination encoding is `byte`, `memcmp()` is used,
and `c_str` is never invoked as the two sources are byte
compared directly.
*/
template <class T = typename filesystem::path::value_type
          class Deleter = std::default_delete<T[]>,
          size_t _internal_buffer_size = default_internal_buffer_size
>
requires(path_view_component::is_source_acceptable<T>)
constexpr int compare(const path_view_component &p) const;

//! \overload
template <class T = typename filesystem::path::value_type
          class Deleter = std::default_delete<T[]>,
          size_t _internal_buffer_size = default_internal_buffer_size,
          class Char
>
requires(path_view_component::is_source_acceptable<T> && path_view_component::
    is_source_chartype_acceptable<Char>)
constexpr int compare(const Char *s) const;

//! \overload
template <class T = typename filesystem::path::value_type
          class Deleter = std::default_delete<T[]>,
          size_t _internal_buffer_size = default_internal_buffer_size,
          class Char
>
requires(path_view_component::is_source_acceptable<T> && path_view_component::
    is_source_chartype_acceptable<Char>)
constexpr int compare(const basic_string_view<Char> s) const;

/*! Instantiate from a `path_view_component` to get a path suitable for feeding to other code.

\tparam T The destination encoding required.
\tparam Deleter A custom deleter for any temporary buffer.
```

```
\tparam _internal_buffer_size Override the size of the internal temporary buffer, thus
reducing stack space consumption (most compilers optimise away the internal temporary buffer
if it can be proved it will never be used). The default is 1024 values of `T`.

This makes the input to the path view component into a destination format suitable for
consumption by other code. If the source has the same format as the destination, and
the zero termination requirements are the same, the source is used directly without
memory copying nor reencoding.

If the format is compatible, but the destination requires zero termination,
and the source is not zero terminated, a straight memory copy is performed
into the temporary buffer.

`c_str` contains a temporary buffer sized according to the template parameter. Output
below that amount involves no dynamic memory allocation. Output above that amount calls
`operator new[]`. You can use an externally supplied larger temporary buffer to avoid
dynamic memory allocation in all situations.
*/
template <class T = typename filesystem::path::value_type,
          class Deleter = std::default_delete<T[]>,
          size_t _internal_buffer_size = default_internal_buffer_size
>
struct c_str
{
  static_assert(is_source_acceptable<T>, "path_view_component::c_str<T> does not have a T which is
      one of byte, char, wchar_t, char8_t nor char16_t");

  //! Type of the value type
  using value_type = T;
  //! Type of the deleter
  using deleter_type = Deleter;
  //! The size of the internal temporary buffer
  static constexpr size_t internal_buffer_size = (_internal_buffer_size == 0) ? 1 :
      _internal_buffer_size;

  //! Number of values, excluding zero terminating char, at buffer
  size_t length{0};
  //! Pointer to the possibly-converted path
  const value_type *buffer{nullptr};

public:
  /*! Construct, performing any reencoding or memory copying required.

  \param view The path component view to use as source.
  \param no_zero_terminate Set to true if zero termination is not required.
  \param allocate A callable with prototype `value_type *(size_t length)` which
  is defaulted to `return new value_type[length];`. You can return `nullptr` if
  you wish, the consumer of `c_str` will see a `buffer` set to `nullptr`.

  If an error occurs during any conversion from UTF-8 or UTF-16, an exception of
  `system_error(errc::illegal_byte_sequence)` is thrown.

  This is because if you tell `path_view` that its source is UTF-8 or UTF-16, then that
  must be **valid** UTF. If you wish to supply UTF-invalid paths (which are legal
  on most filesystems), use native narrow or wide encoded source, or binary.
  */
```

```
    template <class U>
    c_str(const path_view_component &view,
          bool no_zero_terminate,
          U &&allocate);

    //! \overload
    c_str(const path_view_component &view,
          bool no_zero_terminate = false);

    ~c_str() = default;
    c_str(const c_str &) = delete;
    c_str(c_str &&) = delete;
    c_str &operator=(const c_str &) = delete;
    c_str &operator=(c_str &&) = delete;

  private:  // For exposition only ...
    bool _call_deleter{false};
    Deleter _deleter;

    // MAKE SURE this is the final item in storage, the compiler will elide the storage
    // under optimisation if it can prove it is never used.
    value_type _buffer[internal_buffer_size]{};
  };
};

// These are IDENTITY equality comparisons i.e. equality is same source encoding, same content
inline constexpr bool operator==(path_view_component x, path_view_component y) noexcept;
inline constexpr bool operator!=(path_view_component x, path_view_component y) noexcept;

inline std::ostream &operator<<(std::ostream &s, const path_view_component &v);

// relative comparison disabled
// hashing disabled

// Visitation of source representation, calls f(const T *, size_t, bool)
template<class F>
inline constepxr auto visit(F &&f, path_view_component);
```

3.2 path_view

```
class path_view
{
public:
  //! Const iterator type
  using const_iterator = path_view_iterator;
  //! iterator type
  using iterator = const_iterator;
  //! Reverse iterator
  using reverse_iterator = std::reverse_iterator<iterator>;
  //! Const reverse iterator
  using const_reverse_iterator = std::reverse_iterator<const_iterator>;
  //! Size type
  using size_type = std::size_t;
  //! Difference type
```

```
  using difference_type = std::ptrdiff_t;

  //! The preferred separator type
  static constexpr auto preferred_separator = filesystem::path::preferred_separator;

public:
  path_view() = default;
  path_view(const path_view &) = default;
  path_view(path_view &&) = default;
  path_view &operator=(const path_view &) = default;
  path_view &operator=(path_view &&) = default;
  ~path_view() = default;

  //! Implicitly constructs a path view from a path. The input path MUST continue to
  //! exist for this view to be valid.
  path_view(const filesystem::path &v) noexcept;

  //! Implicitly constructs a path view from a path view component. The input path
  //! MUST continue to exist for this view to be valid.
  path_view(path_view_component v) noexcept;

  //! Implicitly constructs a path view from a zero terminated ‘const char *‘.
  //! The input string MUST continue to exist for this view to be valid.
  constexpr path_view(const char *v) noexcept;

  //! Implicitly constructs a path view from a zero terminated ‘const wchar_t *‘.
  //! The input string MUST continue to exist for this view to be valid.
  constexpr path_view(const wchar_t *v) noexcept;

  //! Implicitly constructs a path view from a zero terminated ‘const char8_t *‘.
  //! The input string MUST continue to exist for this view to be valid.
  constexpr path_view(const char8_t *v) noexcept;

  //! Implicitly constructs a path view from a zero terminated ‘const char16_t *‘.
  //! The input string MUST continue to exist for this view to be valid.
  constexpr path_view(const char16_t *v) noexcept;


  /*! Constructs a path view from a lengthed array of one of
  ‘byte‘, ‘char‘, ‘wchar_t‘, ‘char8_t‘ or ‘char16_t‘. The input
  string MUST continue to exist for this view to be valid.
  */
  template<class Char>
  requires(path_view_component::is_source_acceptable<Char>)
  constexpr path_view(const Char *v,
                      size_t len,
                      bool is_zero_terminated) noexcept;

  /*! Constructs from a basic string if the character type is one of
  ‘char‘, ‘wchar_t‘, ‘char8_t‘ or ‘char16_t‘.
  */
  template<class Char>
  requires(path_view_component::is_source_chartype_acceptable<Char>)
  constexpr path_view(const std::basic_string<Char> &v) noexcept;

  /*! Constructs from a basic string view if the character type is one of
```

```
  ‘char‘, ‘wchar_t‘, ‘char8_t‘ or ‘char16_t‘.
  */
  template<class Char>
  requires(path_view_component::is_source_chartype_acceptable<Char>)
  constexpr path_view(basic_string_view<Char> v,
                      bool is_zero_terminated) noexcept;


  //! Swap the view with another
  constexpr void swap(path_view &o) noexcept;

  //! True if empty
  [[nodiscard]] constexpr bool empty() const noexcept;
  constexpr bool has_root_path() const noexcept;
  constexpr bool has_root_name() const noexcept;
  constexpr bool has_root_directory() const noexcept;
  constexpr bool has_relative_path() const noexcept;
  constexpr bool has_parent_path() const noexcept;
  constexpr bool has_filename() const noexcept;
  constexpr bool has_stem() const noexcept;
  constexpr bool has_extension() const noexcept;
  constexpr bool is_absolute() const noexcept;
  constexpr bool is_relative() const noexcept;

  // True if the path view contains any of the characters ‘*‘, ‘?‘, (POSIX only: ‘[‘ or ‘]‘).
  constexpr bool contains_glob() const noexcept;

#ifdef _WIN32
  // True if the path view is a NT kernel path starting with ‘\!!\‘ or ‘\??\‘
  constexpr bool is_ntpath() const noexcept;
#endif

  //! Returns an iterator to the first path component
  constexpr inline const_iterator cbegin() const noexcept;
  //! Returns an iterator to the first path component
  constexpr inline const_iterator begin() const noexcept;
  //! Returns an iterator to the first path component
  constexpr inline iterator begin() noexcept;
  //! Returns an iterator to after the last path component
  constexpr inline const_iterator cend() const noexcept;
  //! Returns an iterator to after the last path component
  constexpr inline const_iterator end() const noexcept;
  //! Returns an iterator to after the last path component
  constexpr inline iterator end() noexcept;

  //! Returns a copy of this view with the end adjusted to match the final separator.
  constexpr path_view remove_filename() const noexcept;

  //! Returns the size of the view in characters.
  constexpr size_t native_size() const noexcept;

  //! Returns a view of the root name part of this view e.g. C:
  constexpr path_view root_name() const noexcept;

  //! Returns a view of the root directory, if there is one e.g. /
  constexpr path_view root_directory() const noexcept;
```

```

//! Returns, if any, a view of the root path part of this view e.g. C:/
constexpr path_view root_path() const noexcept;

//! Returns a view of everything after the root path
constexpr path_view relative_path() const noexcept;

//! Returns a view of the everything apart from the filename part of this view
constexpr path_view parent_path() const noexcept;

//! Returns a view of the filename part of this view.
constexpr path_view_component filename() const noexcept;

//! Returns a view of the filename without any file extension
constexpr path_view_component stem() const noexcept;

//! Returns a view of the file extension part of this view
constexpr path_view_component extension() const noexcept;

//! Return the path view as a path. Allocates and copies memory!
filesystem::path path() const;

/*! Compares the two path views for equivalence or ordering using ‘T‘
as the destination encoding, if necessary.

If the source encodings of the two path views are compatible, a
lexicographical comparison is performed. If they are incompatible,
either or both views are converted to the destination encoding
using ‘c_str<T, Delete, _internal_buffer_size>‘, and then a
lexicographical comparison is performed.

This can, for obvious reasons, be expensive. It can also throw
exceptions, as ‘c_str‘ does.

If the destination encoding is ‘byte‘, ‘memcmp()‘ is used,
and ‘c_str‘ is never invoked as the two sources are byte
compared directly.
*/
template <class T = typename filesystem::path::value_type
          class Deleter = std::default_delete<T[]>,
          size_t _internal_buffer_size = path_view_component::default_internal_buffer_size
>
requires(path_view_component::is_source_acceptable<T>)
constexpr int compare(const path_view_component &p) const;

//! \overload
template <class T = typename filesystem::path::value_type
          class Deleter = std::default_delete<T[]>,
          size_t _internal_buffer_size = path_view_component::default_internal_buffer_size,
          class Char
>
requires(path_view_component::is_source_acceptable<T> && path_view_component::is_source_chartype_acceptable<Char>)
constexpr int compare(const Char *s) const;

//! \overload
```

```
  template <class T = typename filesystem::path::value_type
            class Deleter = std::default_delete<T[]>,
            size_t _internal_buffer_size = path_view_component::default_internal_buffer_size,
            class Char
  >
  requires(path_view_component::is_source_acceptable<T> && path_view_component::
      is_source_chartype_acceptable<Char>)
  constexpr int compare(const basic_string_view<Char> s) const;

  //! Instantiate from a ‘path_view‘ to get a path suitable for feeding to other code.
  //! See ‘path_view_component::c_str‘.
  template <class T = typename filesystem::path::value_type,
            class Deleter = std::default_delete<T[]>,
            size_t _internal_buffer_size = path_view_component::default_internal_buffer_size
        >
  struct c_str
};

// These are IDENTITY equality comparisons i.e. equality is same source encoding, same content
inline constexpr bool operator==(path_view x, path_view y) noexcept;
inline constexpr bool operator!=(path_view x, path_view y) noexcept;

inline std::ostream &operator<<(std::ostream &s, const path_view &v);

// relative comparison disabled
// hashing disabled

// Visitation of source representation, calls f(const T *, size_t, bool)
template<class F>
inline constepxr auto visit(F &&f, path_view);
```

3.3 Example of use

The use idiom would be as follows:

```
int open_file(path_view path)
{
  // I am on POSIX which requires zero terminated char filesystem paths.
  // So here if the view is zero terminated, and the view refers to
  // char*, char8_t* or byte data, we can use it directly without memory copying.
  path_view::c_str<> p(path);
  return ::open(p.buffer, O_RDONLY);
}
```

4 Design decisions, guidelines and rationale

There are a number of non-obvious design decisions in the proposed path view object. These decisions were taken after a great deal of empirical trial and error with ‘more obvious’ designs, where those designs were found wanting in various ways. The author believes that the current set of tradeoffs is close to the ideal set.

The design imperatives for an allocating `std::filesystem::path` are not those for a non-allocating `std::filesystem::path_view`. A 'handy feature' of an allocating path object is that it must always copy its input into its allocation. If it is allocating memory and copying the path content in any case, performing an implicit conversion of a native narrow input encoding to say a native wide encoding seems like a reasonable design choice, given the relative cost of the other overheads.

In the case of a path view however, we are trying very hard to not copy memory. If the local platform uses the same narrow or wide input encoding as the source backing the view, and the path view is already terminated by a null character where that is relevant on the local platform, no copying is required. The original is used unmodified, bytes are passed through as-is. Only if necessary, a copy and/or conversion of the input onto the stack is performed into whatever format the local platform requires.

One might argue that in the case of `std::filesystem::path`, we might reuse the path across multiple calls, and thus the path view approach of just-in-time copying per syscall is wasteful on those platforms. However it is exceeding rare to open the same file more than once, and anyone caring strongly about performance will simply modify their source to use the same native encoding and null termination as the platform.

The next argument is usually one of the form that paths get commonly reused with just the leafname modified, and therefore path's approach is more efficient as only the leafname gets converted per iteration. I would counter that this proposed path view object comes from [P1031] *Low level file i/o library* where using absolute paths is bad form: you use a `path_handle` to indicate the base directory and supply a path view for the leafname – this is *far* more efficient than any absolute path based mechanism as it avoids the kernel having to traverse the filesystem hierarchy, typically taking a read lock on each inode in the absolute path.

4.1 Fixed use of stack in `struct c_str`

Firstly, note that the compiler elides completely the fixed stack buffer for zero termination and UTF conversions caused by instantiating `struct c_str` if the compiler can prove that it will never be used. So if you supply native format, zero terminated input, to the path view constructor, the compiler should spot that the temporary stack buffer is never used, and thus eliminate it. This ought to be the case most of the time, especially under link time optimisation.

Secondly, the fixed stack buffer tends to get allocated just before a syscall, and released just after that syscall. Stack cache locality is therefore generally unaffected, and the fixed stack buffer does not remain allocated for long. It is thus a once-off stack allocation.

Microsoft Windows systems can have a maximum path of 64Kb, but most paths are likely to be under 1024 codepoints. Of the major POSIX implementations, `PATH_MAX` is 4096 for Linux, MacOS 1024, FreeBSD 1024. All this suggests that a reasonable default for the internal buffer ought to be 1024 codepoints, which is one of 1Kb, 2Kb or 4Kb of stack consumption depending on what the path view consumer is rendering to.

Thus, on Microsoft Windows and Linux only, if the input path exceeds 1024 codepoints, `malloc()` will be used to create a temporary internal buffer. On MacOS and FreeBSD, the internal buffer is always large enough.

For those Linux implementations running on embedded systems where 1Kb stack allocations would be unwise, we do provide for the ability to choose a smaller fixed size buffer in the template parameter, and override the custom allocator to issue a trap if the smaller path limit is exceeded.

Again, I would stress that the programmer can be careful to never send a non-zero terminated string in as a path, and thus completely eliminate the use of temporary buffers on an embedded Linux solution. In any case, path views are considerably less heavy on free RAM than `std::filesystem::path`.

4.2 Path view consumer determines the path interpretation semantics

Path view has been designed around the *consumer* defining what reencoding semantics are in play. For example, a Java JNI might define UTF-16 as the destination encoding for `c_str` irrespective of the native filesystem encoding or platform, and all input is therefore converted to UTF-16 by the JNI's use of `c_str`. This is why `.compare()` is templated exactly as `c_str` is templated, and it is on whomever consumes filesystem path views to define a locally customised `path_view` if the defaults are inappropriate for their use case.

Some have asked for detailed reencoding semantics for the filesystem. Here are those as defined by [P1031] *Low level file i/o*, but let me stress once again that it is the *consumer* of path views which defines how path views are to be interpreted.

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

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

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

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

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

- `char` = Compatibility DOS format paths, narrow system encoding (program locale determined). Compatibility DOS format paths start with `X:\`, or no prefix at all. These call the ANSI editions of Win32 APIs.
- `wchar_t` = Compatibility DOS format paths, wide system encoding (UTF-16). These call the Unicode editions of Win32 APIs.
- `char16_t` = Compatibility DOS format paths in UTF-16. Input must be valid UTF-16. These call the Unicode editions of the Win32 APIs.
- `char8_t` = Compatibility DOS format paths in UTF-8. Input must be valid UTF-8. This is converted C++-side to UTF-16 at the point of use, and the Unicode editions of Win32 APIs are called.

- `char` = Extended DOS format paths, narrow system encoding (program locale determined). As per Win32 API documentation, extended DOS format paths are prefixed with `\\?\` or `\\.\`. These call the ANSI editions of Win32 APIs.
- `wchar_t` = Extended DOS format paths, wide system encoding (UTF-16). These call the Unicode editions of Win32 APIs.
- `char16_t` = Extended DOS format paths in UTF-16. Input must be valid UTF-16. These call the Unicode editions of the Win32 APIs.
- `char8_t` = Extended DOS format paths in UTF-8. Input must be valid UTF-8. This is converted C++-side to UTF-16 at the point of use, and the Unicode editions of Win32 APIs are called.

- `char` = NT format paths, narrow system encoding (program locale determined). This is a LLFIO-only extension, NT format paths are prefixed with `\!!\`. Paths prefixed with this never use the Win32 APIs, only the NT kernel APIs.
- `wchar_t` = NT format paths, wide system encoding (UTF-16).
- `char16_t` = NT format paths in UTF-16. Input must be valid UTF-16.
- `char8_t` = NT format paths, UTF-8. This is converted C++-side to UTF-16 at the point of use.

- `byte` = Unique variable width binary number identifier (NTFS and ReFS permit a 128-bit key-value lookup of inodes, this may be accelerated in hardware by suitable storage devices).

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

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

5 Technical specifications

No Technical Specifications are involved in this proposal.

6 Frequently asked questions

6.1 Does this mean that all APIs consuming `std::filesystem::path` ought to now consume `std::filesystem::path_view` instead?

Most of the time, perhaps almost always, yes. `std::filesystem::path_view` implicitly constructs from explicitly encoded strings, paths and explicitly encoded string literals. Anywhere you are currently consumg `std::filesystem::path` as a parameter, you can start using `std::filesystem::path_view` instead if this proposal is approved. It would remain the case that where a function is returning a new path, `std::filesystem::path` is the correct choice. So inputs would be mostly path views, outputs would be paths.

Path views can represent more encodings of filesystem paths than `std::filesystem::path` can e.g. unique variable with binary numbers.

This author has replaced paths with path views in an existing piece of complex path decomposition and recomposition, and apart from a few minor source code changes to fix lifetime issues, the code compiled and worked unchanged. Path views are mostly a drop-in replacement for paths, except for when one is creating wholly new paths.

Incidentally, performance of that code improved by approximately twenty fold (20x).

7 Acknowledgements

My thanks to Nicol Bolas, Bengt Gustafsson and Billy O'Neal for their feedback upon this proposal.

8 References

[P0482] Tom Honermann,
char8_t: A type for UTF-8 characters and strings
https://wg21.link/P0482

[P0882] Yonggang Li
User-defined Literals for std::filesystem::path
https://wg21.link/P0882

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