D1750R0 A proposal to add process management to the C++ standard library

Klemens Morganstern, Jeff Garland, Elias Kosunen, Fatih Bakir

June 16, 2019

Contents

1 Revision History

1.1 Revision 0

Initial release.

2 Introduction

The creation and management of processes is a widely used and fundamental tool in computing systems. Unfortunately, C++ does not have a portable way to create and manage processes. Most other language standard libraries support facilities to wrap the complexities in support of application programmers. The functionality has been on standard library wishlists going back to 2007. The proposal is based on boost.process which provides cross-platform implementation experience.

3 Motivation and Scope

We propose a library to facilitate the following functionality:

- create child processes on the current machine
- setup streams for communication with child stdout and stderr
- communicate with child processes through streams
- wait for processes to exit
- terminate processes
- capture the return result of a child process
- manage groups of processes
- optionally associate the child-process to the parent-child to children die with their parents, but not vice-versa.

The following illustrates an example usage of the proposed library.

```
#include <process>

int main() {
  std::vector<std::string> args { "--version", "--std=c++2a" };

  try {
    std::ipstream pipe_stream;

    //capture stdout, leave the others as the OS defaults
    std::process child("gcc", args, std::process_io({}, pipe_stream, {}));

    std::string line;

    while (pipe_stream && std::getline(pipe_stream, line) && !line.empty()) {
      std::cerr << line << std::endl;
    }
    child.wait();
  }
  catch(const std::process_error& e) {
    std::cerr << e.what();
  }
}
```

Additional user examples can be seen online.

4 Domain Terminology

4.1 Processes

A process is an instance of a program in execution. A process has at least one thread. A process starts execution in the thread that invokes its main function. A child process is the result of another process creating or spawning the child.

4.2 Process Groups

Process groups allow for managing a set of processes at the operating system level. This allows behavior such as process termination to be automatically coordinated by the operating system. For example, child processes in a group can be set to terminate together.

4.3 Pipes

A pipe is a unidirectional, serial communication line across processes. A pipe has two ends: a write end and a read end.

A pipe is buffered. The size of the buffer is implementation defined. When there's no data in the buffer, the pipe is called empty. When the buffer is full, the pipe is called full.

Reading from an empty pipe is a blocking operation. Writing to a pipe resumes any blocked threads that are waiting to read on that pipe.

Writing to a full pipe is a blocking operation. Reading from a pipe resumes any blocked threads that are writing to that pipe.

If there are multiple threads reading or writing from the same pipe at the same time the order in which they read the data is unspecified.

4.4 Environment and Command Line Arguments

Creation of a child process sometimes involves modifying the environment for the child process. This proposal references a current proposal for referencing a process environment. However, the proposal P1275 (see References in the bottom) would need to be enhanced to support multiple instances of environments for access and modification of child process environment.

This proposal currently contains a synopsis of similar functionality.

5 Survey of facilities in other standard libraries

5.1 C/C++ system function

C and C++ currently provide a minimal process launching capability via the `system` function. The C++ function takes a `const char*` parameter that represents the command string to execute and an integer return code that signifies the execution return status.

```
int result = system("echo \"foo\" > bar.txt");
if (result == 0) {
  //success
}
```

This minimal facility lacks many aspects of process control needed for even basic applications, including access to the standard streams (stdin, stdout, stderr) of the child.

In addition it uses the system shell to interpret the command, which is a huge security hazard because of shell injection.

5.2 Java

Java provides a ProcessBuilder and stream piping facilities similar to what is proposed here.

```
// ProcessBuilder takes variadic string arguments
// or a List<String>
var builder = new ProcessBuilder("/bin/cat", "-");

// start()-method will spawn the process
// Standard streams are piped automatically
Process p = builder.start();

// Write into process stdin
new OutputStreamWriter(p.getOutputStream())
      .write("foo\n")
      .close(); // close() needed to flush the buffer

// Read from stdout
var reader = new BufferedReader(
                  new InputStreamReader(p.getInputStream()));
String output = reader.readLine();

assert output == "foo";

System.out.println("Exited with " + p.exitValue())
```

5.3 Python

```
from subprocess import run

# Command line arguments are all passed in a single list
# Standard streams aren't piped by default
result = run([ '/bin/cat', '-' ],
          input='foo\n', capture_output=True)
assert result.stdout == 'foo'
print("Exited with", result.returncode)
```

5.4 Rust

As with other languages Rust provides the ability to pipe the results of the process into the parent.

```
use std::process::{Command, Stdio};

let mut child = Command("/bin/cat")
                .arg("-") // .args() also available, taking a range
                          // strings passed to .arg() are escaped
                .stdin(Stdio::piped())
                .stdout(Stdio::piped())
                .spawn()?; // ?-operator is for error handling
child.stdin.as_mut()?.write_all(b"foo\n")?;
// .wait_with_output() will, well, wait
// child.stdout/stderr exposes standard streams directly
let output = child.wait_with_output()?;
assert_eq!(b"foo", output.stdout.as_slice());
println!("Exited with {}", output.status.code.unwrap());
```

5.5 Nodejs

```
const { spawn } = require('child_process');

// First argument is argv[0], rest of argv passed in a list
const p = spawn('/bin/cat', ['-']);
p.stdin.write('foo\n');
// Idiomatic node.js uses callbacks everywhere
p.stdout.on('data', (data) => {
  assert.StrictEqual(data, 'foo\n');
});
p.on('close', (code) => {
  console.log(`Exited with ${code}`);
});
```

6 Design Discussion & Examples

6.1 Concept `ProcessLauncher`

The process launcher is a class that impements the actual launch of a process. In most cases there are different versions to do this. On linux for example, `vfork` can be required as an alternative for fork on low-memory devices. And while posix can change a user by utilizing setuid in a ProcessInitializer, windows requires the invocation of a different function (CreateProcessAsUserA).

As an example for linux:

```
#include <gnu_cxx_process>

__gnu_cxx::vfork_launcher launcher;
std::process my_proc("/bin/programm", {}, launcher);
```

or for windows:

```
__msvc::as_user_launcher{"1234-is-not-a-safe-user-token"};
std::process my_proc("C:\program", {}, launcher);
```

In addition libraries may provide their launchers. The requirement is that there is an actual process with a pid as the result of launching the process.

Furthermore, the fact that the launcher has a well-specified `launch` function allows to launch a process like this:

```
std::process_launcher launcher;
auto proc = launcher.launch("/bin/foo", {});
```

Both versions make sense in their own way: on the one hand using the process constructor fits well in with the STL and it's RAII classes like thread. On the other hand it actually uses a factory-class, which can be used so explicitly.

6.2 Concept `ProcessInitializer`

The process initializer is a class that modifies the behaviour of a process. There is no guarantee that a custom initializer is portable, i.e. it will not only be dependable on the operating system but also on the process launcher. This is because an initializer might need to modify members of the launcher itself (common on windows) and thus might break with another launcher.

Note that the concept might look different on other implementation, since additional event hooks might exist.

```
struct switch_user {
```

```
  ::uid_t uid;

  template<ProcessLauncher Launcher>
  // Linux specific event, after the successful fork, called from the child process
  void on_exec_setup(Launcher&) {
    ::setuid(this->uid);
  }
};

std::process proc("/bin/spyware", {}, switch_user{42});
```

6.3 Class process

6.3.1 Constructor

`process(std::filesystem::path, std::ranges::Range<String>, Inits...init)`

This is the default launching function, and forwards to the `std::process_launcher`. boost.process supports a cmd-style execution (similar to `std::system`), which we opted to remove from this library. This is because the syntax obscures what the library actually does, while introducing a security risk (shell injection). Instead we require the actually used (absolute) path of the executable. Since it is common to just type a command and expect the shell to search for the executabe in the `PATH` environment variable, there is a helper function for that. Either in the `std::environment` class or the `std::this_process::environment` namespace.

```
std::system("git --version"); // Launches to cmd or /bin/sh

std::process("git", {"--version"}); // Does err, exe not found
std::process(std::this_process::environment::find_executable("git"), {"--version"}); // Finds the exe

// Or if we want to run it through the shell, note that /c is windows specific
std::process(std::this_process::environment::shell(), {"/c", "git --version"});
```

Another solution is for a user to provide their own `ProcessLauncher` as a `ShellLauncher`.

6.3.2 function `wait`

The wait function waits for a process to exit. When replacing `std::system` it can be used like this:

```
const auto result_sys = std::system("gcc --version");

std::process proc(std::this_process::environment::find_executable("gcc"), {"--version"});
proc.wait();
const auto result_proc = proc.exit_code();
```

6.3.3 function `wait_for`

In case the child process might hang, `wait_for` might be used.

```
std::process proc(std::this_process::environment::find_executable("python"), {"--version"});

int res = -1;
if (proc.wait_for(std::chrono::seconds(1))
    res = proc.exit_code();
else
    proc.terminate();
```

6.3.4 function `native_handle`

Since there is a lot funcitonality that is not portable, the `native_handle` is accessible. For example, there is no clear equivalent for `SIGTERM` on windows. If a user still wants to use this, he might still do so:

```
std::process proc("/usr/bin/python", {});

::kill(proc.native_handle(), SIGTERM);
proc.wait();
```

6.3.5 function `native_exit_code`

The exit-code may contain more information on a specific system. Practically this is the case on posix. If a user wants to extract additional information he might need to use `native_exit_code`.

```
std::process proc(std::this_process::environment::find_executable("gcc"), {});
proc.wait();
const auto exit_code = proc.exit_code(); //equals 1, since no input files

//Linux specific:
const exited_normally = WIFEXITED(proc.native_exit_code();
```

6.3.6 function `async_wait`

To allow asynchronous operations, the process library shall integrate with the networking TS.

```
extern std::net::system_executor exec;
std::process proc(std::this_environment::find_executable("gcc"), {});

auto fut = proc.async_wait(exec, std::net::use_future_t());
const exit_code = fut.get();
assert(exit_code == proc.exit_code());
```

6.4 Class `process_group`

The process group can be used for managing several processes at once. Because of the underlying implementation on the OS, there is no guarantee that the exit-code for a process can be obtained. Because of this the ‘wait_{one}‘ and related functions do not yield the $\text{exit}_{\text{code}}$ or pid. To make workaround easy, the launch function returns the pid, so a user can write his own code.

6.4.1 Example: Attaching a debugger to a process and grouping them.

```
std::process_group grp;

auto pid_target = grp.emplace("./my_proc", {});
auto pid_gdb = grp.emplace("/usr/bin/gdb", {-p, std::to_string(pid_target)});
// Do something

// Kill gdb and use the process individually:

grp.detach(pid_gdb).terminate();
std::process target = grp.detach(pid_target);
```

6.4.2 Example: Having a worker, e.g. for a build system

```
void run_all(const std::queue<std::pair<std::filesystem::path, std::vector<std::string>>>& jobs, int pa
  std::process_group grp;
  for (auto idx = 0u; (idx < parallel) && !jobs.empty(); idx++) {
    const auto [exe, args] = jobs.front();
    grp.emplace(exe, args);
    jobs.pop();
  }

  while (!jobs.empty()) {
    grp.wait_one();
    const auto [exe, args] = jobs.front();
    grp.emplace(exe, args);
    jobs.pop();
  }
}
```

6.5 Class `process_io`

`process_io` takes the three standard handles, because some OS (windows that is) does not allow to cherry-pick those. Either all three are set or all are defaults.

The default of course is to forward it to `std*`.

6.5.1 Using pipes

```
std::pipe pin, pout, perr;
std::process proc("foo", {}, std::process_io(pin, pout, perr));

pin.write("bar", 4);
```

Forwarding between processes:

```
std::system("./proc1 | ./proc2");

std::optional<std::pipe> fwd = std::pipe();

std::process proc1("./proc1", {}, std::process_io({}, *fwd, {});
std::process proc2("./proc1", {}, std::process_io(*fwd, {}, {});

fwd = std::nullopt; // Not needed anymore
```

You can of course use any `pstream` type instead.

6.5.2 Using files

```
std::filesystem::path log_path = std::this_process::environment::home() / "my_log_file";
std::system("foo > ~/my_log_file");
// Equivalent:
std::process proc ("foo", std::process_io({}, log_path, {}));
```

With an extension to fstream:

```
std::ifstream fs{"/my_log_file"};
std::process proc ("./foo", std::process(fs, {}, {});
```

6.5.3 `std::this_process::stdio`

Since `std::cout` can be redirected programmatically and has the same type as `std::cerr` it does not seem like a proper fit, unless it's type is changed

```
// Redirect stderr to stdout
std::process proc ("./foo", std::process_io({}, {}, std::this_process::io().stdout());
```

6.5.4 Closing streams

A closed stream means that the child process cannot read or write from the stream. That is, an attempt to do so yields an error. This can be done by using `nullptr`.

```
std::process proc ("./foo", std::process_io(nullptr, nullptr, nullptr));
```

6.5.5 Other objects

Other objects, that use an underlying stream handle, shall be used. This is the case for tcp sockets (i.e. `std::net::basic_stream_socket`).

```
std::net::tcp::socket sock(...)
// Connect the socket

std::process proc ("./server", std::process_io(socket, socket, "log-file"));
```

6.5.6 Null device (not yet specified)

The null-device is a a feature of both posix ("/dev/null") and windows ("NUL"). It accepts and write, and always returns It might be worth considering adding it.

```
std::system("./foo > /dev/null");

std::process proc("./foo", {}, std::process_io(std::process_io::null(), std::process_io::null(), std::p
```

6.6 Class `environment`

6.6.1 `operator[]`

Unlike Muerte's proposal, ours does not contain an `operator[]`. The reason is that environment variables are not uniform on their handling of case-sensitivity. For example `"PATH"` might be `"Path"` between different versions of windows. However, both maybe defined on windows. This can cause a problem:

```
std::environment env = std::this_process::environment::native_environment();

// Let's say it's "Path", but we expect "PATH"
env["PATH"].add_value("C:\\python");
std::process proc (env.find_executable("python"), {"./my_script.py"}, env); //error -> python not found
```

If we however have it explicitly, things are more obvious.

```
std::string to_upper(const std::string& in); // Implement a conversion to upper case here

auto keys = env.keys();
auto path_key = std::find_if(keys.begin(), keys.end(), [](auto& str) { return to_upper(str) == "PATH";
```

```
auto entry = env.get(path_key);

auto val = entry.as_vector();
val.push_back("C:\\python");
env.set(path_key, val);
```

6.6.2 Function `environment::home()`

This should be it's own function, because it is one value on posix (`"HOME"`) but two on windows (`"HOME_DRIVE"`, `"HOME_DIR"`).

6.6.3 Function `environment::extensions`

This environment variable is only used on systems that use file extensions to determine executables (i.e. windows).

```
// Assume /home/hello_world.py is the executable and "/home" is in PATH already
// --> It names hello_world on linux, hello_world.py on windows.
std::environment env = std::this_process::environment::native_environment();
auto extensions = env.extensions();

std::process proc;

// We can use find_executable on linux only if the file does not have the syntax.
// This is in accordance with the shell rules
if (std::find(extensions.begin(), extensions.end(), ".py")) {
  proc = std::process(env.find_executable("hello_world"), {});
}
else {
  proc = std::process("/home/hello_world.py", {});
}
```

6.6.4 Function `environment::find_executable`

This function shall find an executable with the name. if the OS uses file extensions it shall compare those, if not it shall check the executable flag.

```
auto pt = std::this_process::environment::find_executable("readme.txt"); //finds a file, but is not exe
assert(pt.empty());
```

7 Design Decisions

7.1 Namespace std versus std::process

The classes and functions for this proposal could be put into namespace `std`, or a sub-namespace, such as `std::process`. Process is more similar to `std::thread` than `std::filesystem`. Since `thread` is in namespace `std` this proposal suggests the same for process. The proposal also introduces namespace `std::this_process` for accessing attributes of the current process environment.

7.2 Using a builder method to create

Have a `run()` method versus immediate launching in the constructor

This is solved through the extended launcher concept.

```
// These are the same:
process(...) : process(process_launcher.launch(...)) {}
```

```
process_launcher().launch(...) -> process;

// These are the same:
process(..., custom_launcher& cl) : process(cl.launch) {}
cl.launch(...);
```

7.3 Handling of parameters

- There's an issue of escaping the argument properly
- see issues below on P1275

7.4 `wait` or `join`

The name of the method in `class process` was discussed at length. The name `join` would be similar to `std::thread` while `wait` is more like various locking classes in the standard. `boost.process` supports both. The decision was to use `wait`, but the name is open to bike shedding.

7.5 Native Operating System Handle Support

The solution provides access to the operating system, like `std::thread`, for programmers who to go beyond the provided facilities.

7.6 pipe close and EOF

Compared to the `boost.process` implementation, this proposal adds classes for different `pipe_ends` and uses C++17 aggregate initialization. The reason is that the following behaviour is not necessarily intuitive:

```
boost::process::pipe p;

boost::process::child c("foo", boost::process::std_in < p);
```

In boost.process this closes the write end of `p`, so an `EOF` is read from `p` when `c` exists. In most cases this would be expected behaviour, but it is far from obvious. By using two different types this can be made more obvious, especially since aggregate initialization can be used:

```
auto [p_read, p_write] = std::pipe();
std::process("foo", std::process_io(p_read));
p_read.close();

p_write.write("data", 5);
```

Note that overloading allows us to either copy or move the pipe, i.e. the given example only moves the handles without duplicating them.

7.7 Security and User Management Implications

`std::system` is dangerous because of shell injection, which cannot happen with the uninterpreted version that is proposed here. A shell might easily still be used by utilizing `std::this_process::environment::shell()`.

The standard process library does not touch on user management. As with file level visibility and user access the resposibility for user permissions lies outside the standard. For example, a process could fail to spawn as a result of the user lacking sufficient permissions to create a child process. This would be reflected as `system_error`.

7.8 Extensibility

To be extensible this library uses two concepts: `ProcessLauncher` and `ProcessInitializer`.

A `ProcessLauncher` is the actual function creating the process. It can be used to provide platform dependent behaviour such as launching a process a new user (Using `CreateProcessAsUser` on Windows) or to use `vfork` on Linux. The vendor can thus just provide a launcher, and the user can then just drop it into their code.

A `ProcessInitializer` allows minor additions, that just manipulate the process. E.g. on Windows to set a `SHOW_WINDOW` flag, or on Linux to change the user with `setuid`.

Not having these customization points would greatly limit the applicability of this library.

JG TODO fix `on_error` On error passes an `std::error_code` to the initializer, so it can react, e.g. free up resources. The launcher must only throw after every initializer was notified.

`on_success`

`on_setup`

7.9 Error Handling

Uses exceptions by throwing a `std::process_error`. `boost.process` has an alternative error code based api similar to `std::filesystem`. Field experience shows little actual usage of this api so it was not included in the proposal.

7.10 Synchronous Versus Asynchronous and Networking TS

Synchronous process management is prone to potential deadlocks. However used in conjunction with `std::thread` and other facilities synchronous management can be useful. Thus the proposal supports both styles.

`boost.process` is currently integrated with `boost.asio` to support asynchronous behaviors. This proposal currently references the Networking TS for this behavior. However, this proposal can be updated to reflect changes to this aspect of the design since the committee is actively working on this design.

7.11 Integration of iostreams and pipes

Pipes bring their own streams, that can be used within a process (e.g. between threads). Thus the proposal provides header `pstream` and the various pipe stream classes as a separate entity.

8 Technical Specification

The following represents a first draft of an annotated technical specification without formal wording. For an initial proposal this is rather extensive, but hopefully clarifies the proposed library scope.

8.1 Header <process> Synopsis

```
#include <chrono>
#include <filesystem>
#include <ranges>
#include <string>
#include <system_error>
#include <vector>

namespace std {
  // A launcher is an object that has a launch function that takes a path,
  // arguments and a variadic list of process initializers and returns a process object.
  template<class T>
  concept bool ProcessLauncher = requires(T launcher) {
    {launcher.set_error(error_code(), "message")} -> void; //so initializers can post internal errors
    {launcher.launch(filesystem::path(), vector<string>())} -> process; //refine that so check tha para
  };

  // The default process-launcher of the impementation
  class process_launcher;
```

```
  // An initializer is an object that changes the behavior of a process during launch
  // and thus listens to at least one of the hooks of the launcher.
  // Note that the following example only uses portable hooks, but non portables might suffice as well
  template<class Init, ProcessLauncher Launcher = process_launcher>
  concept bool ProcessInitializer =
       requires(Init initializer, Launcher launcher) { {initializer.on_setup(launcher)}   -> void; }
    || requires(Init initializer, Launcher launcher) { {initializer.on_success(launcher)} -> void; }
    || requires(Init initializer, Launcher launcher) { {initializer.on_error(launcher, error_code())} -
  }

  // This is to be defined, but should allow any stream that can yield a system-handle
  // (e.g. pipes, files & sockets) and to close the stream by passing nullptr.
  // Additionally a path should be possible to open a file just for the child process.
  concept ProcessReadableStream = TODO;
  concept ProcessWritableStream = TODO;

  // A pid_type is an identifier for a process, that satisfies StrictTotallyOrdered
  using pid_type = implementation-defined;

  // Provides a portable handle to an operating system process
  // process satisfies Movable and Boolean, but not Copyable.
  class process;

  // Exception type thrown on error
  // Can have a filesystem::path attached to it (failing before launch),
  // or pid_type (failing after)
  class process_error;

  // Provides a portable wrapper for a process group
  class process_group;

  // Provides initializers for the standard io.
  class process_io;

  // Provides a way to set the starting directory of the new process.
  class process_start_dir;

  // Satisfies ProcessInitializer
  class environment;

  // Satisfies ProcessInitializer
  class process_limit_handles;
}
```

8.2 Class process

```
namespace std {
  class process {
  public:
    // Provides access to underlying operating system facilities
    using native_handle_type = implementation-defined;

    // Construct a child from a property list and launch it.
    template<ranges::InputRange R, ProcessInitializer... Inits>
      requires ConvertibleTo<iter_value_t<R>, string>
    explicit process(const filesystem::path& exe, const R& args, Inits&&... inits);

    // Construct a child from a property list and launch it with a custom process launcher
    template<ranges::InputRange R, ProcessInitializer... Inits, ProcessLauncher Launcher>
```

```
      requires ConvertibleTo<iter_value_t<R>, string>
    explicit process(const filesystem::path& exe, const R& args, Inits&&... inits, Launcher&& launcher)

    // Attach to an existing process
    explicit process(pid_type& pid);

    // An empty process is similar to a default constructed thread. It holds an empty
    // handle and is a place holder for a process that is to be launched later.
    process() = default;

    // The destructor terminates
    ~process();

    // Accessors

    pid_type id() const;

    native_handle_type native_handle() const;

    // Return code of the process, only valid if !running()
    int exit_code() const;

    // Return the system native exit code.
    // That is on linux it contains the reason of the exit, such as can be detected by WIFSIGNALED
    int native_exit_code() const;

    // Check if the process is running. If the process has exited already, it might store the exit_code
    bool running() const;

    // Check if this handle holds a child process.
    // NOTE: That does not mean, that the process is still running. It only means, that the handle does
    bool valid() const;
    explicit operator bool() const; // Equivalent to this->valid()

    // Process management functions

    // Detach a spawned process -- let it run after this handle destructs
    void detach();

    // Terminate the child process (child process will unconditionally and immediately exit)
    // Implemented with SIGKILL on POSIX and TerminateProcess on Windows
    void terminate();

    // Block until the process to exits
    void wait();

    // Block for the process to exit for a period of time.
    template<class Rep, class Period>
    bool wait_for(const chrono::duration<Rep, Period>& rel_time);

    // wait for the process to exit until a point in time.
    template<class Clock, class Duration>
    bool wait_until(const chrono::time_point<Clock, Duration>& timeout_time);

    //The following is dependent on the networking TS. CompletionToken has the signature (int, error_co
    template<class CompletionToken>
    auto async_wait(net::Executor& ctx, CompletionToken&& token);
  };
}
```

8.3 Class process_error

```
class process_error : public system_error {
public:
  using system_error::system_error;
};
```

8.4 Class process_group

```
namespace std {
  class process_group {
  public:
    // Provides access to underlying operating system facilities
    using native_handle_type = implementation-defined;

    process_group() = default;

    process_group(process_group&& lhs);
    process_group& operator=(process_group&& lhs);

    // The destructor terminates all processes in the group
    ~process_group();

    native_handle_type native_handle() const;

    // Check if at least one process of the group is running
    bool running() const;

    // Check if this handle holds a process group.
    // NOTE: That does not mean, that the process is still running.
    // It only means, that the handle does or did exist.
    bool valid() const;
    explicit operator bool() const; // Equivalent to this->valid()

    // Process management functions

    // Emplace a process into the group, i.e. launch it attached to the group
    template<ranges::InputRange Args, ProcessInitializer... Inits>
      requires ConvertibleTo<iter_value_t<Args>, string>
    pid_type emplace(const filesystem::path& exe, const Args& args, Inits&&...inits);

    // Emplace a process into the group, i.e. launch it attached to the group with a custom process lau
    template<ranges::InputRange Args, ProcessInitializer... Inits, ProcessLauncher Launcher>
      requires ConvertibleTo<iter_value_t<Args>, string>
    pid_type emplace(const filesystem::path& exe, const Args& args, Inits&&...inits, Launcher&& launche

    // Attach an existing process to the group. The process object will be empty afterwads
    pid_type attach(process&& proc);

    // Take a process out of the group
    [[nodiscard]] process detach(pid_type);

    // Detach a process group -- let it run after this handle destructs
    void detach();

    // Terminate the child processes (child processes will unconditionally and immediately exit)
    // Implemented with SIGKILL on POSIX and TerminateProcess on Windows
    void terminate();
```

```
    // Block until all processes exit
    void wait();

    // Block until one process exit
    // NOTE: Windows does not yield information on which process exited.
    void wait_one();

    // Block for all processes to exit for a period of time.
    template<class Rep, class Period>
    bool wait_for(const chrono::duration<Rep, Period>& rel_time);

    // Block for one process to exit for a period of time.
    template<class Rep, class Period>
    bool wait_for_one(const chrono::duration<Rep, Period>& rel_time);

    // Wait for all processes to exit until a point in time.
    template<class Clock, class Duration>
    bool wait_until(const chrono::time_point<Clock, Duration>& timeout_time);

    // Wait for one process to exit until a point in time.
    template<class Clock, class Duration>
    bool wait_until_one(const chrono::time_point<Clock, Duration>& timeout_time);

    // The following is dependent on the networking TS. CompletionToken has
    //the signature (error_code) and waits for all processes to exit
    template<class CompletionToken>
    auto async_wait(net::Executor& ctx, CompletionToken&& token);

    // The following is dependent on the networking TS. CompletionToken has
    // the signature (error_code) and waits for one process
    template<class CompletionToken>
    auto async_wait_one(net::Executor& ctx, CompletionToken&& token);
  };
}
```

8.5 Class process_io

```
namespace std {
  // This class describes I/O redirection for the standard streams (stdin, stdout, stderr).
  // They all are to be set, because Windows either inherits all or all need to be set.
  // Satisfies ProcessInitializer
  class process_io {
  public:
    // OS dependent handle type
    using native_handle = implementation-defined;

    using in_default  = implementation-defined;
    using out_default = implementation-defined;
    using err_default = implementation-defined;

    template<ProcessReadableStream In = in_default,
             ProcessWritableStream Out = out_default,
             ProcessWritableStream Err = err_default>
    process_io(In&& in, Out&& out, Err&& err);

    // Rest is implementation-defined
  };
}
```

8.6 Class process_start_dir

```
namespace std {
  // This class the starting directory for the child process.
  // Satisfies ProcessInitializer
  class process_start_dir {
  public:
    process_start_dir(const filesystem::path&);
  };
}
```

8.7 Class environment

An environment class that can manipulate and query any environment variables. Note that this is not for direct manipulation of the current processes environment, but it satisfies ProcessInitializers

```
namespace std {
  // Satisfies ProcessInitializer
  class environment {
  public:
    using native_environment_type = implementation-defined;

    native_environment_type native_environment();

    // Empty environment
    environment();

    // Construct from a native type, so the current environment can be cloned
    environment(native_environment_type native_environment);

    class entry;

    using value_type = entry;

    // Note that windows uses wchar_t here, the key type should be able to be constructed from a char*
    // So it needs to be similar to filesystem::path
    using key_type   = implementation-defined;
    using pointer    = implementation-defined;

    value_type  get(const key_type& id);
    void        set(const key_type& id, const value_type& value);
    void      reset(const key_type& id);

    // Get all the keys
    // Return type satisfies ForwardRange with iter_value_t convertible to string
    implementation-defined keys() const;

    // Utility functions to query common values

    // Home folder
    filesystem::path home() const;
    // Temporary folder as defined in the env
    filesystem::path temp() const;

    // Shell command, see ComSpec for windows
    filesystem::path shell() const;
```

```
    // The path variable, parsed.
    vector<filesystem::path> path() const;

    // The path extensions, that mark a file as executable (empty on posix)
    vector<filesystem::path> extensions() const;

    // Find an executable file with this name.
    filesystem::path find_executable(const string& name);
  };

  class environment::entry {
  public:
    using value_type = implementation-defined;

    entry();

    entry& operator=(const entry&);
    entry& operator=(entry&&);

    string string();
    wstring wstring();
    value_type native_string();

    // Split according to the OS specifics
    vector<value_type> as_vector();

    entry& operator=(const string&);
    entry& operator=(const wstring&);

    entry& operator=(const vector<value_type>&);
  };
}
```

8.8 Class `process_limit_handles`

This $\text{limit}_{\text{handles}}$ property sets all properties to be inherited only expcitly. It closes all unused file-descriptors on POSIX after the fork and removes the inherit flags on Windows.

Since limit also closes the standard handles unless they are explicitly redirected, they can be ignored by `limit_handles`, through passing in `this_process::stdio()`.

```
namespace std {
  // Satisfies ProcessInitializers
  class process_limit_handles {
  public:
    // Select all the handles that should be inherited even though they are not used by any initializer
    template<typename... Handles>
    process_limit_handles(Handles&&... handles);
  };
}
```

8.9 Namespace `this_process`

This namespace provides information about the current process.

```
namespace std::this_process {
  using native_handle_type = implementation-defined;
```

```
using pid_type = implementation-defined;

// Get the process id of the current process.
pid_type get_id();
// Get the native handle of the current process.
native_handle_type native_handle();

struct stdio_t {
  native_handle_type in();
  native_handle_type out();
  native_handle_type err();
};

// Get the handles to the standard streams
stdio_t stdio();

// Get a snapshot of all handles of the process (i.e. file descriptors on POSIX and handles on Window
// NOTE: This function might not work on certain posix systems.
// NOTE: On Windows version older than Windows 8 this function will iterate all the system handles, m
// NOTE: This functionality is utterly prone to race conditions, since other threads might open or cl
vector<native_handle_type> get_handles();

// Determines if a given handle is a a stream-handle, i.e. any handle that can be used with read and
// Stream handles include pipes, regular files and sockets.
bool is_stream_handle(native_handle_type handle);

// Note that this might also be a global object, i.e. this is yet to be defined.
namespace environment {
  using native_environment_type = implementation-defined;
  native_environment_type native_environment();

  using value_type = entry;
  using key_type   = implementation-defined; // Note that Windows uses wchar_t here, the key type sho
  using pointer    = implementation-defined;

  value_type  get(const key_type& id);
  void        set(const key_type& id, const value_type& value);
  void      reset(const key_type& id);

  // Get all the keys
  implementation-defined keys() const;

  // Home folder
  filesystem::path home() const;
  // Temporary folder as defined in the env
  filesystem::path temp() const;

  // Shell command, see ComSpec for windows
  filesystem::path shell() const;

  // The path variable, parsed.
  vector<filesystem::path> path() const;

  // The path extensions, that marka file as executable (empty on posix)
  vector<filesystem::path> extensions() const;

  // Find an executable file with this name.
  filesystem::path find_executable(const string& name);

  struct entry {
```

```
      using value_type = implementation-defined;

      entry();

      entry(const string&);
      entry(const wstring&);
      entry(const vector<value_type>&);

      entry& operator=(const string&);
      entry& operator=(const wstring&);
      entry& operator=(const vector<value_type>&);

      string string();
      wstring wstring();
      value_type native_string();

      // Split according to the OS specifics
      vector<value_type> as_vector();

      entry& operator=(const string&);
      entry& operator=(const wstring&);

      entry& operator=(const vector<value_type>&);
    };
  }
}
```

8.10 Header <pstream> Synopsis

```
#include <istream>
#include <ostream>
#include <streambuf>
#include <net> // Networking TS

namespace std {
  template<class CharT, class Traits = char_traits<CharT>>
  class basic_pipe_read_end;

  using pipe_read_end = basic_pipe_read_end<char>;
  using wpipe_read_end = basic_pipe_read_end<wchar_t>;

  template<class CharT, class Traits = char_traits<CharT>>
  class basic_pipe_write_end;

  using pipe_write_end = basic_pipe_write_end<char>;
  using wpipe_write_end = basic_pipe_write_end<wchar_t>;

  template<class CharT, class Traits = char_traits<CharT>>
  class basic_pipe;

  using pipe = basic_pipe<char>;
  using wpipe = basic_pipe<wchar_t>;

  template<class CharT, class Traits = char_traits<CharT>>
  class basic_pipebuf;

  using pipebuf = basic_pipebuf<char>;
  using wpipebuf = basic_pipebuf<wchar_t>;
```

```
template<class CharT, class Traits = char_traits<CharT>>
class basic_ipstream;

using ipstream = basic_ipstream<char>;
using wipstream = basic_ipstream<wchar_t>;

template<class CharT, class Traits = char_traits<CharT>>
class basic_opstream;

using opstream = basic_opstream<char>;
using wopstream = basic_opstream<wchar_t>;

template<class CharT, class Traits = char_traits<CharT>>
class basic_pstream;

using pstream = basic_pstream<char>;
using wpstream = basic_pstream<wchar_t>;

class async_pipe;
class async_pipe_read_end;
class async_pipe_write_end;

template<class CharT, class Traits>
struct tuple_size<basic_pipe<Char, Traits>> {
    constexpr static size_t size = 2;
};
template<class CharT, class Traits>
struct tuple_element<0, basic_pipe<Char, Traits>> {
    using type = basic_pipe_read_end<CharT, Traits>;
};
template<class CharT, class Traits>
struct tuple_element<1, basic_pipe<Char, Traits>> {
    using type = basic_pipe_write_end<CharT, Traits>;
};

template<size_t Index, class CharT, class Traits>
auto get(basic_pipe<Char, Traits>&&);
template<size_t Index, class CharT, class Traits>
auto get(const basic_pipe<Char, Traits>&);

template<class CharT, class Traits>
basic_pipe_read_end<CharT, Traits> get<0>(const basic_pipe<Char, Traits>&);
template<class CharT, class Traits>
basic_pipe_read_end<CharT, Traits> get<0>(basic_pipe<Char, Traits>&&);

template<class CharT, class Traits>
basic_pipe_write_end<CharT, Traits> get<1>(const basic_pipe<Char, Traits>&);
template<class CharT, class Traits>
basic_pipe_write_end<CharT, Traits> get<1>(basic_pipe<Char, Traits>&&);

template<class CharT, class Traits>
struct tuple_size<basic_pipe<Char, Traits>> {
    constexpr static size_t size = 2;
};

template<class CharT, class Traits>
struct tuple_size<basic_pstream<Char, Traits>> {
    constexpr static size_t size = 2;
};
template<class CharT, class Traits>
```

```
  struct tuple_element<0, basic_pipe<Char, Traits>> {
      using type = basic_ipstream<CharT, Traits>;
  };
  template<class CharT, class Traits>
  struct tuple_element<1, basic_pipe<Char, Traits>> {
      using type = basic_opstream<CharT, Traits>;
  };

  template<size_t Index, class CharT, class Traits>
  auto get(basic_pstream<Char, Traits>&&);
  template<size_t Index, class CharT, class Traits>
  auto get(const basic_pstream<Char, Traits>&);

  template<class CharT, class Traits>
  basic_ipstream<CharT, Traits> get<0>(const basic_pstream<Char, Traits>&);
  template<class CharT, class Traits>
  basic_ipstream<CharT, Traits> get<0>(basic_pstream<Char, Traits>&&);

  template<class CharT, class Traits>
  basic_opstream<CharT, Traits> get<1>(const basic_pstream<Char, Traits>&);
  template<class CharT, class Traits>
  basic_opstream<CharT, Traits> get<1>(basic_pstream<Char, Traits>&&);

  template<class CharT, class Traits>
  struct tuple_size<basic_pipe<Char, Traits>> {
      constexpr static size_t size = 2;
  };

  template<class CharT, class Traits>
  struct tuple_size<async_pipe> {
      constexpr static size_t size = 2;
  };
  template<class CharT, class Traits>
  struct tuple_element<0, async_pipe> {
      using type = async_pipe_read_end;
  };
  template<class CharT, class Traits>
  struct tuple_element<1, async_pipe> {
      using type = async_pipe_write_end;
  };

  template<size_t Index, class CharT, class Traits>
  auto get(const async_pipe&);
  template<size_t Index, class CharT, class Traits>
  auto get(async_pipe&&);

  template<class CharT, class Traits>
  async_pipe_read_end get<0>(const async_pipe&);
  template<class CharT, class Traits>
  async_pipe_read_end get<0>(async_pipe&&);

  template<class CharT, class Traits>
  async_pipe_write_end get<1>(const async_pipe&);
  template<class CharT, class Traits>
  async_pipe_write_end get<1>(async_pipe&&);
}
```

8.11 Classes `basic_pipe_read_end`, `basic_pipe_write_end`, `basic_pipe`

```
namespace std {
  template<class CharT, class Traits = char_traits<CharT>>
  class basic_pipe_read_end {
  public:
    using char_type = CharT;
    using traits_type = Traits;
    using int_type = typename Traits::int_type;
    using pos_type = typename Traits::pos_type;
    using off_type = typename Traits::off_type;
    using native_handle_type = implementation-defined;

    // Default construct the pipe_end. Will not be opened.
    basic_pipe_read_end();

    basic_pipe_read_end(native_handle_type handle);

    basic_pipe_read_end(const basic_pipe_read_end& p);
    basic_pipe_read_end(basic_pipe_read_end&& lhs);

    basic_pipe_read_end& operator=(const basic_pipe_read_end& p);
    basic_pipe_read_end& operator=(basic_pipe_read_end&& lhs);

    // Destructor closes the handles
    ~basic_pipe_read_end();

    native_handle_type native_handle() const;

    void assign(native_handle_type h);

    // Read data from the pipe.
    int_type read(char_type* data, int_type count);

    // Check if the pipe is open.
    bool is_open();
    // Close the pipe
    void close();
  };

  template<class CharT, class Traits = char_traits<CharT>>
  class basic_pipe_write_end {
  public:
    using char_type = CharT;
    using traits_type = Traits;
    using int_type = typename Traits::int_type;
    using pos_type = typename Traits::pos_type;
    using off_type = typename Traits::off_type;
    using native_handle_type = implementation-defined;

    // Default construct the pipe_end. Will not be opened.
    basic_pipe_write_end();

    basic_pipe_write_end(native_handle_type handle);

    basic_pipe_write_end(const basic_pipe_write_end& p);
    basic_pipe_write_end(basic_pipe_write_end&& lhs);

    basic_pipe_write_end& operator=(const basic_pipe_write_end& p);
    basic_pipe_write_end& operator=(basic_pipe_write_end&& lhs);

    // Destructor closes the handles.
```

```
  ~basic_pipe_write_end();

  native_handle_type native_handle() const;

  void assign(native_handle_type h);

  // Write data to the pipe.
  int_type write(const char_type* data, int_type count);

  // Check if the pipe is open.
  bool is_open();
  // Close the pipe
  void close();
};

template<class CharT, class Traits = char_traits<CharT>>
class basic_pipe {
public:
  using char_type = CharT;
  using traits_type = Traits;
  using int_type = typename Traits::int_type;
  using pos_type = typename Traits::pos_type;
  using off_type = typename Traits::off_type;
  using native_handle_type = implementation-defined;

  using read_end_type = basic_pipe_read_end<CharT, Traits>;
  using write_end_type = basic_pipe_write_end<CharT, Traits>;

  // Default construct the pipe. Will be opened.
  basic_pipe();

  basic_pipe(const read_end_type& read_end, const write_end_type& write_end);
  basic_pipe(read_end_type&& read_end, write_end_type&& write_end);

  // Construct a named pipe.
  explicit basic_pipe(const filesystem::path& name);

  basic_pipe(const basic_pipe& p);
  basic_pipe(basic_pipe&& lhs);

  basic_pipe& operator=(basic_pipe&& lhs);

  // Destructor closes the handles
  ~basic_pipe();


  write_end_type& write_end() &;
  write_end_type&& write_end() &&;
  const write_end_type& write_end() const &;

  read_end_type& read_end() &;
  read_end_type&& read_end() &&;
  const read_end_type& read_end() const &;

  // Write data to the pipe
  int_type write(const char_type* data, int_type count);
  // Read data from the pipe
  int_type read(char_type* data, int_type count);

  // Check if the pipe is open
```

```
    bool is_open();
    // Close the pipe
    void close();
  };
}
```

8.12 Class templates basic_pipebuf, basic_opstream, basic_ipstream and basic_pstream

```
namespace std {
  template<class CharT, class Traits = char_traits<CharT>>
  struct basic_pipebuf : basic_streambuf<CharT, Traits> {
    using pipe_read_end = basic_pipe<CharT, Traits>;

    using int_type = typename Traits::int_type;
    using pos_type = typename Traits::pos_type;
    using off_type = typename Traits::off_type;

    constexpr static int default_buffer_size = implementation-defined;

    ///Default constructor, will also construct the pipe.
    basic_pipebuf();
    basic_pipebuf(const basic_pipebuf&) = default;
    basic_pipebuf(basic_pipebuf&&) = default;

    basic_pipebuf(const basic_pipebuf&) = default;
    basic_pipebuf(basic_pipebuf&&) = default;

    basic_pipebuf& operator=(const basic_pipebuf&) = delete;
    basic_pipebuf& operator=(basic_pipebuf&&) = default;

    // Destructor writes the rest of the data
    ~basic_pipebuf();

    // Construct/assign from a pipe
    basic_pipebuf(const pipe_type& p);
    basic_pipebuf(pipe_type&& p);

    basic_pipebuf& operator=(pipe_type&& p);
    basic_pipebuf& operator=(const pipe_type& p);

    // Write characters to the associated output sequence from the put area
    int_type overflow(int_type ch = traits_type::eof()) override;

    // Synchronize the buffers with the associated character sequence
    int sync() override;

    // Reads characters from the associated input sequence to the get area
    int_type underflow() override;

    // Set the pipe of the streambuf
    void pipe(const pipe_type& p);
    void pipe(pipe_type&& p);

    // Get a reference to the pipe
    pipe_type&       pipe() &;
    const pipe_type& pipe() const &;
    pipe_type&&      pipe() &&;

    // Check if the pipe is open
```

```
  bool is_open() const;

  // Open a new pipe
  basic_pipebuf<CharT, Traits>* open();

  // Open a new named pipe
  basic_pipebuf<CharT, Traits>* open(const filesystem::path& name);

  // Flush the buffer and close the pipe
  basic_pipebuf<CharT, Traits>* close();
};

template<class CharT, class Traits = char_traits<CharT>>
class basic_ipstream : public basic_istream<CharT, Traits> {
public:
  using pipe_end_type = basic_pipe_read_end<CharT, Traits>;
  using opposite_pipe_end_type = basic_pipe_write_end<CharT, Traits>;

  using char_type = CharT;
  using traits_type = Traits;

  using int_type = typename Traits::int_type;
  using pos_type = typename Traits::pos_type;
  using off_type = typename Traits::off_type;

  // Get access to the underlying streambuf
  basic_pipebuf<CharT, Traits>* rdbuf() const;

  basic_ipstream();

  basic_ipstream(const basic_ipstream&) = delete;
  basic_ipstream(basic_ipstream&& lhs);

  basic_ipstream& operator=(const basic_ipstream&) = delete;
  basic_ipstream& operator=(basic_ipstream&& lhs);

  // Construct/assign from a pipe
  basic_ipstream(const pipe_type& p);
  basic_ipstream(pipe_type&& p);

  basic_ipstream& operator=(const pipe_type& p);
  basic_ipstream& operator=(pipe_type&& p);

  // Set the pipe of the streambuf
  void pipe_end(const pipe_end_type& p);
  void pipe_end(pipe_end_type&& p);

  // Get a reference to the pipe
  pipe_end_type& pipe_end() &;
  const pipe_end_type& pipe_end() const&;
  pipe_end_type&& pipe_end() &&;

  // Check if the pipe is open
  bool is_open() const;

  // Open a new pipe
  opposite_pipe_end_type ovpen();

  // Open a new named pipe
  opposite_pipe_end_type open(const filesystem::path& name);
```

```
  // Flush the buffer and close the pipe
  void close();
};

template<class CharT, class Traits = char_traits<CharT>>
class basic_opstream : public basic_ostream<CharT, Traits> {
public:
  using pipe_end_type basic_pipe_write_end<CharT, Traits>;
  using opposite_pipe_end_type = basic_pipe_read_end<CharT, Traits>;

  using int_type = typename Traits::int_type;
  using pos_type = typename Traits::pos_type;
  using off_type = typename Traits::off_type;

  // Get access to the underlying streambuf
  basic_pipebuf<CharT, Traits>* rdbuf() const;

  basic_opstream();

  basic_opstream(const basic_opstream&) = delete;
  basic_opstream(basic_opstream&& lhs);

  basic_opstream& operator=(const basic_opstream&) = delete;
  basic_opstream& operator=(basic_opstream&& lhs);

  // Construct/assign from a pipe
  basic_opstream(const pipe_end_type& p);
  basic_opstream(pipe_end_type&& p);

  basic_opstream& operator=(const pipe_end_type& p);
  basic_opstream& operator=(pipe_end_type&& p);

  // Set the pipe_end
  void pipe_end(pipe_end_type&& p);
  void pipe_end(const pipe_end_type& p);

  // Get the pipe_end
  pipe_end_type&       pipe_end() &;
  const pipe_end_type& pipe_end() const&;
  pipe_end_type&&      pipe_end() &&;

  // Open a new pipe
  opposite_pipe_end_type open();
  // Open a new named pipe
  opposite_pipe_end_type open(const filesystem::path& name);

  // Flush the buffer & close the pipe
  void close();
};

template<class CharT, class Traits = char_traits<CharT>>
class basic_pstream : public basic_iostream<CharT, Traits> {
  mutable basic_pipebuf<CharT, Traits> _buf; // exposition-only
public:
  using pipe_type = basic_pipe<CharT, Traits>;

  using char_type   = CharT;
  using traits_type = Traits;
```

```
    using int_type = typename Traits::int_type;
    using pos_type = typename Traits::pos_type;
    using off_type = typename Traits::off_type;

    // Get access to the underlying streambuf
    basic_pipebuf<CharT, Traits>* rdbuf() const;

    basic_pstream();

    basic_pstream(const basic_pstream&) = delete;
    basic_pstream(basic_pstream&& lhs);

    basic_pstream& operator=(const basic_pstream&) = delete;
    basic_pstream& operator=(basic_pstream&& lhs);

    // Construct/assign from a pipe
    basic_pstream(const pipe_type& p);
    basic_pstream(pipe_type&& p);

    basic_pstream& operator=(const pipe_type& p);
    basic_pstream& operator=(pipe_type&& p);

    // Set the pipe of the streambuf
    void pipe(const pipe_type& p);
    void pipe(pipe_type&& p);

    // Get a reference to the pipe.
    pipe_type&       pipe() &;
    const pipe_type& pipe() const &;
    pipe_type&&      pipe() &&;

    // Open a new pipe
    void open();

    // Open a new named pipe
    void open(const filesystem::path& name);

    // Flush the buffer & close the pipe
    void close();
  };
}
```

The structure of the streams reflects the pipe_{end} distinction of `basic_pipe`. Additionally, the open function on the `ipstream` / `opstream` allows to open a full pipe and be handled by another class, e.g.:

```
std::ipstream is; // Not opened
std::opstream os{is.open()}; // Now is & os point to the same pipe
```

Or using aggregate initialization:

```
auto [is, os] = std::pstream();
```

Or to be used in a process

```
std::ipstream is; // Not opened
std::process proc("foo", std::process_io({}, is.open(), {})); // stdout can be read from is
```

8.13 Classes `async_pipe_read_end`, `async_pipe_write_end`, `async_pipe`

```
// The following is dependent on the networking TS
namespace std {
  class async_pipe_read_end {
```

```
public:
using native_handle_type = implementation-defined;

  async_pipe_read_end(net::Executor& ios);
  async_pipe_read_end(net::Executor& ios, native_handle_type native_handle);

  async_pipe_read_end(const async_pipe_read_end& lhs);
  async_pipe_read_end(async_pipe_read_end&& lhs);

  async_pipe_read_end& operator=(const async_pipe_read_end& lhs);
  async_pipe_read_end& operator=(async_pipe_read_end&& lhs);

  // Construct form pipe_end
  template<class CharT, class Traits = char_traits<CharT>>
  explicit async_pipe_read_end(net::Executor& ios, const basic_pipe_read_end<CharT, Traits>& p);

  // NOTE: Windows requires a named pipe for this, if a the wrong type is used an exception is thrown
  template<class CharT, class Traits = char_traits<CharT>>
  inline async_pipe_read_end& operator=(const basic_pipe_read_end<CharT, Traits>& p);

  // Destructor closes the pipe handles
  ~async_pipe_read_end();

  // Explicit conversion operator to basic_pipe/
  template<class CharT, class Traits = char_traits<CharT>>
  explicit operator basic_pipe_read_end<CharT, Traits>() const;

  template<typename CharT = char, typename Traits = char_traits<CharT>>
  basic_pipe_write_end<CharT, Traits> open();
  template<typename CharT = char, typename Traits = char_traits<CharT>>
  basic_pipe_write_end<CharT, Traits> open(const filesystem::path& path);

  // Cancel the current asynchronous operations
  void cancel();

  void close();

  // Check if the pipe end is open
  bool is_open() const;

  // Read some data from the handle.
  // See the Networking TS for more details.
  template<class MutableBufferSequence>
  size_t read_some(const MutableBufferSequence& buffers);

  native_handle_type native_handle() const;

  // Start an asynchronous read
  template<typename MutableBufferSequence,
           typename ReadHandler>
  implementation-defined async_read_some(
      const MutableBufferSequence& buffers,
            ReadHandler&& handler);
};

class async_pipe_write_end {
public:
  using native_handle_type = implementation-defined;

  async_pipe_write_end(net::Executor& ios);
```

```
  async_pipe_write_end(net::Executor& ios, native_handle_type native_handle);

  async_pipe_write_end(const async_pipe_write_end& lhs);
  async_pipe_write_end(async_pipe_write_end&& lhs);

  async_pipe_write_end& operator=(const async_pipe_write_end& lhs);
  async_pipe_write_end& operator=(async_pipe_write_end&& lhs);

  // Construct from pipe_end
  template<class CharT, class Traits = char_traits<CharT>>
  explicit async_pipe_write_end(net::Executor& ios, const basic_pipe_write_end<CharT, Traits>& p);

  // NOTE: Windows requires a named pipe for this, if a the wrong type is used an exception is thrown
  template<class CharT, class Traits = char_traits<CharT>>
  async_pipe_write_end& operator=(const basic_pipe_write_end<CharT, Traits>& p);

  // Destructor closes the pipe handles
  ~async_pipe_write_end();

  // Explicit conversion operator to basic_pipe
  template<class CharT, class Traits = char_traits<CharT>>
  explicit operator basic_pipe_write_end<CharT, Traits>() const;

  // Open the pipe
  template<typename CharT = char, typename Traits = char_traits<CharT>>
  basic_pipe_read_end<CharT, Traits> open();
  template<typename CharT = char, typename Traits = char_traits<CharT>>
  basic_pipe_read_end<CharT, Traits> open(const filesystem::path& path);

  // Cancel the current asynchronous operations
  void cancel();

  void close();

  // Check if the pipe end is open
  bool is_open() const;

  // Write some data to the handle
  template<typename ConstBufferSequence>
  size_t write_some(const ConstBufferSequence& buffers);

  // Get the native handle of the source
  native_handle_type native_handle() const;

  // Start an asynchronous write
  template<typename ConstBufferSequence,
           typename WriteHandler>
  implementation-defined async_write_some(
      const ConstBufferSequence& buffers,
      WriteHandler&& handler);
};

// Class for async I/O with the Networking TS
// Can be used directly with net::async_read/write
class async_pipe {
public:
  using native_handle_type = implementation-defined;

  // Construct a new async_pipe
  // Automatically opens the pipe
```

```
// Initializes source and sink with the same net::Executor
// NOTE: Windows creates a named pipe here, where the name is automatically generated.
async_pipe(net::Executor& ios);

// NOTE: Windows restricts possible names
async_pipe(net::Executor& ios, const filesystem::path& name);

// NOTE: Windows requires a named pipe for this, if a the wrong type is used an exception is thrown
async_pipe(const async_pipe& lhs);
async_pipe(async_pipe&& lhs);

async_pipe& operator=(const async_pipe& lhs);
async_pipe& operator=(async_pipe&& lhs);

// Construct from a pipe
// @note Windows requires a named pipe for this, if a the wrong type is used an exception is thrown
template<class CharT, class Traits = char_traits<CharT>>
explicit async_pipe(net::Executor& ios, const basic_pipe<CharT, Traits>& p);

// NOTE: Windows requires a named pipe for this, if a the wrong type is used an exception is thrown
template<class CharT, class Traits = char_traits<CharT>>
async_pipe& operator=(const basic_pipe<CharT, Traits>& p);

// Returns a copied pipe read end
const async_pipe_read_end&  read_end() const &;
      async_pipe_read_end&& read_end() &&;

// Returns a copied pipe write end
const async_pipe_write_end&  write_end() const &;
      async_pipe_write_end&& write_end() &&;

// Destructor, closes the pipe handles
~async_pipe();

// Explicit conversion operator to basic_pipe
template<class CharT, class Traits = char_traits<CharT>>
explicit operator basic_pipe<CharT, Traits>() const;

// Cancel the current asynchronous operations
void cancel();

// Close the pipe handles
void close();

// Check if the pipes are open
bool is_open() const;

// Read some data from the handle.
// See the Networking TS for more details.
template<class MutableBufferSequence>
size_t read_some(const MutableBufferSequence& buffers);

// Write some data to the handle.
// See the Networking TS for more details.
template<class ConstBufferSequence>
size_t write_some(const ConstBufferSequence& buffers);

native_handle native_source() const;
native_handle native_sink() const;
```

```
    // Start an asynchronous read
    template<class MutableBufferSequence,
             class ReadHandler>
    implementation-defined async_read_some(
        const MutableBufferSequence& buffers,
        ReadHandler&& handler);

    // Start an asynchronous write
    template<class ConstBufferSequence,
             class WriteHandler>
    implementation-defined async_write_some(
        const ConstBufferSequence& buffers,
        WriteHandler&& handler);
  };
};
}
```

`async_pipe` is structured similar to the `basic_pipe` triple. The `async_pipe_end*::open` returns a 'basic$_{\text{pipeend}*}$' to the other side. This allows to use it in a process or to construct an opposite async$_{\text{pipe}}$:

```
std::net::system_executor exec;
std::async_pipe_read_end ip{exec}; // Not opened
std::async_pipe_read_end op{exec, ip.open()}; // Now re & os point to the same pipe, though can use dif
```

Or using aggregate initialization:

```
std::net::system_executor exec;
auto [ip, op] = std::async_pipe(exec);
```

Or to be used in a process

```
std::net::system_executor exec;
std::async_pipe_read_end ip{exec};
std::process proc("foo", std::process_io({}, ip.open(), {}));
```

9 Open Questions

9.1 Core language impact

The group is aware that there maybe core changes required for this proposal to correctly the define the needed terminology and behavior. None of us are expert in this and would appreciate help in this area. Some questions the group asked include:

- Can we piggyback on the thread's forward progress stuff for process as well?
- Can we assume all threads on the system behave like C++ threads?

10 Acknowledgements

This proposal reflects the effort of the c++ community at C++Now 2019 and afterward. The primary participants are listed as authors on the paper, but many others participated in discussion of details during morning workshop sessions and conference breaks.

None of this would have been possible without the work and guidance of Klemens Morgenstern, author of boost.process.

11 References

- Github repository for this proposal `https://github.com/JeffGarland/liaw2019-process`
- Additional user examples not included in the proposal `https://github.com/JeffGarland/liaw2019-process/tree/master/example`
- Isabella Muerte Desert Sessions: Improving hostile environment interactions `http://wg21.link/p1275`
- boost.process documentation `https://www.boost.org/libs/process`
- Standard Library wishlist (Matt Austern) `https://docs.google.com/document/d/1AC3vkOgFezPaeSZO-fvxgwzEIabwseE7yFG_16Bk/preview`
- cppcast with Klemens on boost.process `https://channel9.msdn.com/Shows/CppCast/Episode-72-BoostProcess-wi`
- Pacific c++ Klemens on boost.process design `https://www.youtube.com/watch?v=uZ4IG1OfeR0`