A Proposal to Add Process Management to the C++ Standard Library

Document #:	D1750R2
Date:	2019-11-08
Project:	Programming Language C++ LEWGI SG1 SG16

Reply-to:	Klemens Morgenstern Jeff Garland <jeff@crystalclearsoftware.com> Elias Kosunen <isocpp@eliaskosunen.com> Fatih Bakir

Contents

1 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 feature wishlists ([wishlist]) going back to 2007. This proposal is based on Boost.Process: [Boost.Process documentation]. which provides cross-platform implementation experience.

2 Revision History

2.1 R2

TODO

2.2 R1

— Added a section of review questions and dicussion to record committee discussions - see Committee Questions and Discussions
— LEWGI feedback: removed `environment` class
— LEWGI feedback: can `native_exit_code` and `exit_code` be combined?
— LEWGI feedback: removed `process::operator bool`
— LEWGI feedback: removed `process_group` class
— LEWGI feedback: removed `native_handles` from pipes
— LEWGI feedback: are pids fundamental?
— SG1 feedback: added a section on other C++ libraries with process management
— SG1 feedback: added section discussing the `process` destructor behavior
— SG1 feedback: added section discussing issues around forward progress
— Make `pipe` a concrete, byte-based type, instead of a template over a character type. Removed `basic_pipe`.
— SG16 feedback: Remove `char_traits` and type aliases based on it from `pipe`
— Make `pipe::read` and `pipe::write` take a `span` instead of a pointer and a size
— `snake_case` concepts, `process_launcher` -> `default_process_launcher` to avoid collisions
— Add an exposition only `process-argument-list` concept
— Fix Range-based templates, adding `requires`-clauses
— Fix `tuple_element` and `get` specializations for `pipe` and `pstream`
— Numerous editorial fixes

2.3 R0

Initial release

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
— 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.

```cpp
#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("/usr/bin/g++", 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: [Examples].

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 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.

4.3 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 [P1275R0] referencing a process environment. However, P1275 will need to be enhanced to support multiple instances of environments for access and modification of child process environment.

At this time the examples show in this proposal will require an enhanced P1275 to be functional.

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 Other C++ Libraries

5.2.1 `ACE::Process`

The Adaptive Communication Environment [ACE::Process] library is an open source library that implements many wrappers around operating system primatives as part of concurrency and communications environment. The library has been ported to a myriad of platforms/operating systems. It has been used in commercial applications since the late 1990's and is the core for TAO Common Object Request Broker (CORBA) implementation and the Data Distribution Service (DDS) openDDS implementation.

The primary type provided by the library for process management is `ACE_Process`. This class provides the mechanisms to create and manage a child process. The `ACE_Process_Options` class facilities the command line and environment setup. In addition, the `ACE_Process_Manager` for managing a group of processes.

5.2.2 Qt `QProcess`

Qt provides the core class `QProcess` [QProcess] with the facilities for process spawning and management.

5.2.3 GNOME `glib::spawn`

The GNOME open source libraries [glib::spawn] (linux only) provide a set of functions to spawn and manage child processes in C++. Beyond basic functions, the functions provide both synchronous and asynchronous execution as well as pipe integration.

5.2.4 cpp-subprocess

The cpp-subprocess ([cpp-subprocess]) library uses C++11 to provide a Python-like interface to process management for a limited set of Unix-like platforms. The library supports pipe integration and environment setup.

5.2.5 Redirected Process (reproc)

The reproc ([reproc]) library provides a cross-platform (Windows and POSIX) process management facility including stream integration. The library supports stream and environment setup facilities.

5.3 Java

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

```java
// 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())
```

5.4 Python

```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)
```

5.5 Rust

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

```rust
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());
```

5.6 Node.js

```
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 Committee Questions and Discussion

6.1 Investigate combining `exit_code` and `native_exit_code`

This question was raised in LEWGI in Cologne. The two types are not obviously combinable and serve different purposes. The reason for `exit_code` is so one can write portable cross-platform code. The reason for `native_exit_code` is so one can write platform specific code.

6.2 Are process ids (pids) fundamental for version 1?

The view of the authors is, that pids are fundamental in the same way that `std::thread::id` is fundamental. Aside from being useful for applications in logging, they are also needed for interacting with the native APIs using the `native_handle`.

6.3 Investigate a non-exception api for error handling

This question was raised in LEWGI in Cologne. It's clear, that the library can provide an API that uses error codes instead of exceptions. This would look something like the following:

```
namspace std {

struct process_make_ret {
  process    a_process;
  error_code error;
};

class process {
  friend make_process_ret make_process(...);
public:
  process(...);
};

process_make_ret make_process(...);
```

The unfortunate result is an API inconsistency with `std::jthread` and `std::thread`, which are otherwise similar in usage to `process`.

Alternatively, users can write their own wrapper using the current proposal, since `process` has a default constructor and a `valid` member function.

```
// User code

struct process_make_ret {
  std::process    process;
  std::error_code error;
};

process_make_ret make_process(...) {
  process_make_ret ret;
  try {
    ret.process = std::process(...);
  }
  catch (const std::system_error& err) {
    ret.error = err.code();
  }
}
```

6.4 `process` destructor should not terminate

SG1 in Cologne discussed the behavior of the `process` destructor at length. Originally, it was proposed, that the program would terminate if `wait` had not been called, like `std::thread` does. The over arching backdrop of SG1 discussion was, that `std::thread` destructor calling terminate was a poor design choice, that was not to be repeated (see `std::jthread`).

The authors would like committee guidance and discussion of some possible options, including:

— call `process::terminate` on child
— call `wait` by default in the destructor
— add some sort of `request_stop` interface to process to mirror `std::jthread` api
— a constructor option to pick from pre-defined behaviors like `wait` or `detach`

Note that calling `wait` in the destructor can also cause poor behavior if the child process never exits. This, however, is similar to the problem with `jthread::join`, if the user fails to implement cooperative shutdown logic.

6.5 Can `std::process` and `std::thread` interchangeably be used in generic code?

This was discussed in some length in SG1 in Cologne, with the general conclusion, that this proposal should not provide this feature. While there was weak support for the idea, the domains are different enough, that it was deemed unnecessary. Advice was to not pursue this issue further.

6.6 Forward progress and core language impact

This was discussed at length by SG1 in Cologne. The question that started the discussion was:

— Can we piggyback on `std::thread`'s forward progress stuff for `process` as well?
— Can we assume all threads on the system behave like C++ threads?
— What can we say about the executing process?

Some key points included:

— It is impossible, in the scope of the standard, to describe the external process which is not necessarily C++
— We cannot assume forward progress, since it's not really possible for us to describe
— We should avoid trying to describe forward progress for `process`

So, at this time, the proposal will say nothing.

7 Design Discussion and Examples

7.1 Concept `process_launcher`

The process launcher is a class that implements 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. Also, while POSIX can change a user by utilizing `setuid` in a `process_initializer`, 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/program", {}, 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::default_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 in well 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.

7.2 Concept `process_initializer`

The process initializer is a class that modifies the behavior of a process. There is no guarantee that a custom initializer is portable, i.e. it will not only be dependent 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<process_launcher 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});
```

7.3 Class `process`

7.3.1 Constructor

```
process(const std::filesystem::path&, const process-argument-list&, Inits&&... init)
```

This is the default launching function, and forwards to `std::default_process_launcher`. Boost.Process supports a cmd-style execution (similar to `std::system`), which we opted to remove from this proposal. 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 executable 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.

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

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

// 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 `process_launcher` as a `shell_launcher`.

7.3.2 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();
```

7.3.3 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();
}
```

7.3.4 Function `native_handle`

Since there is a lot functionality 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, they could still do so:

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

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

7.3.5 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 they 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 to 1, since no input files

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

7.3.6 Function `async_wait`

To allow asynchronous operations, the process library integrates 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 bool exit_code = fut.get();
assert(exit_code == proc.exit_code());
```

7.4 Class `process_io`

`process_io` takes three standard handles, because of requirements on some operating systems. Either all three are set or all are defaults.

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

7.4.1 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::pipe fwd = std::pipe();

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

You can also use any `pstream` type instead.

7.4.2 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, {}, {});
```

7.4.3 `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 the type is changed

#“cpp // Redirect stderr to stdout std::process proc (“./foo”, std::process_io({}, {}, std::this_process::io().stdout());

```
### 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`.

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

7.4.4 Other objects

Other objects, that use an underlying stream handle, could also 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"));
```

7.4.5 Null device (not yet specified)

8

The null-device is a a feature of both POSIX (“/dev/null”) and Windows (“NUL”). It accepts writes, and always returns. It might be worth it to consider adding it.

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

// Not (yet) part of this paper
std::process proc("./foo", {}, std::process_io(
  std::process_io::null(), std::process_io::null(), std::process_io::null()));
```

9 Design Decisions

9.1 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.

9.2 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(default_process_launcher.launch(...)) {}
default_process_launcher().launch(...) -> process;
```
```

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

9.3 wait or join

The name of the method in `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 bikeshedding.

9.4 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.

9.5 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 behavior 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 behavior, 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.

9.6 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 responsibility 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`.

9.7 Extensibility

To be extensible this library uses two concepts: `process_launcher` and `process_initializer`.

A `process_launcher` is the actual function creating the process. It can be used to provide platform dependent behavior 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 `process_initializer` 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.

The `process_launcher` has three methods that must be provided by a custom launcher. These are:

— `on_setup`: calls the initializer before attempting to launch
— `on_success`: calls the initializer after successful process launch
— `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.

9.8 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.

9.9 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.

9.10 Integration of `iostream` 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.

10 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.

10.1 Header `<process>` synopsis

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

namespace std {
  // Command line argument list
  // Could also be specified as a set of named requirements
  // An input_range of basic_strings,
  // where the character type is either char, wchar_t, char8_t
  template<class charT>
  inline constexpr bool process-argument-list-char = // exposition only
    is_same_v<charT, char> ||
    is_same_v<charT, wchar_t> ||
    is_same_v<charT, char8_t>;
  template<ranges::input_range R, class charT, class traits>
  concept process-argument-list = // exposition only
```

```
  requires convertible_to<
    ranges::iter_value_t<ranges::iterator_t<R>>, basic_string<charT, traits>> &&
  requires process-argument-list_char<charT>;

// A process-laucher is an object that has a launch function taking
// a path, arguments and a list of initializers, and returns a process.
// Could also be specified as a set of named requirements
template<class T, process-argument-list Args>
concept process-launcher = // exposition only
  requires(T launcher, const Args& args,
           ranges::iter_value_t<ranges::iterator_t<Args>> errmsg,
           error_code& err, const filesystem::path& path) {
    launcher.set_error(err, errmsg);
    { launcher.launch(path, args) } -> process;
  };

// The default process-launcher of the implementation
class default_process_launcher;

// A process-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.
template<class Init,
         process-launcher Launcher = default_process_launcher>
concept process-initializer = // exposition only
  requires(Init i, Launcher l, error_code& err) {
    i.on_setup(l);
  } ||
  requires(Init i, Launcher l) {
    i.on_success(l);
  } ||
  requires(Init i, Launcher l) {
    i.on_error(l, err);
  };

// Identifier for a process
// Is trivially copyable, and satisfies strict_totally_ordered
using pid_type = implementation-defined;

// Provides a portable, unique handle to an operating system process
class process;

// Exception type thrown on error
// Can also have a:
//  - fs::path, if failing before launch
//  - pid_type, if failing after launch
class process_error;

// Provides initializers for the standard I/O
class process_io;

// Satisfies process-initializer
// From P1275
// class environment;
```

```
  // Satisfies process-initializer
  class process_limit_handles;
}
```

10.2 Class process

```
namespace std {
  class process {
  public:
    using native_handle_type = implementation-defined;

    // 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;

    // Construct a child from a property list and launch it.
    template<process-argument-list R, process-initializer... Inits>
    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<process-argument-list R, process_initializer... Inits,
             process-launcher Launcher>
    explicit process(const filesystem::path& exe,
                     const R& args,
                     Inits&&... inits,
                     Launcher&& launcher);

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

    // Not copyable

    process(process&&) = default;
    process& operator=(process&&) = default;

    // tbd behavior
    ~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 internally.
```

```
    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 or did exist.
    bool valid() const;

    // 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_code), i.e. wait for the process to exit and get the exit_code if exited.
    template<class Executor, class CompletionToken>
    auto async_wait(Executor& ctx, CompletionToken&& token);
  };
}
```

10.3 Class process_error

```
namespace std {
  class process_error : public system_error {
    // filesystem_error can take up to two paths in case of an error
    // In the same vein, process_error can take a path or a pid
    process_error(const string& what_arg, error_code ec);
    process_error(const string& what_arg,
                  const filesystem::path& path,
                  std::error_code ec);
    process_error(const string& what_arg,
                  pid_type pid_arg,
                  std::error_code ec);

    const filesystem::path& path() const noexcept;
    pid_type pid() const noexcept;

    const char* what() const noexcept override;
  };
```

```
}
```

10.4 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 process-initializer
  class process_io {
  public:
    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
  };
}
```

10.5 Class `process_limit_handles`

This `limit_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 process-initializer
  class process_limit_handles {
  public:
    // Select all the handles that should be inherited even though they are not
    // used by any initializer.
    template<class... Handles>
    process_limit_handles(Handles&&... handles);
  };
}
```

10.6 Namespace `this_process`

This namespace provides information about the current process.

```
namespace std::this_process {
  using pid_type = implementation-defined;
  using native_handle_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 Windows) of the current process.
// 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, meaning it might be quite slow.
// NOTE: This functionality is utterly prone to race conditions, since other
// threads might open or close handles.
vector<native_handle_type> get_handles();
template<ranges::output_iterator It>
It get_handles(It it);

// Determines if a given handle is a stream-handle, i.e. any handle that can
// be used with read and write functions.
// 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;
  // Note that Windows uses wchar_t for key_type, the key type should be able to be
  // constructed from a char* though. 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
  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.
    template<ranges::output_iterator It>
    It path(It it) const;

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

    template<ranges::output_iterator It>
    It extensions(It it) const;

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

    struct entry {
      using value_type = implementation-defined;

      entry();

      entry(string_view);
      entry(const string&);
      entry(const wstring&);
      entry(const vector<value_type>&);
      template<ranges::input_range Rng>
        requires convertible_to<ranges::iter_value_t<ranges::iterator_t<Rng>>, value_type>
      entry(const Rng& r);

      entry& operator=(string_view);
      entry& operator=(const string&);
      entry& operator=(const wstring&);
      entry& operator=(const vector<value_type>&);
      template<ranges::input_range Rng>
        requires convertible_to<ranges::iter_value_t<ranges::iterator_t<Rng>>, value_type>
      entry& operator=(const Rng& r);

      string as_string();
      wstring as_wstring();
      value_type as_native_string();

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

      template<ranges::output_iterator It>
      It as_range(It it) const;
    };
  }
}
```

10.7 Header <pstream> Synopsis

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

```
namespace std {
  class pipe_read_end;
  class pipe_write_end;
  class pipe;

  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>;

  struct tuple_size<pipe> {
  class async_pipe;
  class async_pipe_read_end;
  class async_pipe_write_end;

  struct tuple_size<pipe> {
    constexpr static size_t size = 2;
  };
  struct tuple_element<0, pipe> {
    using type = pipe_read_end;
  };
  struct tuple_element<1, pipe> {
    using type = pipe_write_end;
  };

  template<size_t Index>
  auto get(pipe&&);
  template<size_t Index>
  auto get(const pipe&);

  pipe_read_end get<0>(const pipe&);
  pipe_read_end get<0>(pipe&&);

  pipe_write_end<CharT, Traits> get<1>(const pipe&);
```

```
pipe_write_end<CharT, Traits> get<1>(pipe&&);

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_pstream<Char, Traits>> {
  using type = basic_ipstream<CharT, Traits>;
};
template<class CharT, class Traits>
struct tuple_element<1, basic_pstream<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>&&);

struct tuple_size<pipe> {
  constexpr static size_t size = 2;
};

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

template<size_t Index>
auto get(const async_pipe&);
template<size_t Index>
auto get(async_pipe&&);

async_pipe_read_end get<0>(const async_pipe&);
async_pipe_read_end get<0>(async_pipe&&);

async_pipe_write_end get<1>(const async_pipe&);
```

```
  async_pipe_write_end get<1>(async_pipe&&);
}
```

10.8 Classes `pipe_read_end`, `pipe_write_end`, `pipe`

```
namespace std {
  class pipe_read_end {
  public:
    // Default construct the pipe_end. Will not be opened.
    pipe_read_end();

    pipe_read_end(const pipe_read_end& p);
    pipe_read_end(pipe_read_end&& lhs);

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

    // Destructor closes the handles
    ~pipe_read_end();

    // Read data from the pipe.
    size_t read(span<byte> data);

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

  class pipe_write_end {
  public:
    // Default construct the pipe_end. Will not be opened.
    pipe_write_end();

    pipe_write_end(const pipe_write_end& p);
    pipe_write_end(pipe_write_end&& lhs);

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

    // Destructor closes the handles.
    ~pipe_write_end();

    // Write data to the pipe.
    size_t write(span<const byte> data);

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

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

```
  class pipe {
  public:
    // Default construct the pipe. Will not be opened.
    pipe();

    pipe(const pipe_read_end& read_end, const pipe_write_end& write_end);
    pipe(pipe_read_end&& read_end, pipe_write_end&& write_end);

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

    pipe(pipe&& lhs);
    pipe& operator=(pipe&& lhs);

    // Destructor closes the handles
    ~pipe();

    pipe_write_end& write_end() &;
    pipe_write_end&& write_end() &&;
    const pipe_write_end& write_end() const &;

    pipe_read_end& read_end() &;
    pipe_read_end&& read_end() &&;
    const pipe_read_end& read_end() const &;

    // Write data to the pipe
    size_t write(span<const byte> data);
    // Read data from the pipe
    size_t read(span<byte> data);

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

10.9 Class templates `basic_pipebuf`, `basic_opstream`, `basic_ipstream`, `basic_pstream`

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

    constexpr static int default_buffer_size = implementation-defined;

    // Default constructor, will not 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& p);
  basic_pipebuf(pipe& p);

  basic_pipebuf& operator=(pipe&& p);
  basic_pipebuf& operator=(const pipe& 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 = pipe_read_end;
```

```
  using opposite_pipe_end_type = pipe_write_end;

  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 open();

  // 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 = pipe_write_end;
  using opposite_pipe_end_type = pipe_read_end;
```

```
  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 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& p);
    basic_pstream(pipe&& p);

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

    // Set the pipe of the streambuf
    void pipe(const pipe& p);
    void pipe(pipe&& 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 $_{\text{pipe\_end}}$ distinction of $_{\text{pipe}}$. Additionally, the open function on the $_{\text{ipstream}}$ / $_{\text{opstream}}$ 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
```

10.10 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:
  template<class Executor>
  async_pipe_read_end(Executor& ios);

  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 Executor>
  explicit async_pipe_read_end(Executor& ios,
                               const pipe_read_end& p);

  // NOTE: Windows requires a named pipe for this, if a the wrong type is used an
  // exception is thrown.
  inline async_pipe_read_end& operator=(const pipe_read_end& p);

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

  // Explicit conversion operator to pipe_read_end
  explicit operator pipe_read_end() const;

  pipe_write_end<CharT, Traits> open();
  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);

  // Note: MutableBufferSequence is span<span<byte>>
  // Start an asynchronous read
  template<class MutableBufferSequence,
           class ReadHandler>
  implementation-defined async_read_some(
      const MutableBufferSequence& buffers,
            ReadHandler&& handler);
};

class async_pipe_write_end {
public:
  template<class Executor>
```

```
  async_pipe_write_end(Executor& ios);

  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 Executor>
  explicit async_pipe_write_end(Executor& ios,
                                const pipe_write_end& p);

  // NOTE: Windows requires a named pipe for this, if a the wrong type is used an
  // exception is thrown.
  async_pipe_write_end& operator=(const pipe_write_end<& p);

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

  // Explicit conversion operator to pipe_write_end
  explicit operator pipe_write_end() const;

  // Open the pipe
  pipe_read_end open();
  pipe_read_end 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<class ConstBufferSequence>
  size_t write_some(const ConstBufferSequence& buffers);

  // Start an asynchronous write
  template<class ConstBufferSequence,
           class 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:
  // 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.
template<class Executor>
async_pipe(Executor& ios);

// NOTE: Windows restricts possible names
template<class Executor>
async_pipe(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 Executor>
explicit async_pipe(Executor& ios, const pipe& p);

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

    // 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 `pipe` triple. The `async_pipe_end*::open` returns a `pipe_end_*` to the other side. This allows to use it in a process or to construct an opposite async_pipe:

```
std::net::system_executor exec;
std::async_pipe_read_end ip{exec}; // Not opened
// After next line ip & op point to the same pipe, though can use different executors.
std::async_pipe_read_end op{exec, ip.open()};
```

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(), {}));
```

11 Open question

11.1 Splitting pipes out

In Cologne, both LEWGI and SG1 discussed, if pipes should be part of a different standalone proposal. This does not appear to be such an easy task from the authors view, since pipes are quite fundamental to the proposal. Also, pipes got changed as part of the SG16 review. As a result, they remain in this paper for at least one more round of discussion.

11.2 Elimination of `async_pipe`

There was a suggestion of a technique to eliminate the `async_pipe` while maintaining the functions through streams. The authors need to followup on this suggested design change.

12 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.

13 References

[ACE::Process] ACE::Process library documentation.
http://www.dre.vanderbilt.edu/Doxygen/Stable/libace-doc/a06768.html

[Boost.Process documentation] Klemens Morgenstern. Boost.Process documentation.
https://www.boost.org/libs/process

[cpp-subprocess] cpp-subprocess repository.
https://example.com

[Examples] Additional user examples not included in the proposal.
https://github.com/JeffGarland/liaw2019-process/tree/master/example

[glib::spawn] GNOME glib::spawn documentation.
https://example.com

[P1275R0] Isabella Muerte. 2018. Desert Sessions: Improving hostile environment interactions.
https://wg21.link/p1275r0

[QProcess] QProcess documentation.
https://example.com

[reproc] reproc repository.
https://example.com

[wishlist] Matt Austern. Standard library wishlist.
https://docs.google.com/document/d/1AC3vkOgFezPaeSZO-fvxgwzEIabw8I_seE7yFG_16Bk/preview