Portable yet thin OS abstractions

Klemens Morgenstern

18.10.2018

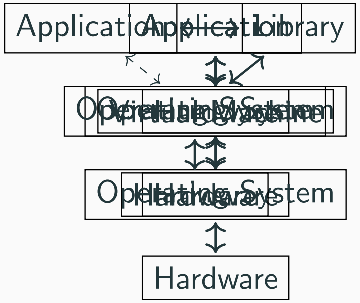
Introduction

- Electrical engineer by training
- C++ developer by passion
- Author of boost.process
- Independent contractor
- Open for consulting

Goal of C++ system libraries

- Portable C++ Applications
- Low/Zero overhead
- Thin rather than thick
- Libraries and not frameworks

Portable C++ Application



- Application code compiles against different OS
- OS specific extensions are possible

Low/Zero overhead

‘In general, C++ implementations obey the zero-overhead principle: What you don’t use, you don’t pay for. And further: What you do use, you couldn’t hand code any better.’ Bjarne Stroustrup

Thin vs Thick

- OS stream: byte sequence with `write()` and `read()`
- C++ stream: byte sequence with formatting, locale, iomanip to an OS stream or memory
- C++ streams are a replacement of *`xprintf`* & *`xscanf`*
- Thick as a standard I/O system library
- Do not give access to the handles (OS layer)

Libraries and not frameworks

- Only pay for what you use
- Do not enforce a design
- Do not conflict with other libraries
- Do not hide the OS

Libraries & OS

Discussed libraries

boost.thread	<thread>	C++11
boost.asio	Networking TS	
boost.process		
boost.filesystem	<filesystem>	C++17

Operating Systems

Windows	
Posix	Linux, Free BSD, Solaris, OSX

Libraries

std::thread

- Based on design of `boost::thread`
- Standardized in C++11
- An RAII wrapper around OS threads

C++ thread expected code

```cpp
class thread {
    native_handle _handle;
public:
    using native_handle = undefined;

    template<typename Func>
    thread(Func && func);

    thread(thread &&);
    thread& operator=(thread &&);

    void join();
    void terminate();
    void detach();

    ~thread();
};
```

C++ thread expected code (Win32)

```cpp
#if defined(_WIN32)
template<typename Func>
void thread::thread(Func&& func)
  : _handle(CreateThread (0,0, +[](Func && f){f();}, &func, 0)) {}

void thread::join() {
  WaitForSingleObject(_handle, INFINITE);
  _handle = nullptr;
}

void thread::terminate() {
  TerminateThread(_handle, 0);
  _handle = nullptr;
}

void thread::detach() {  _handle = nullptr; }

thread::~thread() {
  if (_handle != nullptr)
    terminate();
}
#else
```

C++ thread expected code (posix)

```
template<typename Func>
void thread::thread(Func&& func)
{
  pthread_create(&_handle, nullptr, +[](Func *f){(*f)();}, &func);
}
void thread::join() {
  pthread_join(_handle, nullptr);
  _handle = 0;
}

void thread::terminate() {
  pthread_cancel(_handle);
  _handle = 0;
}

void thread::detach() { _handle = 0; }

thread::~thread() {
  if (_handle != 0)
    terminate();
}
#endif
```

Boost::thread & std::thread

```
class thread {
    native_handle _handle;
public:
    template<typename Func>
    thread(Func && func);

    thread(thread &&);
    thread& operator=(thread &&);
//    void terminate();
    void join();
    void detach();
    ~thread() {
        if (_handle)
            std::terminate();
    }
};
```

- No terminate function?
- std::terminate() in destructor?

Boost::thread & std::thread - rationale

```
 1  mutex mtx;
 2
 3  thread thr{
 4      [&]{
 5          lock_guard<mutex> lock{mtx};
 6          while(true)
 7              this_thread::sleep_for(chrono::seconds(1));
 8      };
 9
10  this_thread::sleep_for(chrono::seconds(5));
11
12  thr.terminate();
13  lock_guard<mutex> lock(mtx);
14  std::cerr << "Is␣this␣line␣reached?" << std::endl;
```

- On windows, line 14 is never reached (no stack unwinding)
- On posix (pthreads), stack unwinding releases the mutex
- Different semantics between Windows / Posix

boost::thread & std::thread - why std::terminate?

- Destructor can't throw
- RAII requires a terminate of the thread

Terminating a thread anyhow

```
struct my_thread : std::thread {
  void terminate() {
#if defined(_WIN32)
    TerminateThread(native_handle(), 0);
#else
    pthread_cancel(native_handle());
#endif
  }
  ~thread() {
    if (joinable()) {
      terminate();
      join();
    }
  }
}
```

boost::thread::interrupt extension

```
using namespace boost;
thread thr{
    [&]{
        try {
            unique_lock<mutex> lock(mtx);
            condition_variable cond;
            cond.wait(lock);
        }
        catch (boost::thread_interrupted&) {}
    };

thr.interrupt();
thr.join();
```

- Consistent semantics
- Not implemented in the OS
- Overhead when not used

Libraries

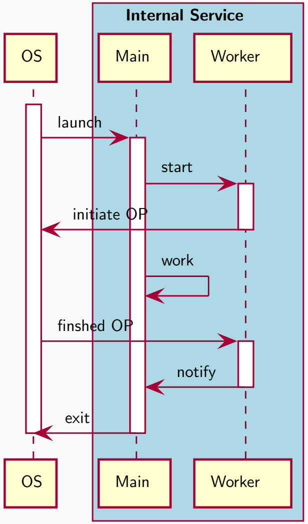
boost::asio

- Basis for the Networking TS
- Provides facilities for sync and async I/O
- TCP, UDP, SerialPort

Asynchronous I/O with threads

```
auto res =
  std::async(std::launch::async,
             os::sync_op);

while(res.wait_for(1_ms)
      == future_status::timeout)
    do_work();

clog << res.get() << endl;
```

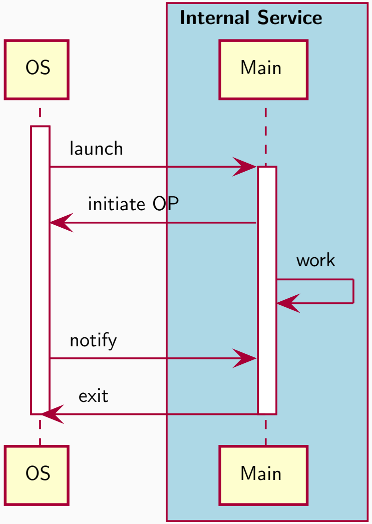
- Threads cause overhead
- One Thread per Stream



Asynchronous I/O with Syscalls

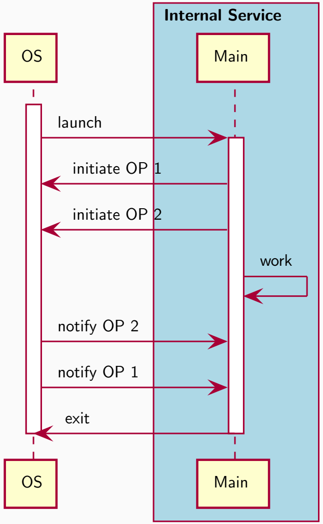
```
auto res = os::async_op();

while(res.wait_for(1_ms)
        == os::timeout)
    do_work();

clog << res.get() << endl;
```

- No overhead
- One Thread for all OPs
- Better design for more OPs



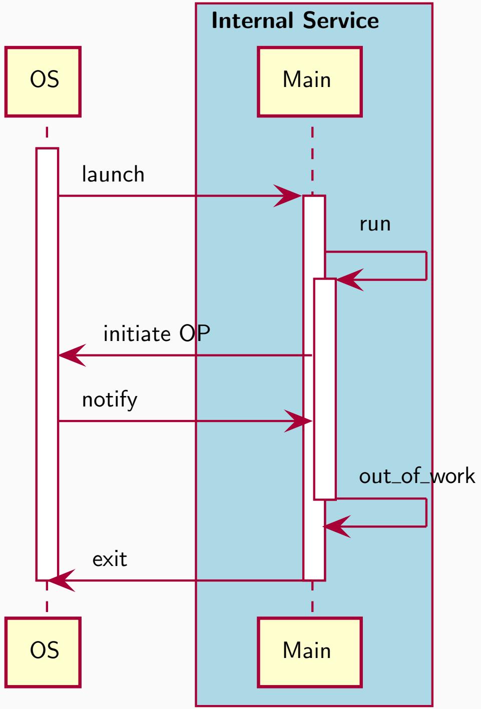
Asynchronous I/O with Syscalls

```
auto res1 = os::async_op1();
auto res2 = os::async_op2();

while((res1.wait_for(1_ms)
        == os::timeout) ||
       (res2.wait_for(1_ms)
        == os::timeout))
    do_work();

clog << res1.get()
     << res2.get() << endl;
```



asio example

```cpp
using resolver = asio::tcp::resolver;
asio::io_context ioc;

resolver resolv{ioc};
resolver::query q{
        "pacificplusplus.com",
        443};

auto fut = resolv.async_resolve(
        q,
        asio::use_future);

ioc.run();

auto res = fut.get();
```



Asynchronous I/O with Syscalls

System Functionality

- Descriptors/Handles
  - Posix: file descriptor (`int`)
  - Windows: stream handle (`void*`)
  - Posix: `read/write`
  - Windows: `ReadFile(Ex)/WriteFile(Ex)`
  - Socket/FileStream/Pipe/SerialPort
- Waitables / Events
  - Posix: signal (`void(*)(int)`)
  - Windows: event handle (`void*`)

Posix Signals

Usage examples

- Child Process Changes
- Interrupt request
- Timers

```cpp
static std::function<void()> handler;

using signal_type = void(*)(int);
signal_type old_sig = nullptr;

handler = [&]{cout << "signal received" << endl; old_sig();};
old_sig = signal(SIGTERM, +[](int){handler();});
```

- Similar to interrupts, i.e. stop execution
- Should just notify, not do work
- Signal handler blocks signal
- Do not scale well

Waitable Objects

Examples

- Process Handles
- Thread Handles
- Events
- Querys, e.g. hostname lookup

Functions

- `WaitForSingleObject(Ex)`
- `WaitForMultipleObjects(Ex)`
- `RegisterWaitForSingleObject`
- `UnregisterWait(Ex)`

Event & RegisterWaitForSingleObject

```
auto handler = []{std::cerr << "Event triggered" << std::endl;};

auto event = CreateEventA(nullptr, false, false, "Pacific++");
HANDLE wait_handle;
//May spawn a thread pool
RegisterWaitForSingleObject(
  &wait_handle,
  event,
  +[](void *p, unsigned char)
  {
    auto h = static_cast<decltype(handler)*>(p);
    (*h)();
  },
  &handler,
  INFINITE,
  WT_EXECUTEONLYONCE);

SetEvent(event);
```

Overlapped & Events

```
HANDLE file_handle = CreateFile("pacifi.c++",
                                GENERIC_READ|GENERIC_WRITE,
                                0, NULL, OPEN_EXISTING,
                                FILE_ATTRIBUTE_NORMAL|
                                    FILE_FLAG_OVERLAPPED, NULL);

OVERLAPPED overlapped{};

//Sync version GetOverlappedResult --> no thread-pool
overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
RegisterWaitForSingleObject(overlapped/*,...*/);//previous slide

auto res =
    WriteFile(file_handle,
              "void foo();", 11,
              NULL, &ovlp);

if (!res && GetLastError() != ERROR_IO_PENDING) {/*errored*/}
```

Overlapped & I/O Completion ports

```
HANDLE file_handle = CreateFile(/*...*/);
OVERLAPPED overlapped{};

auto io_port = CreateIoCompletionPort(
                file_handle, NULL, //ExistingCompletionPort
                NULL, 0);

auto status = WriteFile(/*...*/ &ovlp);

OVERLAPPED * ovl_p;
DWORD bytes_transferred = 0;
ULONG_PTR completion_key = 0;

if (GetQueuedCompletionStatus(
                        io_port, &bytes_transferred,
                        &completion_key, &ovl_p, INFINITE)) {
}
```

Overlapped & I/O Completion ports

```
HANDLE file_handle = CreateFile(/*...*/);
struct my_overlapped : OVERLAPPED { int foo = 42;};
my_overlapped overlapped{};

auto io_port = CreateIoCompletionPort(
               fh, NULL, //ExistingCompletionPort
               NULL, 0);

auto status = WriteFile(/*...*/ &ovlp);

OVERLAPPED * ovl_p;
DWORD bytes_transferred = 0;
ULONG_PTR completion_key = 0;

if (GetQueuedCompletionStatus(
                        io_port, &bytes_transferred,
                        &completion_key, &ovl_p, INFINITE)) {
    static_cast<my_overlapped*>(ovl_p)->foo = 12;
}
```

Summary Windows I/O

- Callbacks for events
- Callback based async I/O per Operation
- Callback may spawn a thread-pool
- Completion Ports for querying and waiting for multiple OPs

Possible on Windows (using the OS thread-pool)

```
auto fut = fd.async_write("foo");
fut.then([fd&]{fd.async_write("bar");});
```

Asynchronous I/O on Posix

- Posix `aio.h` library
  - Callback based
  - Only supports File I/O
  - Used in `boost.afio`
- `SIGIO`
  - Not posix, but some posix implementations
  - Only supports a subset of streams (e.g. sockets, but not pipes)
  - Brings all problems of `signals`
- **A posix implementation cannot be callback based**

Not possible on Posix (without background threads)

```
auto fut = fd.async_write("foo");
fut.then([fd&]{fd.async_write("bar");});
```

select/poll Posix

```
fcntl(STDIN_FILENO,  F_SETFL, O_NONBLOCK);
fcntl(STDOUT_FILENO, F_SETFL, O_NONBLOCK);
fcntl(STDERR_FILENO, F_SETFL, O_NONBLOCK);

struct pollfd fds[3] = {
    {.fd = STDIN_FILENO,  .events = POLLIN},
    {.fd = STDOUT_FILENO, .events = POLLOUT},
    {.fd = STDERR_FILENO, .events = POLLOUT}
  };

write(STDOUT_FILENO, "Some data", 9);
write(STDERR_FILENO, "More data", 9);

auto fd_cnt = ::poll(fds, 3, -1);
```

- select does essentially the same
- POLLOUT triggers when handle is available for a write
- POLLIN triggers when data is available for read

select/poll alternatives

select/poll are slow, because they check all descriptors

OS	Mechanism	Used by asio?
General Posix	poll/select	select_reactor
Linux	epoll	epoll_reactor
BSD	kqueue	kqueue_reactor
Solaris	/dev/poll	dev_poll_reactor
AIX	pollset	

Common functionality?

	Windows	**Posix**
Callbacks as notifications	Yes	Yes
Callbacks for work	Yes	No
Callback per stream	Yes	No
Poll multiple streams	Yes	Yes
Wait for multiple streams	Yes	Yes

`asio::io_context`

- Combining all stream-handles / -descriptors
- `poll` or `IoCompletionPort`
- Polling: `asio::io_context::poll`
- Waiting: `asio::io_context::run`
- Notification/Callback-handling?
  - `PostQueuedCompletionStatus` on windows
  - Event File-Descriptor on posix (self-pipe or `eventfd`)
- Queue work manually with `io_context::post`
- **Introduced a new concept, the `io_context`**
- Little overhead regarding polling
- Requires queueing of operations

asio::io_context

```cpp
namespace bp = boost::process;

asio::io_context ioc;
bp::async_pipe apipe{ioc};

std::string write_buf = "Pacific++";
std::string  read_buf;

std::future<std::size_t> read_fut;
auto write_handler =
    [](auto ec, auto sz) {
        read_buf.resize(sz);
        read_fut = asio::async_read(apipe,
                                    asio::buffer(read_buf),
                                    asio::use_future);
    };
asio::async_write(apipe, asio::buffer(write_buf), write_handler);

ioc.run(); //Start poll/GetQueuedCompletionStatus
```

Libraries

boost::process

- Provides management for processes
- Pipes & Environment

Starting a process

Possible parameters for starting a process

- Command & Args
- Redirection `STDIN`, `STDOUT`, `STDERR`
- Working directory
- Environment
- Platform extensions
- User extensions (at startup)

Variadic interface

```
namespace bp = boost::process;
child c1{bp::exe="./my_app", bp:args={"foo", "bar"},
         bp::start_dir="./working_dir"};

bp::pipe p_stdout;
child c2{bp::exe="./other_app", bp::std_out=p_stdout);
```

Launching a process on Windows

```cpp
namespace bp = boost::process;
child c1{bp::exe="./my_app", bp:args={"foo", "bar"},
          bp::start_dir="./working_dir"};
```

```cpp
STARTUP_INFO startup_info{...};
PROCESS_INFORMATION proc_info;
CreateProcessW(
         L"my_app.exe", L"foo\0bar\0",
         nullptr, nullptr, //process & thread attributes
         FALSE, //inherit handles
         0, //creation flags
         nullptr, //environment
         L"./working_dir",
         &startup_info,
         &proc_info);
```

Launching a process on Posix

```cpp
namespace bp = boost::process;
child c1{bp::exe="./my_app", bp:args={"foo", "bar"},
         bp::start_dir="./working_dir"};
```

```cpp
auto pid = ::fork();

if (pid == -1) { /* error */ }
else if (pid == 0) //child process
{
    chdir("./working_dir");
    const char * args[] = {"foo", "bar", nullptr};
    execv("./my_app", args);
}
```

Launching a process on Posix with stdout

```
bp::pipe p_stdout;
child c2{bp::exe="./other_app", bp::std_out=p_stdout);
```

```
bp::pipe p_stdout;

auto pid = ::fork();

if (pid == -1) { /* error */ }
else if (pid == 0) //child process
{
    dup2(p_stdout.native_sink(), STDOUT_FILENO);
    p_stdout.close();
    const char * args[] = {nullptr};
    execv("./other_app", args);
}

close(p_stdout.native_sink());
```

Launching a process on Windows with stdout

```
bp::pipe p_stdout;
SetHandleInformation(p_stdout.native_sink(),
                     HANDLE_FLAG_INHERIT,
                     HANDLE_FLAG_INHERIT);

STARTUP_INFO startup_info;
startup_info.hStdOutput = p_stdout.native_sink();
startup_info.dwFlags |= STARTF_USESTDHANDLES;

PROCESS_INFORMATION proc_info;
CreateProcessW(
        L"my_app.exe", L"\0\0",
        nullptr, nullptr, //process & thread attributes
        TRUE, //inherit handles
        0, //creation flags
        nullptr, //environment
        nullptr, //Working dir
        &startup_info,
        &proc_info);

CloseHandle(p_stdout.native_sink());
```

Implementation with Initializers

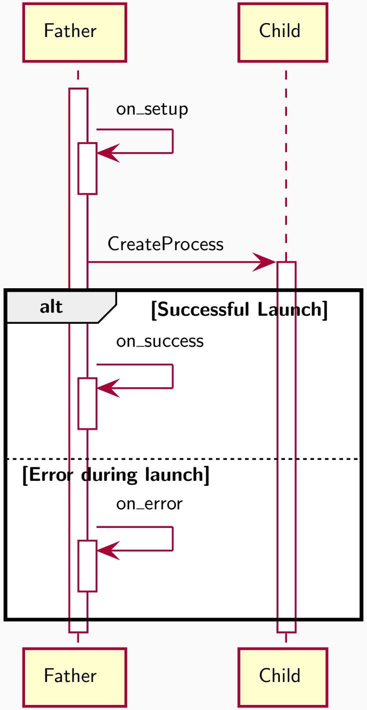
- Arguments build an initializers tuple
- Executor takes the tuple and executes different steps

```
template<typename ...Args> child::child(Args && ...args)
    : child(make_executor(std::tie(args))() {}

template<typename Seq> struct executor {
    Seq seq;
    void operator()() {
        for_each(seq, [this](auto & e){e.on_setup(*this);}
        auto res = CreateProcess(...);

        if (!res) {
            auto err = std::get_last_error();
            for_each(seq, [&](auto & e){e.on_error(*this, err);}
        }
        else
            for_each(seq, [this](auto & e){e.on_success(*this);}
        return child(handle);
    }
};
```

Windows Handler

```
struct handler {

    template <class Executor>
    void on_setup(Executor&) const {}

    template <class Executor>
    void on_error(Executor&, const std::error_code &) const {}

    template <class Executor>
    void on_success(Executor&) const {}
};
```

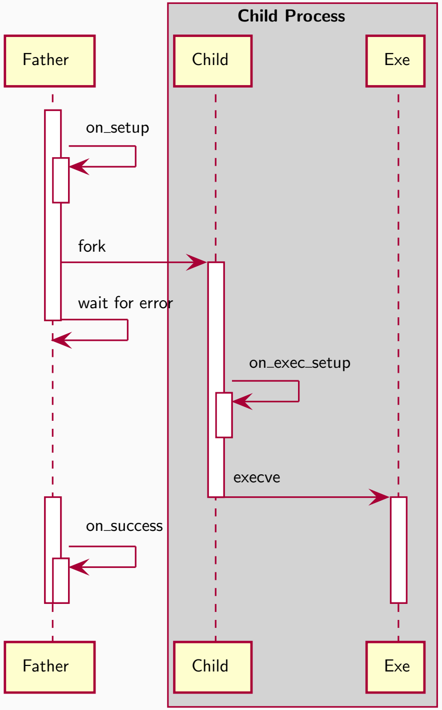
Executor structure windows



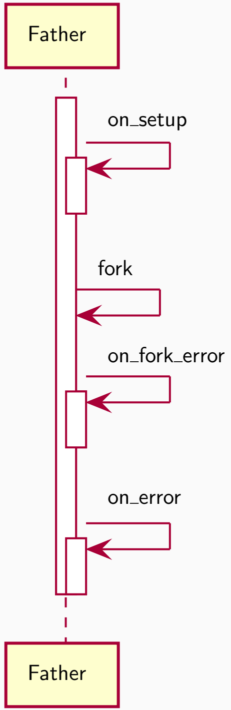
Posix Handler

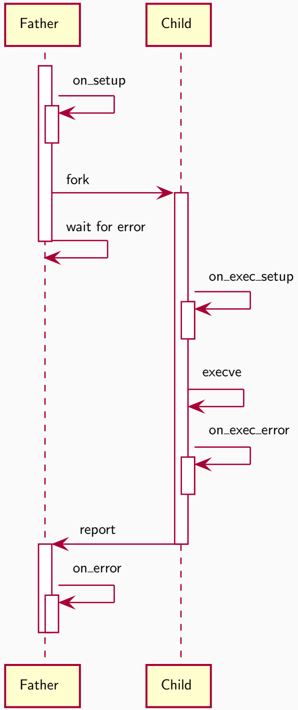
```cpp
struct handler {
    template <class Executor>
    void on_setup(Executor&) const {}

    template <class Executor>
    void on_error(Executor&, const std::error_code &) const {}

    template <class Executor>
    void on_success(Executor&) const {}

    template<typename Executor>
    void on_fork_error(Executor &, const std::error_code&) const {}

    template<typename Executor>
    void on_exec_setup(Executor &) const {}

    template<typename Executor>
    void on_exec_error(Executor &, const std::error_code&) const {}
};
```

Executor structure posix



Executor structure posix





start_dir posix initializer

```cpp
namespace boost::process::detail::posix
{

struct start_dir_init : handler
{
    start_dir_init(const std::string &s) : s_(s) {}

    template <class PosixExecutor>
    void on_exec_setup(PosixExecutor&) const
    {
        ::chdir(s_.c_str());
    }

    const  std::string & str() const {return s_;}
private:
    std::string s_;
};

}
```

start_dir windows initializer

```cpp
namespace boost::process::detail::windows
{

struct start_dir_init : handler
{
    start_dir_init(const std::string &s) : s_(s) {}

    template <class Executor>
    void on_setup(Executor& exec) const
    {
        exec.work_dir = s_.c_str();
    }

    const  std::string & str() const {return s_;}
private:
    std::string s_;
};

}
```

Extension

```
struct handler {

    template <class Executor>
    void on_setup(Executor&) const {}

    template <class Executor>
    void on_error(Executor&, const std::error_code &) const {}

    template <class Executor>
    void on_success(Executor&) const {}
};
```

```
struct my_extension : boost::process::handler {
    template <class Executor>
    void on_setup(Executor &) const {
        std::cout << "Hello Pacific++" << std::endl;
    }
};

bp::child c("foo", my_extension{});
```

Async Pipe

async_pipe

```
asio::io_context ioc;

bp::pipe p;
bp::async_pipe ap{ioc};
```

- On posix, `O_NONBLOCK` can be set on any pipe
- Windows only supports `OVERLAPPED` for named pipes
- `async_pipe` is always named on windows

Async Pipe

Pipe name generation on windows

```
std::string make_pipe_name() {
  std::string name = R"(\\.\pipe\boost_process_auto_pipe_)";
  auto pid = GetCurrentProcessId();
  static std::atomic_size_t cnt{0};
  name += std::to_string(pid);
  name += "_";
  name += std::to_string(cnt++);
  return name;
}
```

- The User should be able to name a pipe
- Posix does not have a named pipe like windows

Named Pipe

Named pipe on posix

```
std::pair<int, int> make_named_pipe(const std::string &name) {
  auto fifo = mkfifo(name.c_str(), 0666 );
  int  read_fd = open(name.c_str(), O_RDWR);
  int write_fd = dup(read_fd);
  return {read_fd, write_fd};
}
```

Using a named pipe

```
#if defined(BOOST_WINDOWS_API)
std::string pipe_name = R"(\\.\pipe\my_pipe)";
#else
std::string pipe_name = "./my_pipe";
#endif
bp::pipe named_pipe(pipe_name);
bp::async_pipe ap(ioc, named_pipe);
```

Summary

- www.github.com/klemens-morgenstern
- klemens.d.morgenstern@gmail.com
- Any questions?