```cpp
// toy-cppcon-2022.cpp                                              -*-C++-*-
// ----------------------------------------------------------------------------
//  Copyright (C) 2022 Dietmar Kuehl http://www.dietmar-kuehl.de
//
//  Permission is hereby granted, free of charge, to any person
//  obtaining a copy of this software and associated documentation
//  files (the "Software"), to deal in the Software without restriction,
//  including without limitation the rights to use, copy, modify,
//  merge, publish, distribute, sublicense, and/or sell copies of
//  the Software, and to permit persons to whom the Software is
//  furnished to do so, subject to the following conditions:
//
//  The above copyright notice and this permission notice shall be
//  included in all copies or substantial portions of the Software.
//
//  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
//  EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
//  OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
//  NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
//  HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
//  WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
//  FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
//  OTHER DEALINGS IN THE SOFTWARE.
// ----------------------------------------------------------------------------

#include <algorithm>
#include <coroutine>
#include <iostream>
#include <optional>
#include <stdexcept>
#include <type_traits>
#include <utility>
#include <vector>

#include <errno.h>
#include <stddef.h>
#include <string.h>

#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <poll.h>

// ----------------------------------------------------------------------------

struct immovable {
    immovable() = default;
    immovable(immovable&&) = delete;
};

// ----------------------------------------------------------------------------

struct task {
    template <typename S>
    struct awaiter {
        using type = typename S::result_t;
        struct receiver {
            awaiter* a;
            friend void set_value(receiver self, auto v) {
                self.a->value.emplace(std::move(v));
                self.a->handle.resume();
            }
            friend void set_error(receiver self, auto e) {
                self.a->error = e;
                self.a->handle.resume();
            }
        };

        using state_t = decltype(connect(std::declval<S>(), std::declval<receiver>()));
        std::coroutine_handle<void> handle;
        state_t                     state;
        std::optional<type>         value;
        std::exception_ptr          error;

        awaiter(S s): state(connect(s, receiver{this})) {}
        bool await_ready() { return false; }
        void await_suspend(std::coroutine_handle<void> handle) {
            this->handle = handle;
            start(state);
        }
        type await_resume() {
            if (error) std::rethrow_exception(error);
            return std::move(*value);
        }
    };

    struct none {};
    using result_t = none;

    struct state_base: immovable {
        virtual void complete() = 0;
    };
    struct promise_type: immovable {
        state_base* state = nullptr;
        task get_return_object() { return {
std::coroutine_handle<promise_type>::from_promise(*this) }; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept {
            if (state) state->complete();
            return {};
        }
        void return_void() {}
        void unhandled_exception() { std::terminate() ; }
        template <typename S>
        awaiter<S> await_transform(S s) { return awaiter<S>(s); }
    };

    template <typename R>
    struct state: state_base {
        std::coroutine_handle<void> handle;
        R                           receiver;

        state(auto&& handle, R receiver): handle(handle), receiver(receiver) {
            handle.promise().state = this;
        }
        friend void start(state& self) {
            self.handle.resume();
        }
        void complete() override final {
            set_value(receiver, none{});
        }
    };

    std::coroutine_handle<promise_type> handle;

    template <typename R>
    friend state<R> connect(task&& self, R receiver) {
        return state<R>(std::move(self.handle), receiver);
    }
};

// -------------------------------------------------------------------------------

struct desc {
    int fd;
    desc(int fd): fd(fd) {}
    desc(desc&& other): fd(other.fd) { other.fd = -1; }
    ~desc() { if (0 <= fd) ::close(fd); }
};

struct io {
    virtual void on_ready() = 0;
};

struct job_base {
    virtual ~job_base() = default;
};

struct receiver {
    job_base* job;
    friend void set_value(receiver self, auto&&) { delete self.job; }
    friend void set_error(receiver self, auto&&) { delete self.job; }
};

template <typename Sender>
struct job
    : job_base {
    decltype(connect(std::declval<Sender>(), std::declval<receiver>())) state;
    job(Sender&& sender)
        : state(connect(std::forward<Sender>(sender), receiver{this})) {
        start(state);
    }
};

struct io_context {
    std::vector<pollfd> fds;
    std::vector<io*>    ops;

    template <typename Sender>
    void spawn(Sender&& sender) {
        new job<Sender>(std::forward<Sender>(sender));
    }

    void add(int fd, short events, io* op) {
        fds.push_back(pollfd{ .fd = fd, .events = events, .revents = 0 });
        ops.push_back(op);
    }

    void run() {
        while (not fds.empty()) {
            if (0 < poll(fds.data(), fds.size(), -1)) {
                for (std::size_t i = fds.size(); i--; ) {
                    if (fds[i].events & fds[i].revents) {
                        ops[i]->on_ready();
                        fds.erase(fds.begin() + i);
                        ops.erase(ops.begin() + i);
                    }
                }
            }
        }
    }
};

struct async_accept {
    using result_t = desc;

    io_context& context;
    int         fd;
    sockaddr*   addr;
    socklen_t*  len;

    template <typename R>
    struct state
        : io {
        R           receiver;
        io_context& context;
        int         fd;
        sockaddr*   addr;
        socklen_t*  len;

        state(R           receiver,
              io_context& context,
              int         fd,
              sockaddr*   addr,
              socklen_t*  len)
            : receiver(receiver), context(context), fd(fd), addr(addr), len(len) {
        }
        friend void start(state& self ) {
            self.context.add(self.fd, POLLIN, &self);
        }
        void on_ready() override {
            int client = accept(fd, addr, len);
            if (client < 0)
                set_error(receiver, std::make_exception_ptr(std::system_error(errno,
std::system_category())));
            else
                set_value(receiver, desc(client));
        }
    };

    template <typename R>
    friend state<R> connect(async_accept self, R receiver) {
        return state<R>(receiver, self.context, self.fd, self.addr, self.len);
    }
};

struct async_readsome {
    using result_t = int;

    io_context& context;
    int         fd;
    char*       data;
    std::size_t len;

    template <typename R>
    struct state
        : io {
        R           receiver;
        io_context& context;
        int         fd;
        char*       data;
        std::size_t len;

        state(R           receiver,
              io_context& context,
              int         fd,
              char*       data,
              std::size_t len)
            : receiver(receiver), context(context), fd(fd), data(data), len(len) {
        }
        friend void start(state& self ) {
            self.context.add(self.fd, POLLIN, &self);
        }
        void on_ready() override {
            int n = read(fd, data, len);
            if (n < 0)
                set_error(receiver, std::make_exception_ptr(std::system_error(errno,
std::system_category())));
            else
                set_value(receiver, n);
        }
    };

    template <typename R>
    friend state<R> connect(async_readsome self, R receiver) {
        return state<R>(receiver, self.context, self.fd, self.data, self.len);
    }
};

struct async_writesome {
    using result_t = int;

    io_context& context;
    int         fd;
    char*       data;
    std::size_t len;

    template <typename R>
    struct state
        : io {
        R           receiver;
        io_context& context;
        int         fd;
        char*       data;
        std::size_t len;

        state(R           receiver,
              io_context& context,
              int         fd,
              char*       data,
              std::size_t len)
            : receiver(receiver), context(context), fd(fd), data(data), len(len) {
        }
        friend void start(state& self ) {
            self.context.add(self.fd, POLLOUT, &self);
        }
        void on_ready() override {
            int n = write(fd, data, len);
            if (n < 0)
                set_error(receiver, std::make_exception_ptr(std::system_error(errno,
std::system_category())));
            else
                set_value(receiver, n);
        }
    };

    template <typename R>
    friend state<R> connect(async_writesome self, R receiver) {
        return state<R>(receiver, self.context, self.fd, self.data, self.len);
    }
};

// ----------------------------------------------------------------------------

int main() {
    desc server{ ::socket(PF_INET, SOCK_STREAM, 0) };

    sockaddr_in addr{ .sin_family = AF_INET, .sin_port = htons(12345), .sin_addr = {
.s_addr = INADDR_ANY } };
    if (::bind(server.fd, (sockaddr*)&addr, sizeof(addr)) < 0
        || ::listen(server.fd, 1) < 0) {
            std::cout << "can't listen on socket: " << strerror(errno) << "\n";
            return EXIT_FAILURE;
    }

    io_context context;

    context.spawn([](io_context& context, desc& server)->task {
        for(int i{}; i != 2; ++i) {
            sockaddr_storage clnt{};
            socklen_t        len{sizeof clnt};
            desc             c{ co_await async_accept{context, server.fd,
(sockaddr*)&clnt, &len} };
```

```
            context.spawn([](io_context& context, desc c)->task {
                char buf[4];
                while (size_t n = co_await async_readsome{context, c.fd, buf, sizeof buf})
{
                    for (size_t o{}, w{1}; o != n && 0 < w; o += w) {
                        w = co_await async_writesome{context, c.fd, buf + o, n - o};
                    }
                }
            }(context, std::move(c)));
        }
    }(context, server));

    context.run();
}
```