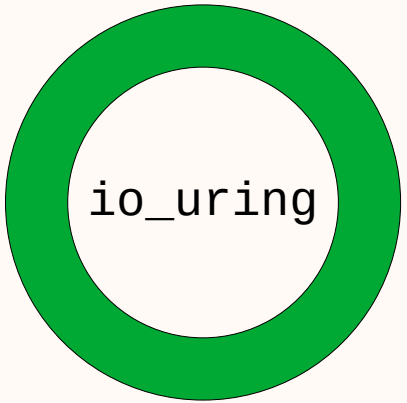
Asynchronous I/O and coroutines for smooth data streaming



```
#include <coroutine>
...
x = co_await source;
co_yield computation(x);
```

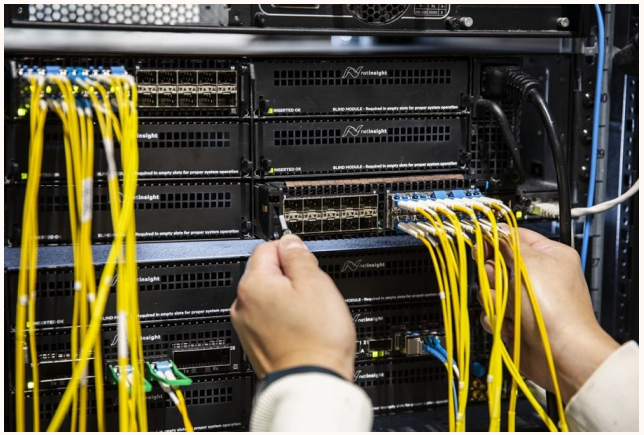
Björn Fahller

Asynchronous I/O and coroutines for smooth data streaming

Asynchronous I/O and coroutines for smooth data streaming

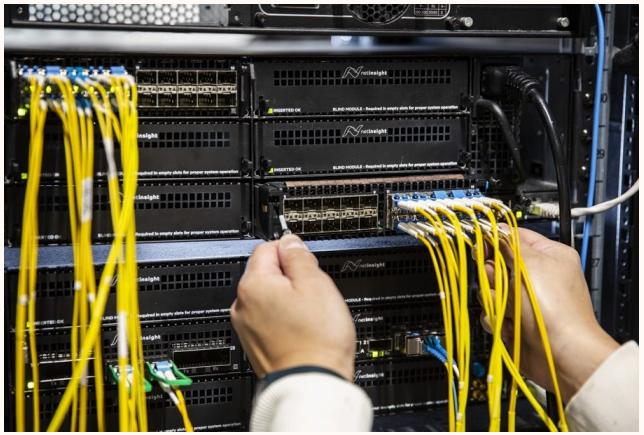
netinsight

Asynchronous I/O and coroutines for smooth data streaming



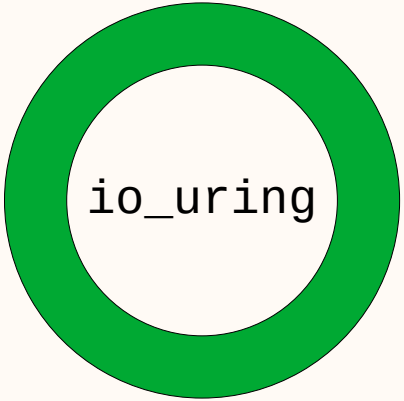
netinsight

Asynchronous I/O and coroutines for smooth data streaming



netinsight

Asynchronous I/O and coroutines for smooth data streaming



```
#include <coroutine>
...
x = co_await source;
co_yield computation(x);
```

Björn Fahller

Linux networking

- Traditionally we use `select`/`poll`/`epoll` to register file descriptors we want to react to
- And `read`/`recv`/`recvmsg`/`recvmmsg` to read the data (and corresponding to send).

```
class poller {
public:
  using worker = std::function<void(std::span<char> data)>;
  void add(int fd, worker w) {
    fds_.push_back({fd, POLLIN, 0});
    cbs_.emplace(fd, std::move(w));
  }
  void wait() {
    auto r = poll(fds_.data(), fds_.size(), -1);
    for (auto& e : fds_) {
      if (e.revents & POLLIN) {
        char buffer[1500];
        auto len = ::read(e.fd, buffer, sizeof(buffer));
        cbs_[e.fd](std::span(buffer).first(len));
      }
    }
  }
private:
  std::vector<pollfd> fds_;
  std::map<int, worker> cbs_;
};
```

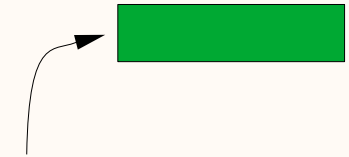
Synchronous I/O with poll/read

Synchronous I/O with poll/read

poll()

Synchronous I/O with poll/read

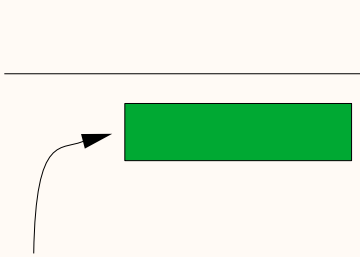
poll()



fill from
kernel

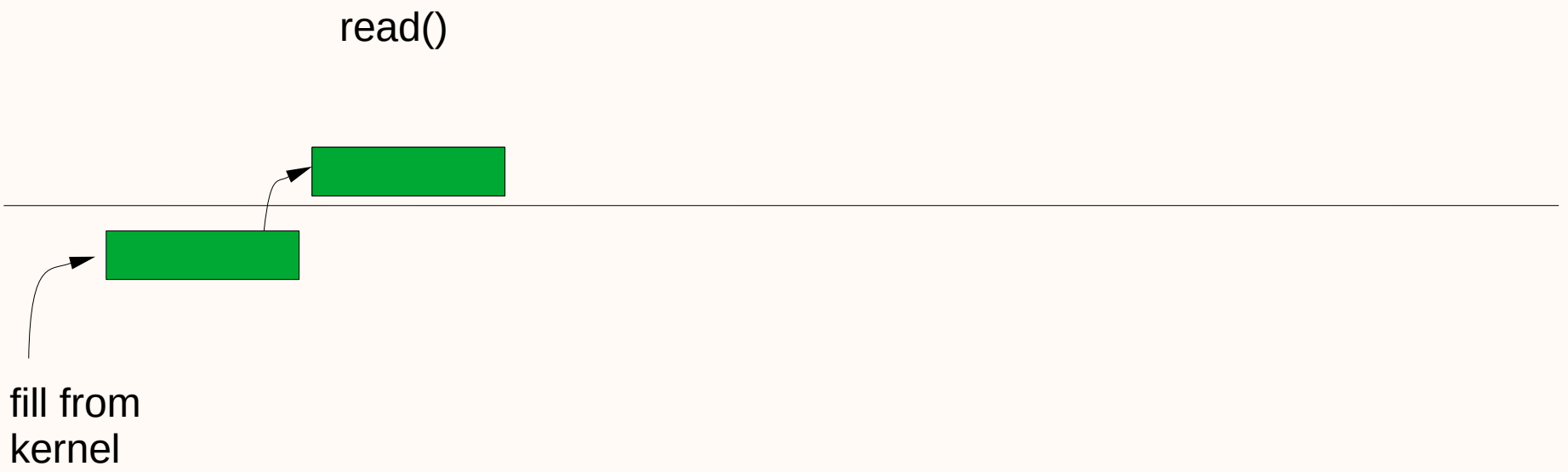
Synchronous I/O with poll/read

read()

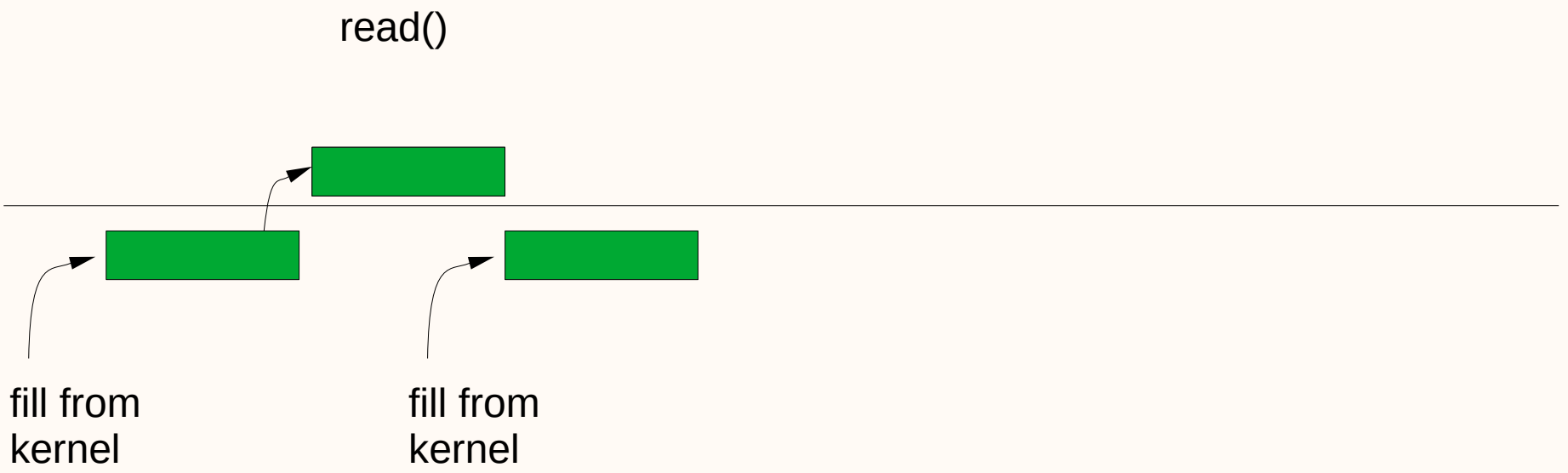


fill from
kernel

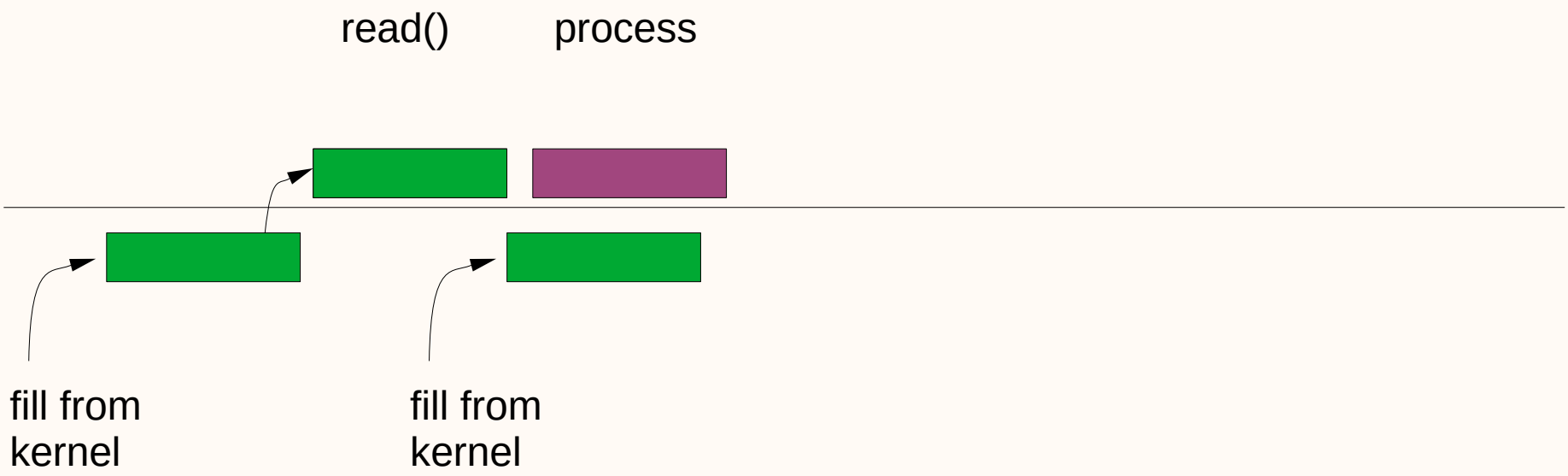
Synchronous I/O with poll/read



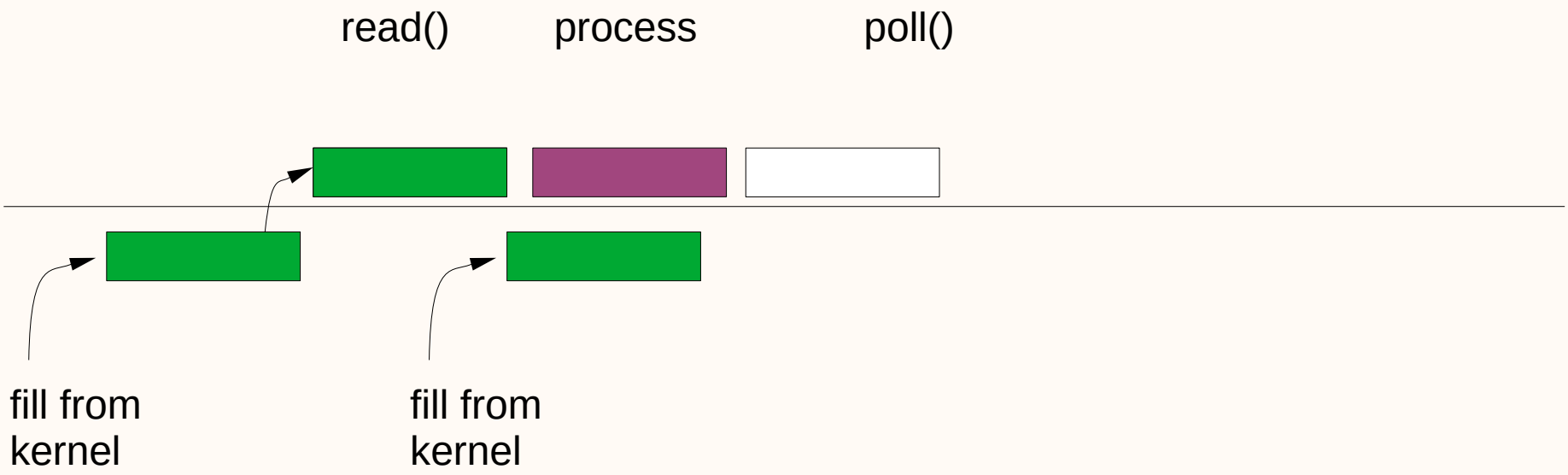
Synchronous I/O with poll/read



Synchronous I/O with poll/read

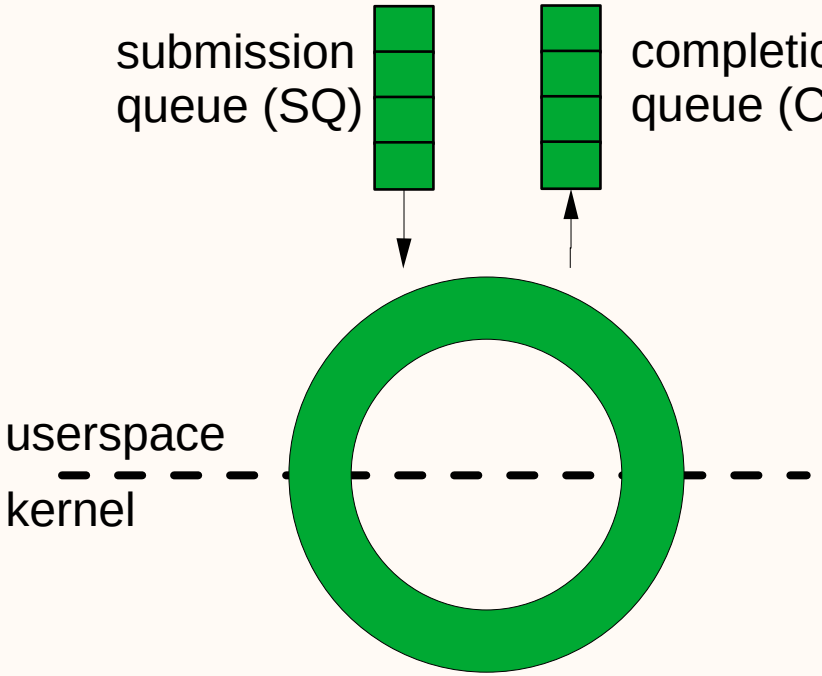


Synchronous I/O with poll/read



Live Demo!

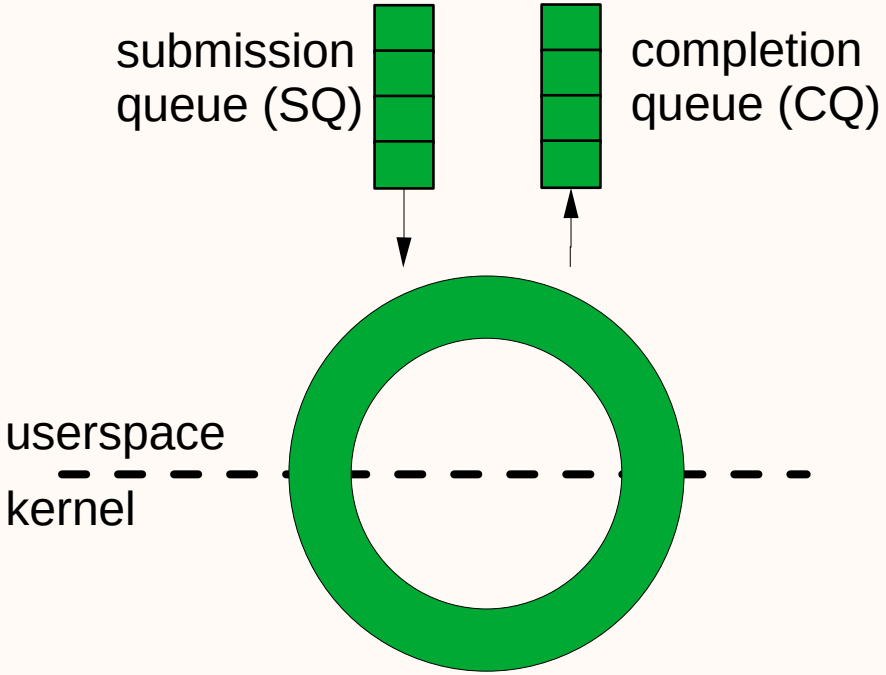
io_uring



IP stack

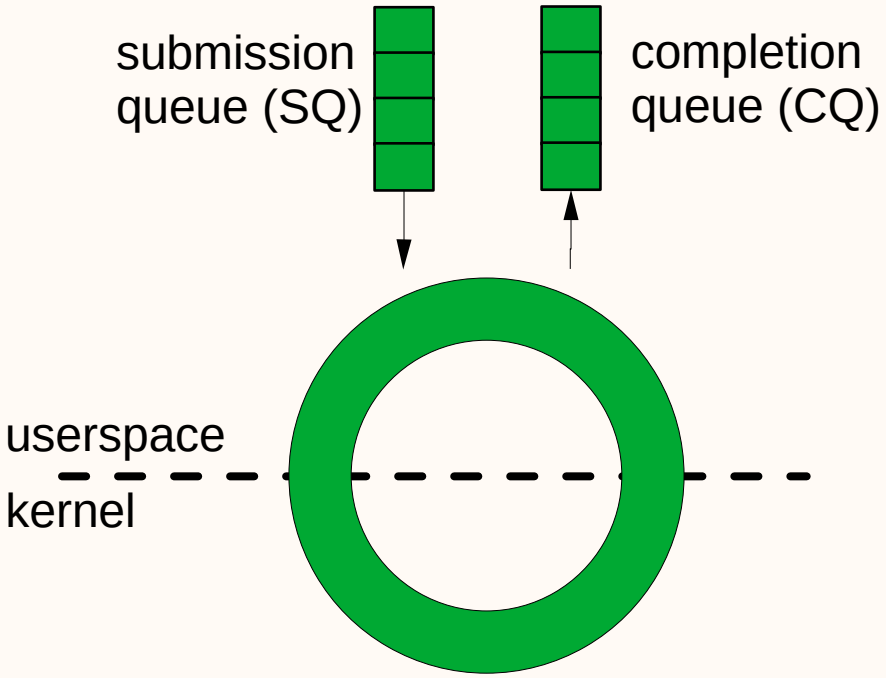
Filesystem

io_uring




```
#include <liburing.h>
```

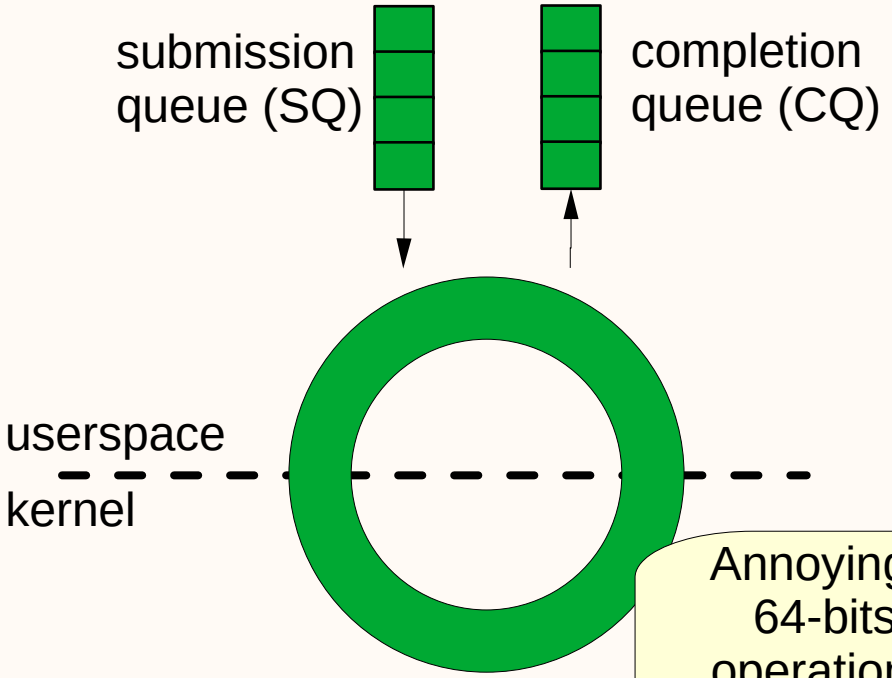
io_uring



Size of queue, i.e. max number of pending entries

```
#include <liburing.h>

io_uring uring;
io_uring_queue_init(8, &uring, 0);

...
```

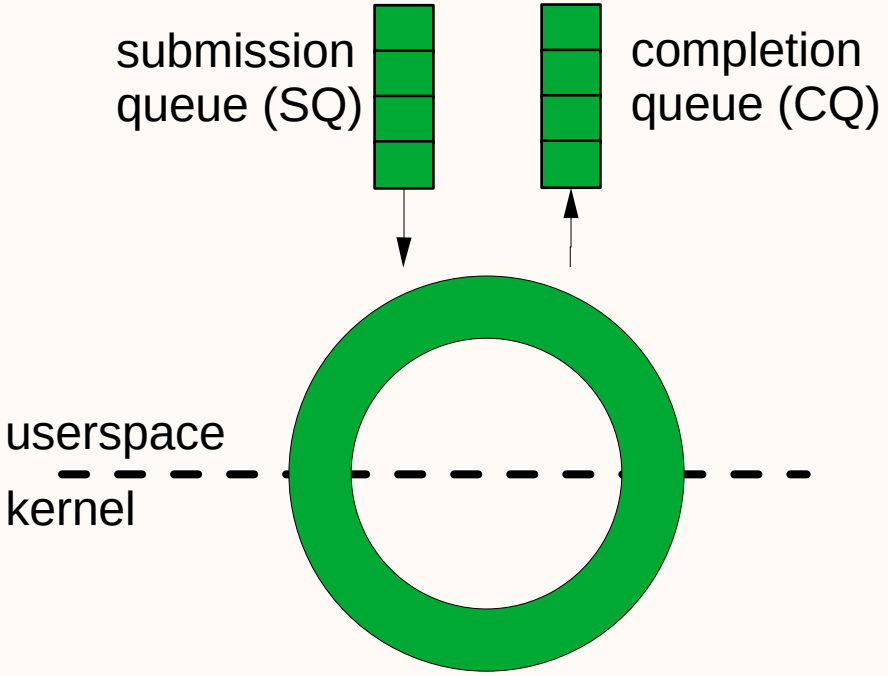
io_uring




```
#include <liburing.h>

io_uring uring;
io_uring_queue_init(8, &uring, 0);

...

auto entry = io_uring_get_sqe(&uring);
io_uring_prep_read(entry, fd, ptr, size, 0);
io_uring_sqe_set_data(entry, work);
```

Annoyingly only one word, 64-bits, to express the operation and the data, so an indirection is almost always needed.

io_uring




```
#include <liburing.h>

io_uring uring;
io_uring_queue_init(8, &uring, 0);

...

auto entry = io_uring_get_sqe(&uring);
io_uring_prep_read(entry, fd, ptr, size, 0);
io_uring_sqe_set_data(entry, work);

...

io_uring_cqe* entry;
auto e = io_uring_wait_cqe(&uring, &entry);
```

uring

```
#include <liburing.h>

class ring
{
public:
  using work = std::function<bool(std::span<char>)>;
  ring();
  ring& operator=(ring&&) = delete;
  ~ring();

  void add(int fd, work);
  void wait();
private:
  struct read_work;
  std::list<read_work> pending_;
  io_uring uring_;
};
```

uring

```
#include <liburing.h>

class ring
{
public:
  using work = std::function<bool(std
  ring();
  ring& operator=(ring&&) = delete;
  ~ring();

  void add(int fd, work);
  void wait();
private:
  struct read_work;
  std::list<read_work> pending_;
  io_uring uring_;
};
```

```
struct ring::read_work {
   work cb_;
   int fd_;
   std::array<char, 1500> buffer_;
};

void ring::add(int fd, work w)
{
  auto& work = pending_.emplace_back();
  work.cb_ = std::move(w);
  work.fd_ = fd;
  auto entry = io_uring_get_sqe(&uring_);
  io_uring_prep_read(entry, fd,
                     work.buffer_.data(),
                     work.buffer_.size(),
                     0);
  io_uring_sqe_set_data(entry, &work);
}
```

uring

```
#include <liburing.h>

class ring
{
public
  usi
  ring
  ring                                elete;
  ~ri

  void add(int fd, work);
  void wait();
private:
  struct read_work;
  std::list<read_work> pending_;
  io_uring uring_;
};
```

The data area to read into needs to be available when preparing the read operation

```
struct ring::read_work {
   work cb_;
   int fd_;
   std::array<char, 1500> buffer_;
};

void ring::add(int fd, work w)
{
  auto& work = pending_.emplace_back();
  work.cb_ = std::move(w);
  work.fd_ = fd;
  auto entry = io_uring_get_sqe(&uring_);
  io_uring_prep_read(entry, fd,
                     work.buffer_.data(),
                     work.buffer_.size(),
                     0);
  io_uring_sqe_set_data(entry, &work);
}
```

uring

```
#include <liburing.h>

class ring
{
public:
  using work = std::function<bool(std
  ring();
                                    ) = delete;
private:
  struct read_work;
  std::list<read_work> pending_;
  io_uring uring_;
};
```

Get a submission queue entry and prepare a read to the buffer from the file descriptor

```
struct ring::read_work {
   work cb_;
   int fd_;
   std::array<char, 1500> buffer_;
};

void ring::add(int fd, work w)
{
  auto& work = pending_.emplace_back();
  work.cb_ = std::move(w);
  work.fd_ = fd;
  auto entry = io_uring_get_sqe(&uring_);
  io_uring_prep_read(entry, fd,
                     work.buffer_.data(),
                     work.buffer_.size(),
                     0);
  io_uring_sqe_set_data(entry, &work);
}
```

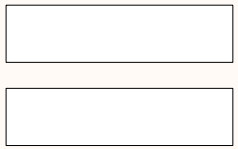
uring

```cpp
#include <liburing.h>

class ring
{
public:
  using work = std::function<bool(std
  ring();
  ring& operator=(ring&&) = delete;
  ~ring();

pr

};
```

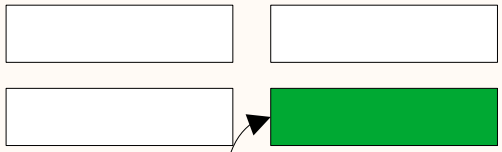
And associate the work struct with the data area and callback with the submission queue entry

```cpp
struct ring::read_work {
   work cb_;
   int fd_;
   std::array<char, 1500> buffer_;
};

void ring::add(int fd, work w)
{
  auto& work = pending_.emplace_back();
  work.cb_ = std::move(w);
  work.fd_ = fd;
  auto entry = io_uring_get_sqe(&uring_);
  io_uring_prep_read(entry, fd,
                     work.buffer_.data(),
                     work.buffer_.size(),
                     0);
  io_uring_sqe_set_data(entry, &work);
}
```

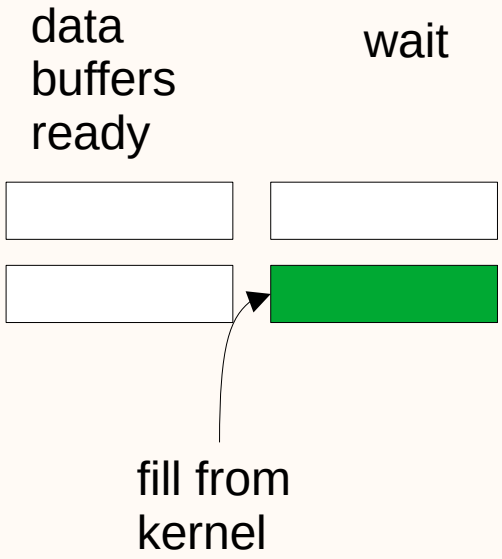
data
buffers
ready

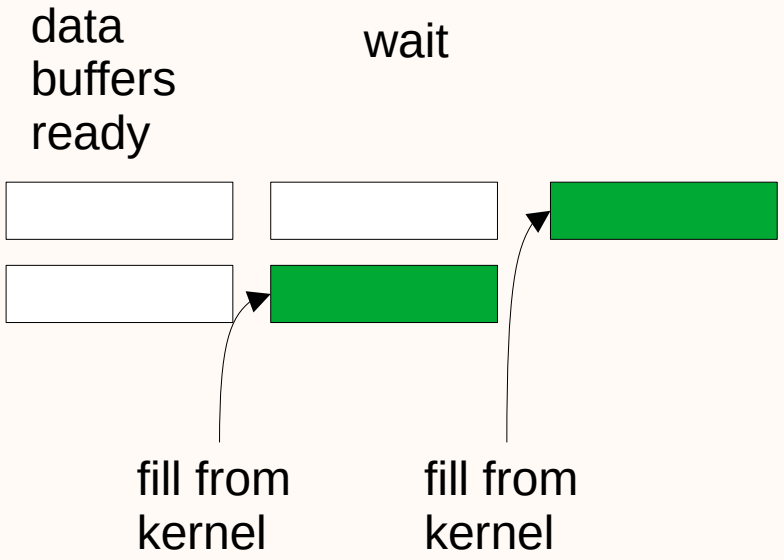


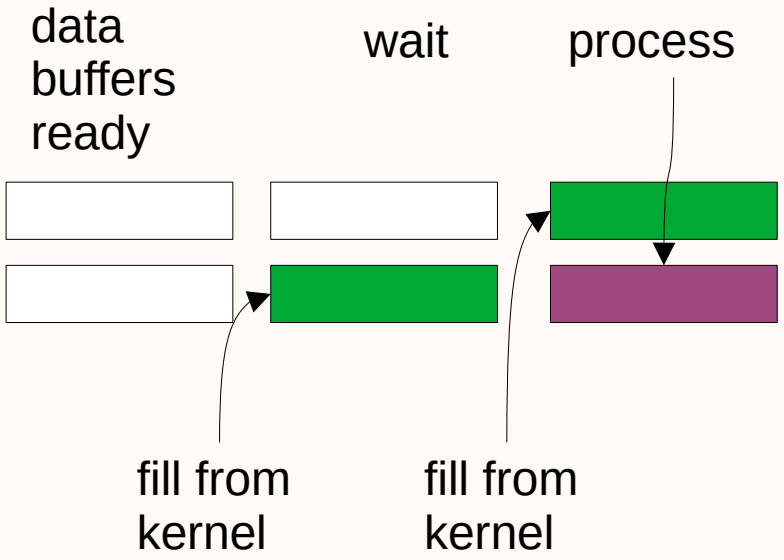
data
buffers
ready

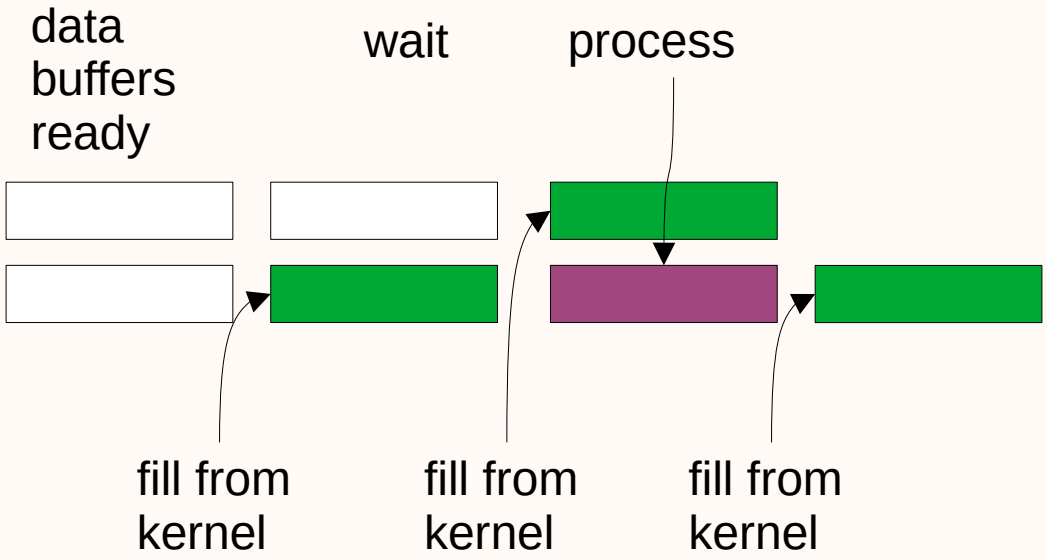


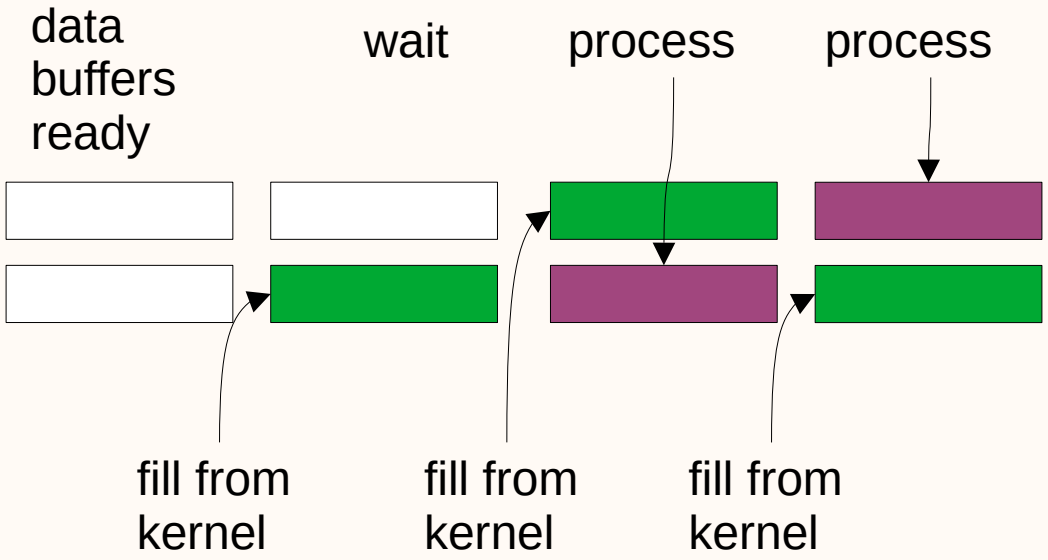
fill from
kernel

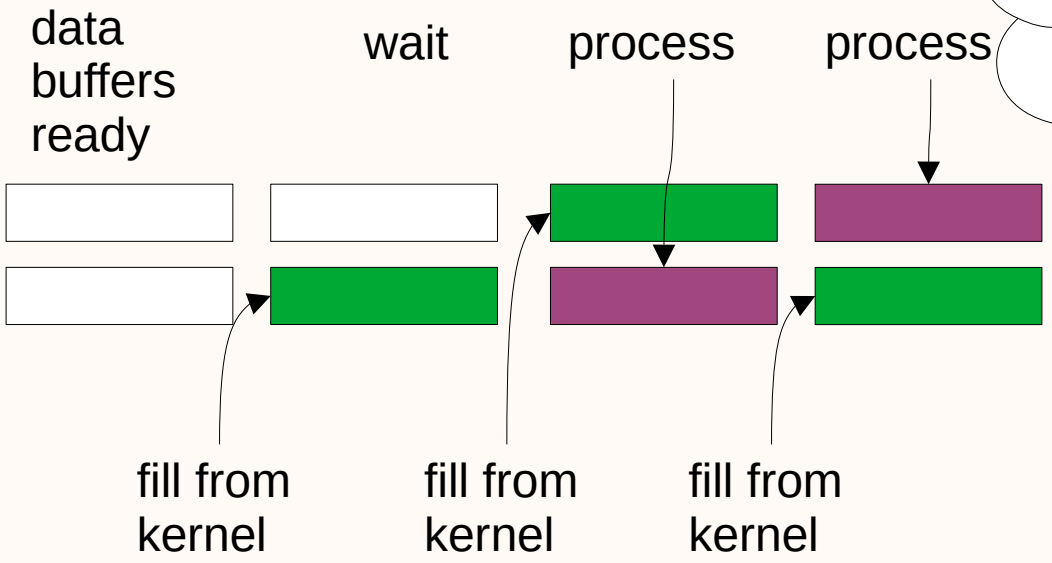

data
buffers
ready
wait
fill from
kernel

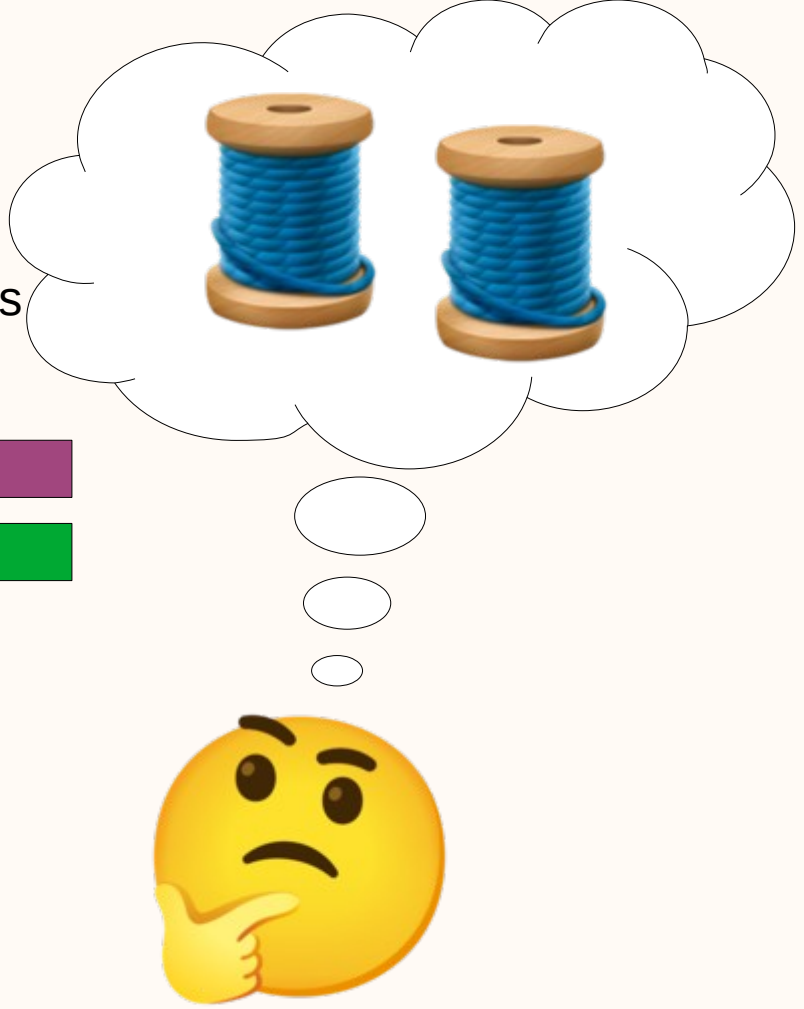

data
buffers
ready
wait
fill from
kernel
fill from
kernel

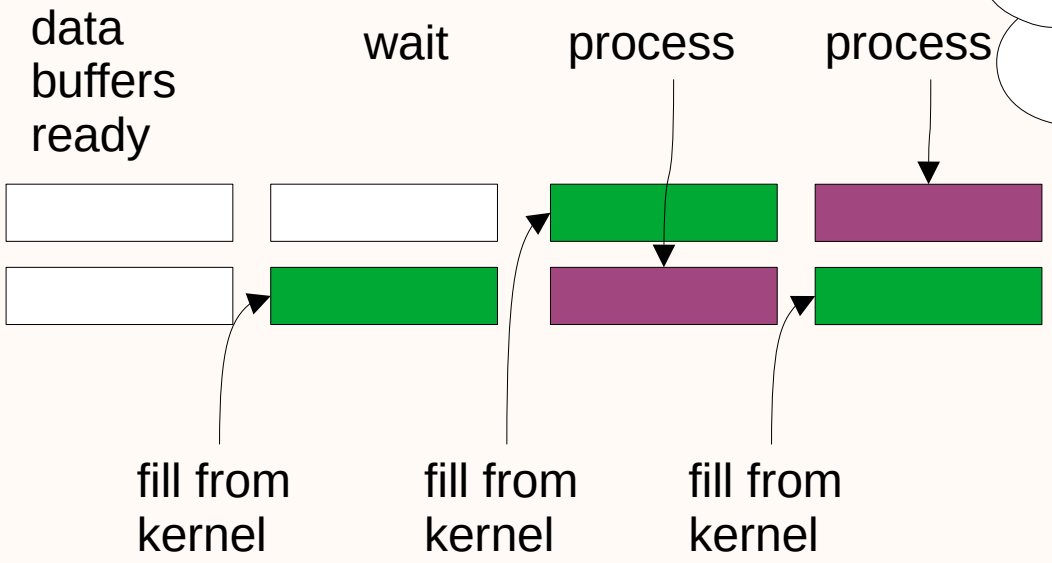

data
buffers
ready
wait
process
fill from
kernel
fill from
kernel

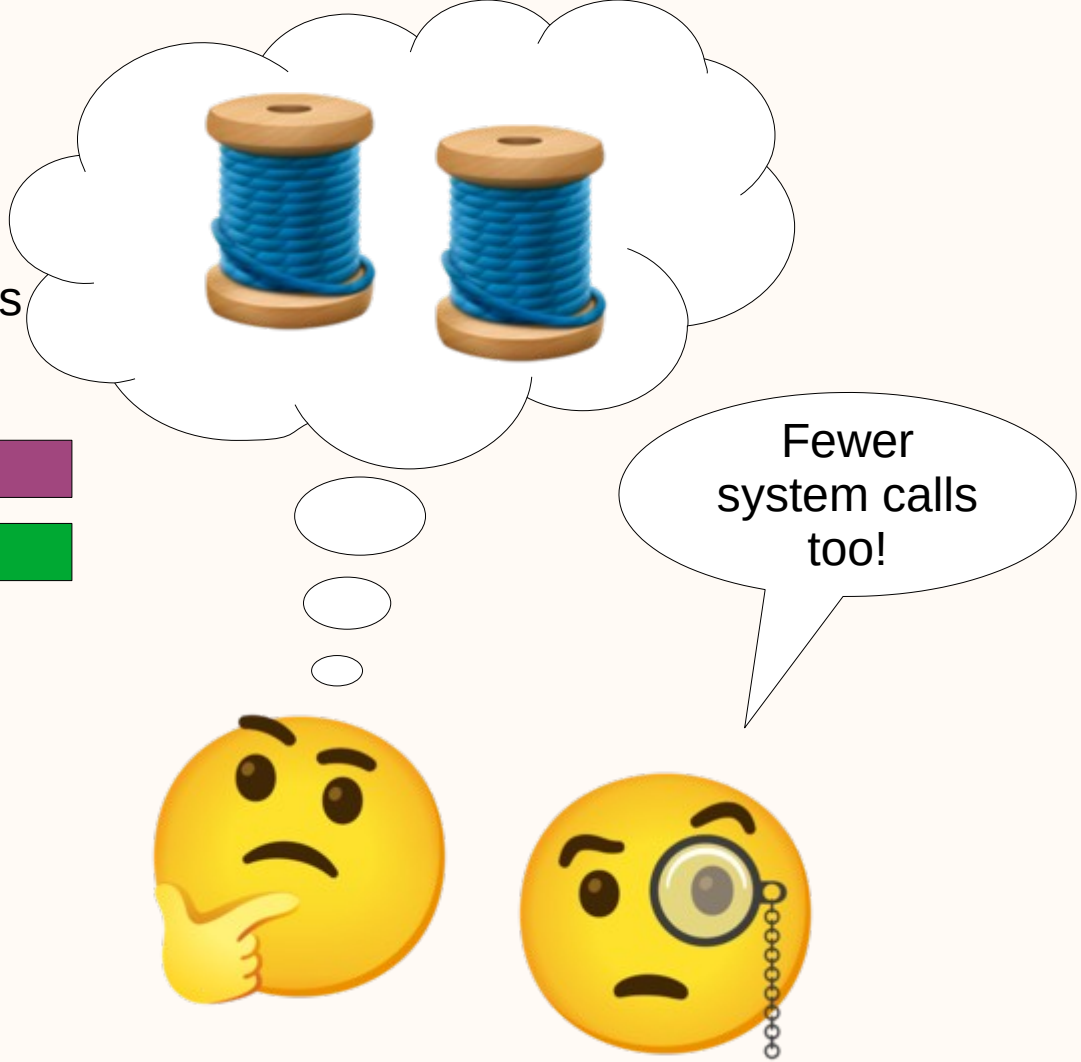

data
buffers
ready
wait
process
fill from
kernel
fill from
kernel
fill from
kernel


data
buffers
ready
wait
process
process
fill from
kernel
fill from
kernel
fill from
kernel


data
buffers
ready
wait
process
process
fill from
kernel
fill from
kernel
fill from
kernel




data buffers ready
wait
process
process
fill from kernel
fill from kernel
fill from kernel


Fewer system calls too!

Live Demo!

coroutines

coroutines

Offers a way for you to write asynchronous code as if they were continuous loops

coroutines

Offers a way for you to write asynchronous code as if they were continuous loops

Language support from C++20

coroutines

Offers a way for you to write asynchronous code as if they were continuous loops

Language support from C++20

Compiler magic converts it to something else, via types that you must write

coroutines

Offers a way for you to write asynchronous code as if they were continuous loops

Language support from C++20

Compiler magic converts it to something else, via types that you must write

- And they’re mindbogglingly hard to understand

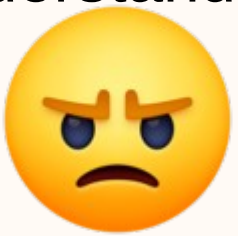
coroutines

Offers a way for you to write asynchronous code as if they were continuous loops

Language support from C++20

Compiler magic converts it to something else, via types that you must write

- And they’re mindbogglingly hard to understand
- And the standard library doesn’t help



coroutines

```
for (;;) {
    …
}
```



```
for (;;) {
    …
}
```

coroutines

```
for (;;) {
    …
}
```



```
for (;;) {
    …
}
```

Suspend execution

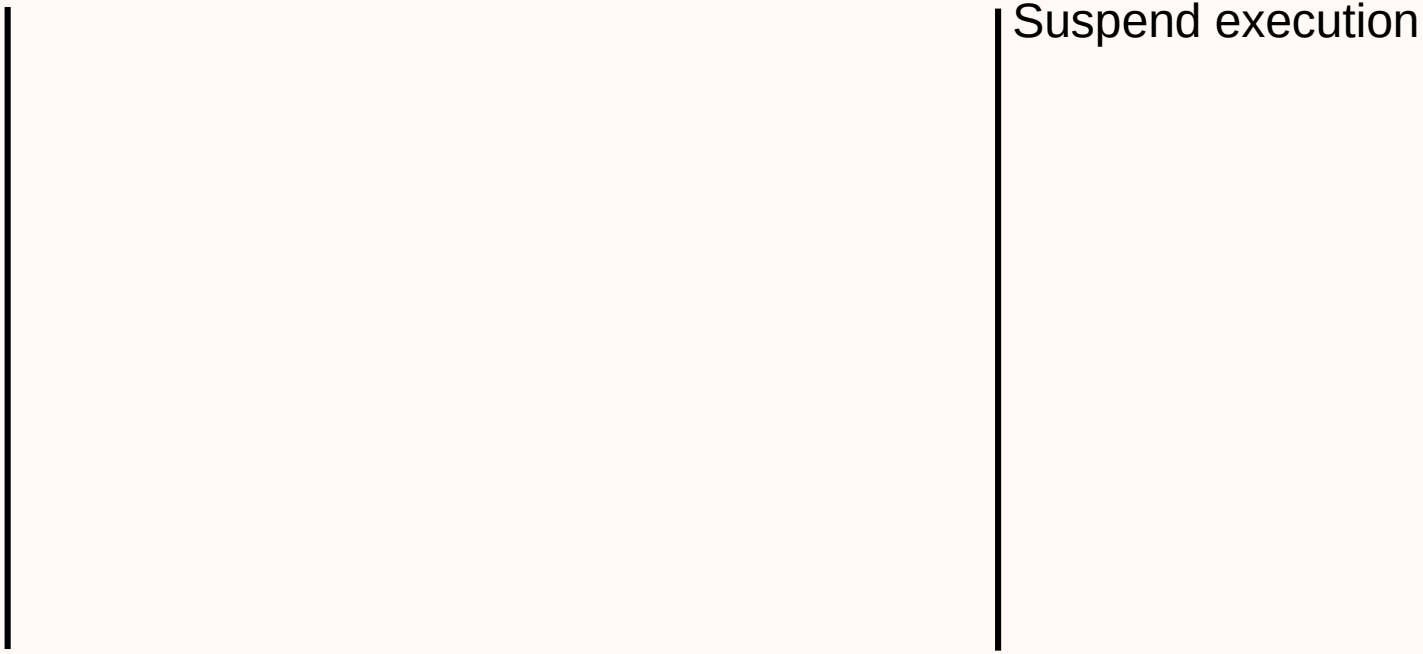
coroutines

```
for (;;) {
    …
}
```

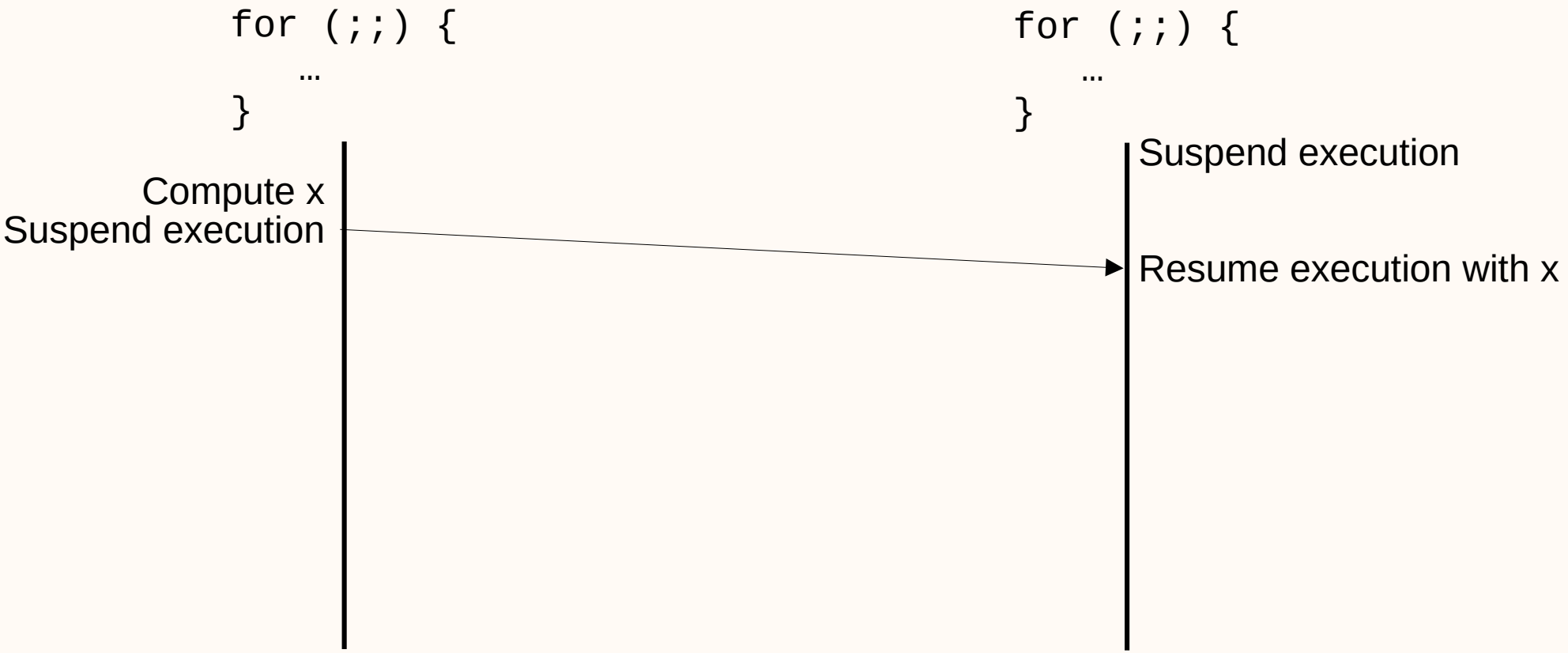
Compute x

```
for (;;) {
    …
}
```

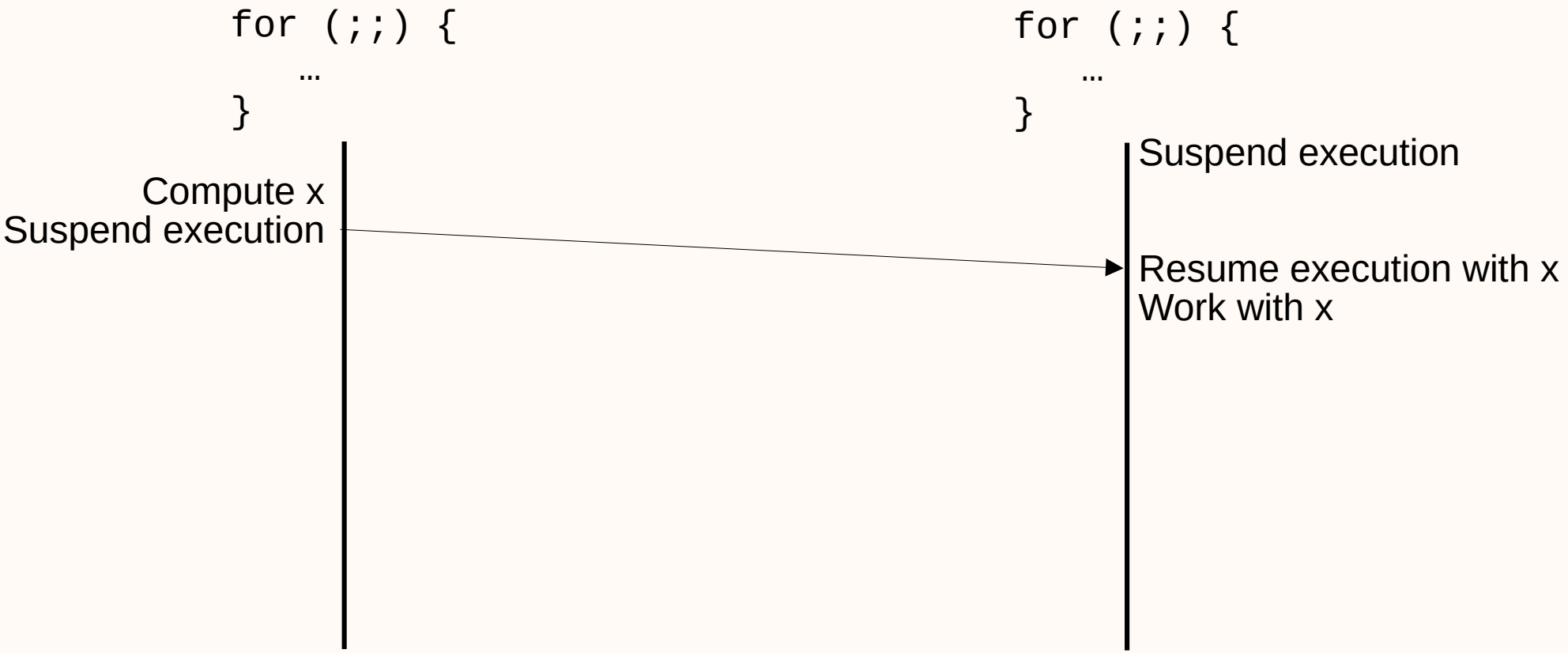
Suspend execution



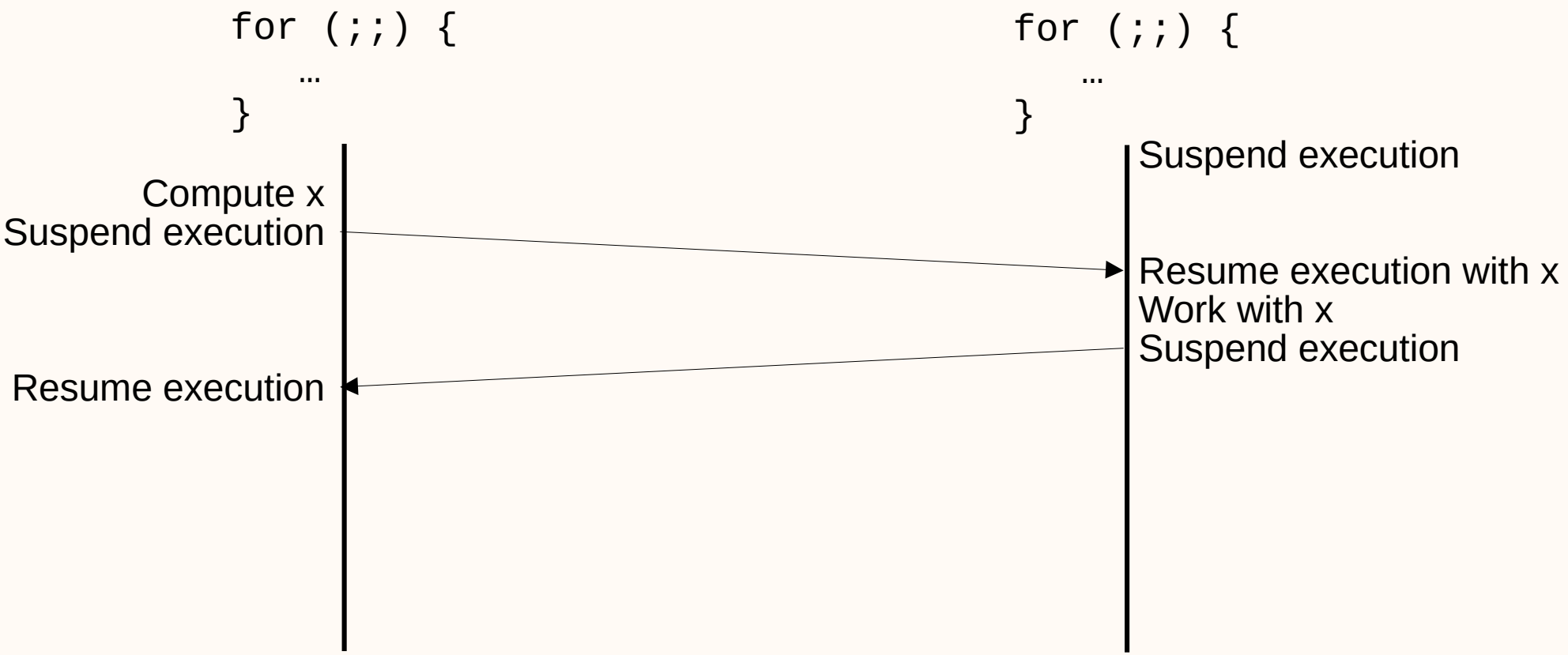
coroutines


for (;;) {
…
}
for (;;) {
…
}
Compute x
Suspend execution
Suspend execution
Resume execution with x

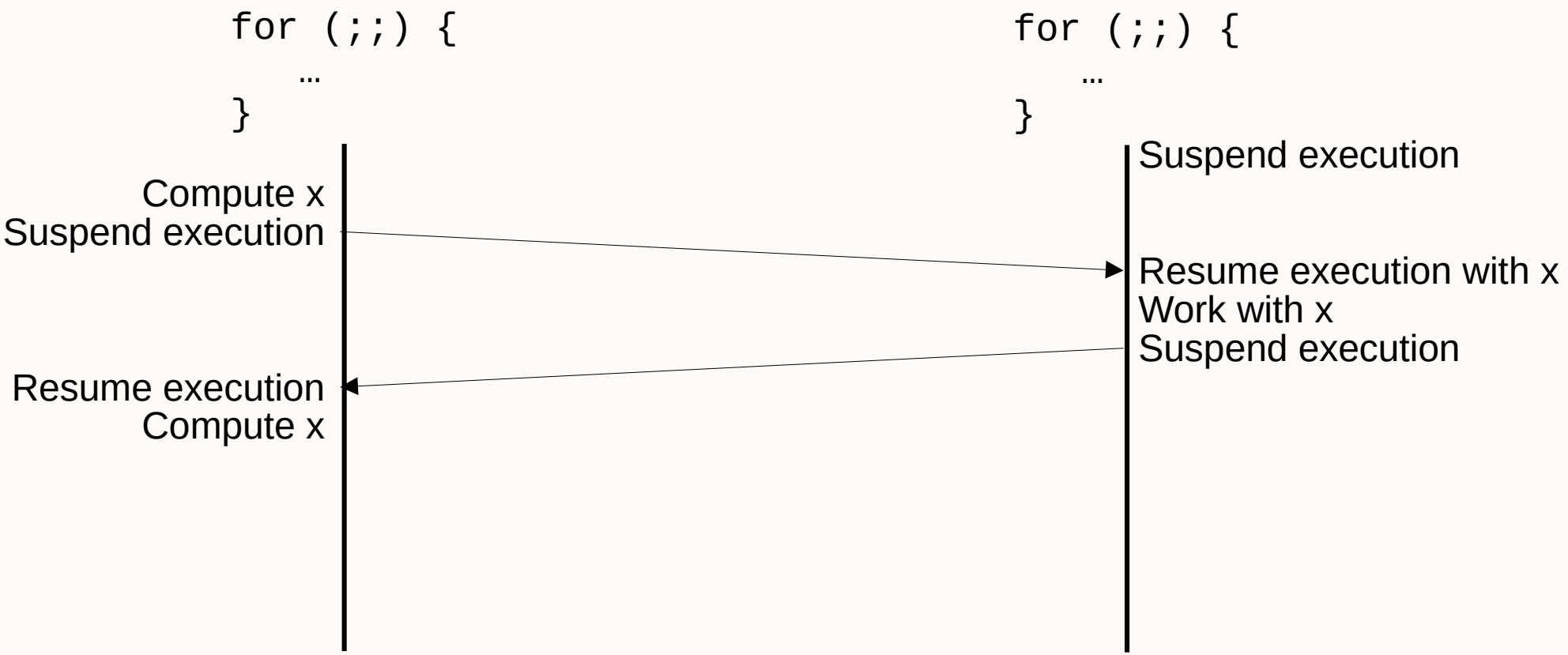
coroutines


for (;;) {
…
}
for (;;) {
…
}
Compute x
Suspend execution
Suspend execution
Resume execution with x
Work with x

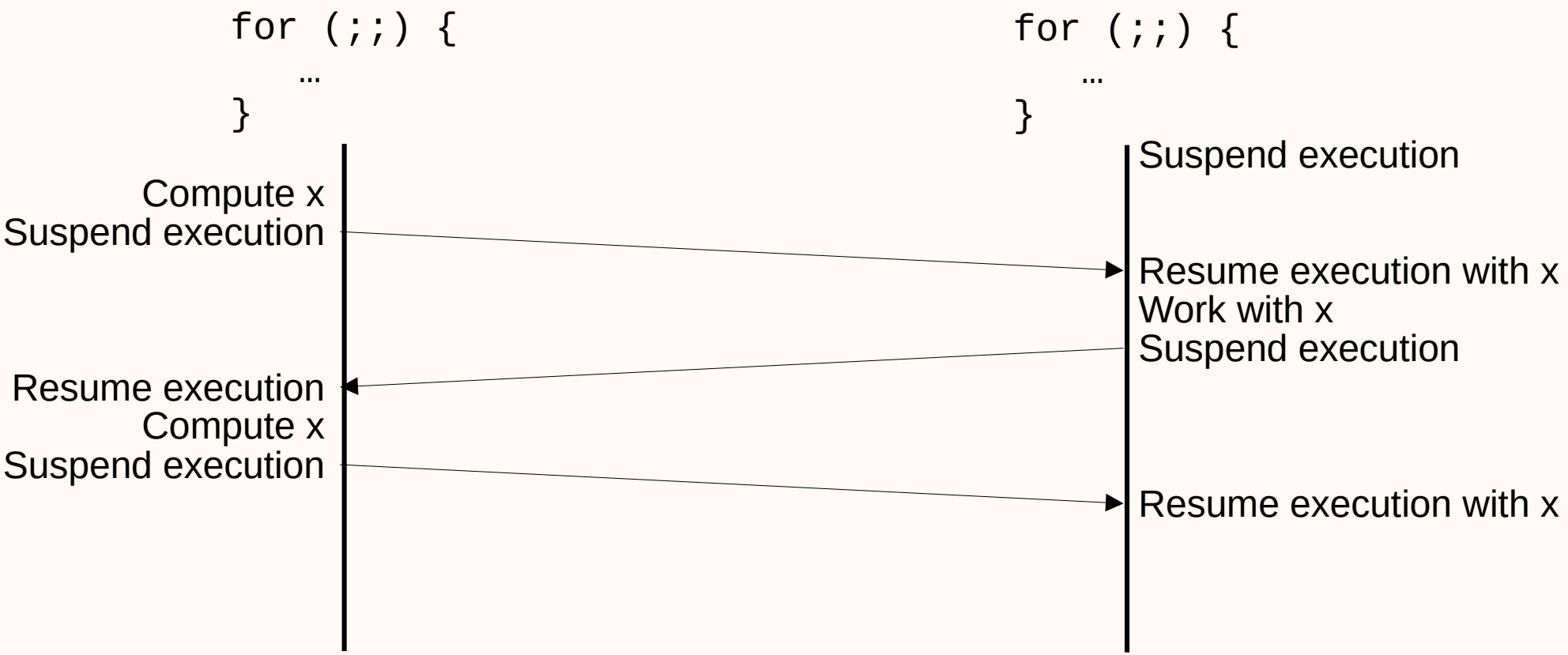
coroutines


for (;;) {
…
}
for (;;) {
…
}
Suspend execution
Compute x
Suspend execution
Resume execution with x
Work with x
Suspend execution
Resume execution

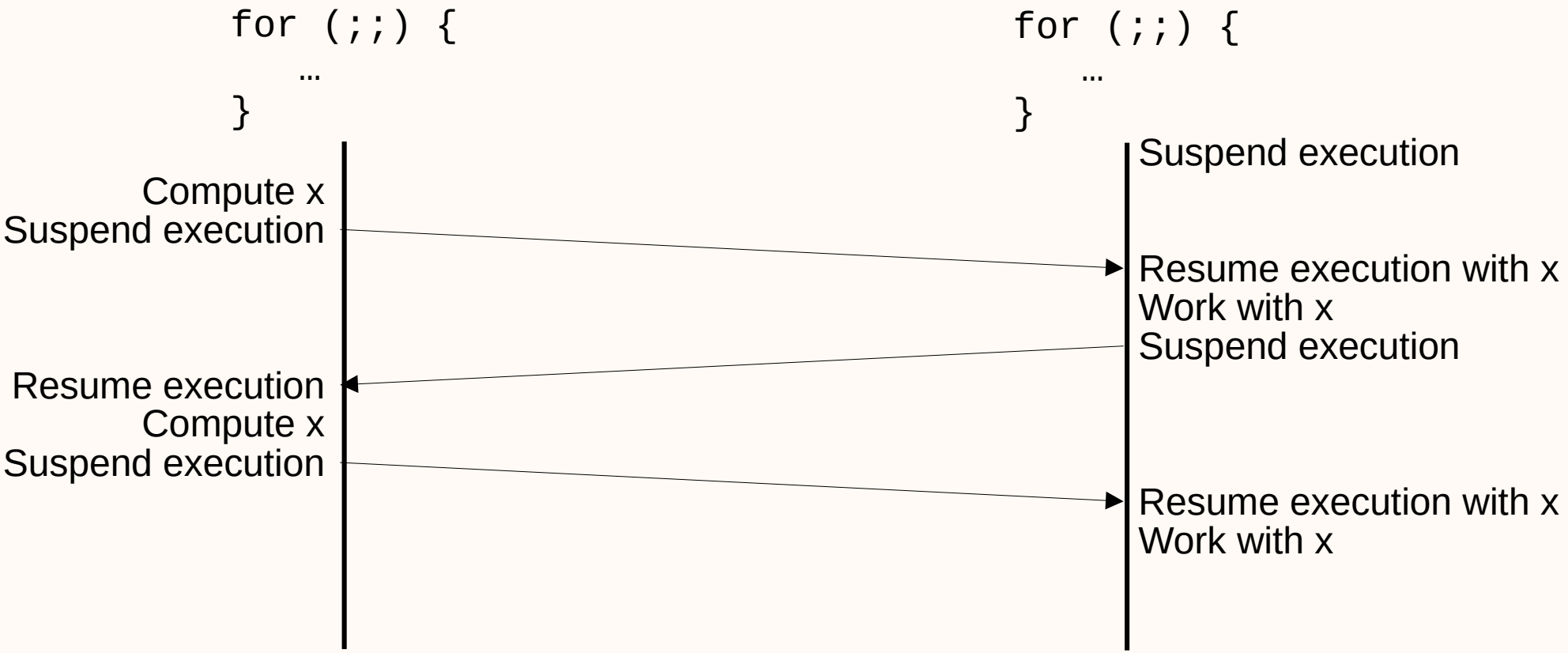
coroutines


for (;;) {
…
}
for (;;) {
…
}
Suspend execution
Compute x
Suspend execution
Resume execution with x
Work with x
Suspend execution
Resume execution
Compute x

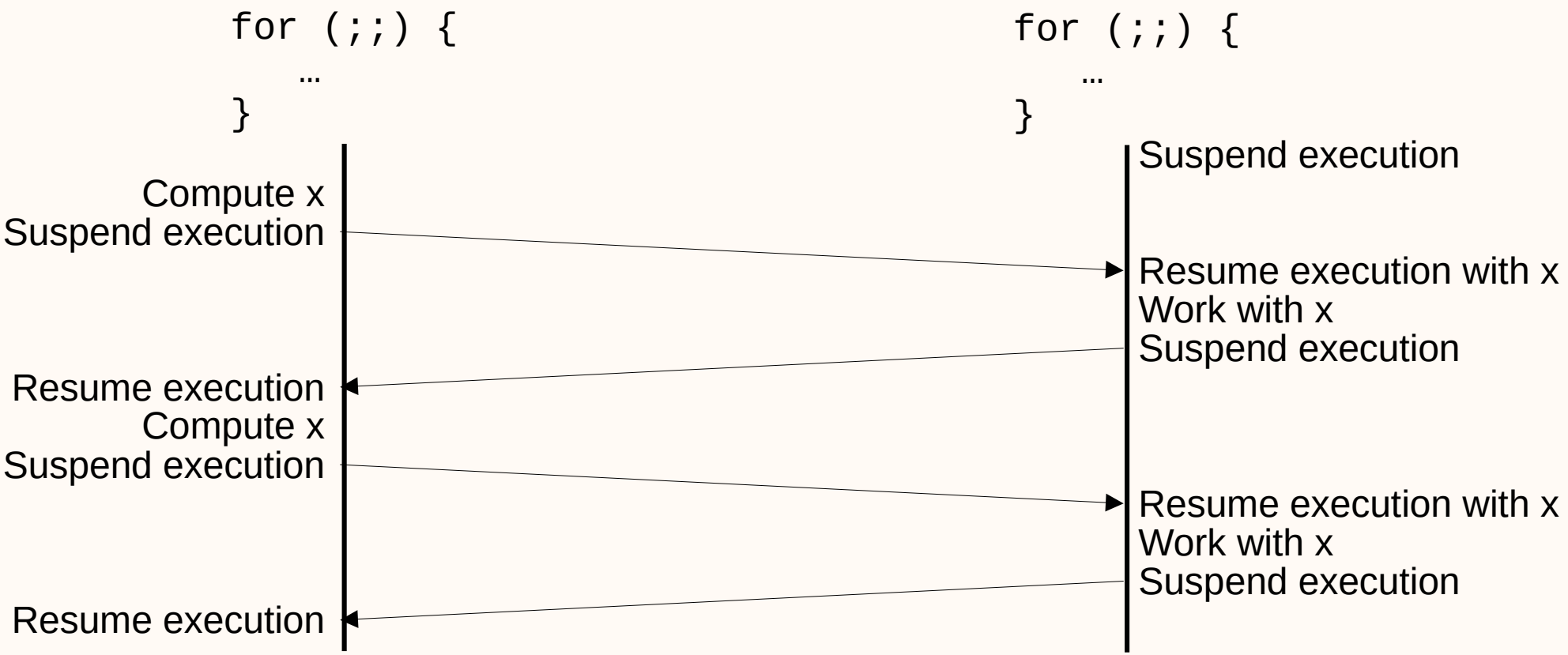
coroutines


for (;;) {
…
}
for (;;) {
…
}
Suspend execution
Compute x
Suspend execution
Resume execution with x
Work with x
Suspend execution
Resume execution
Compute x
Suspend execution
Resume execution with x

coroutines


for (;;) {
…
}
for (;;) {
…
}
Suspend execution
Compute x
Suspend execution
Resume execution with x
Work with x
Suspend execution
Resume execution
Compute x
Suspend execution
Resume execution with x
Work with x

coroutines


for (;;) {
…
}
for (;;) {
…
}
Suspend execution
Compute x
Suspend execution
Resume execution with x
Work with x
Suspend execution
Resume execution
Compute x
Suspend execution
Resume execution with x
Work with x
Suspend execution
Resume execution

coroutines

```
coroutine_type
my_coro(int x, int y, coro_src& src)
{
   int result = 0;
   while (int a = co_await src)
   {
      result += work(a,x,y);
   }
   co_return result;
}

auto coro_obj = my_coro(3, 8, source);
```

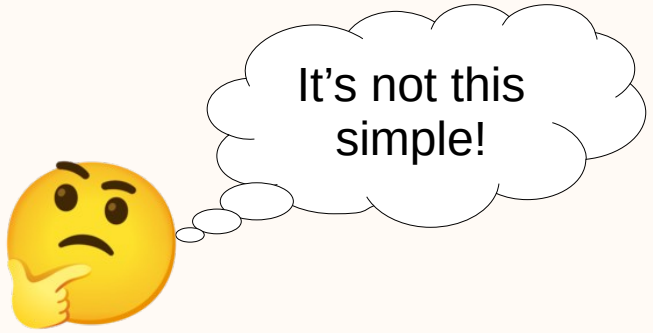
coroutines

```
coroutine_type
my_coro(int x, int y, coro_src& src)
{
   int result = 0;
   while (int a = co_await src)
   {
      result += work(a,x,y);
   }
   co_return result;
}

auto coro_obj = my_coro(3, 8, s
```

Compiler rewrites like

```
class my_coro {
public:
   my_coro(int x_, int y_) : x(x_), y(y_) {}
   int get_result() const { return result; }
   void operator()(int a)
   {
       result += work(a, x, y);
   }
private:
   int x;
   int y;
   int result = 0;
};

auto coro_obj = impl(new my_coro(3, 8), source);
```

coroutines




```
coroutine_type
my_coro(int x, int y, coro_src& src)
{
   int result = 0;
   while (int a = co_await src)
   {
      result += work(a,x,y);
   }
   co_return result;
}

auto coro_obj = my_coro(3, 8, s
```

Compiler rewrites like

```
class my_coro {
public:
   my_coro(int x_, int y_) : x(x_), y(y_) {}
   int get_result() const { return result; }
   void operator()(int a)
   {
       result += work(a, x, y);
   }
private:
   int x;
   int y;
   int result = 0;
};

auto coro_obj = impl(new my_coro(3, 8), source);
```

coroutines

```
coroutine_type
my_coro(int x, int y, coro_src& src)
{
   int result = 0;
   while (int a = co_await src)
   {
      result += work(a,x,y);
   }
   co_return result;
}

auto coro_obj = my_coro(3, 8, source);
```

coroutines

```
coroutine_type
my_coro(int x, int y, coro_src& src)
{
   int resu    = 0;
   while (in    co_await src)
   {
      resul          y);
   }
   co_return result;
}

auto coro_obj = my_coro(3, 8, source);
```

We must write this type

coroutines

```
coroutine_type
my_coro(int x, int y, coro_src& src)
{
   int resu   = 0;
   while (in    co_await src)
   {
      resul        y);
   }
   co_return result;
}

auto coro_obj = my_coro(3, 8, source);
```

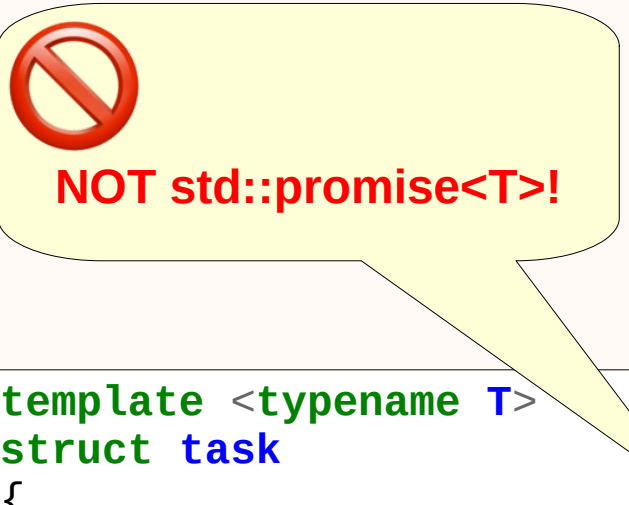
We must write this type

and this type

coroutine return object (task)

```
template <typename T>
struct task
{
  using promise_type = promise<T>;

  auto operator co_await() const noexcept;
private:
  task(promise<T>* p) : m_promise(p) { }
  friend class promise<T>;
  promise_ptr<T> m_promise;
};
```

coroutine return object (task)




```
template <typename T>
struct task
{
  using promise_type = promise<T>;

  auto operator co_await() const noexcept;
private:
  task(promise<T>* p) : m_promise(p) { }
  friend class promise<T>;
  promise_ptr<T> m_promise;
};
```

coroutine return object (task)

NOT std::promise<T>!

```
template <typename T>
struct task
{
  using promise_type = promise<T>;

  auto operator co_await() const noexcept;
private:
  task(promise<T>* p) : m_promise(p) { }
  friend class promise<T>;
  promise_ptr<T> m_promise;
};
```

And we have to write it ourselves

coroutine return object (task)

We will be given a **promise<T>**, allocated by compiler magic.

```
template <typename T>
struct task
{
  using promise_type = promise<T>;

  auto operator co_await() const noexcept;
private:
  task(promise<T>* p) : m_promise(p) { }
  friend class promise<T>;
  promise_ptr<T> m_promise;
};
```

coroutine return object (task)

```
template <typename T>
struct task
{
  using promise_type = promise<T>;

  auto operator co_await() const noexcept
private:
  task(promise<T>* p) : m_promise(p) { }
  friend class promise<T>;
  promise_ptr<T> m_promise;
};
```

And we must destroy it the right way. A smart pointer makes this easy.

coroutine return object (task)

```
struct coro_deleter
{
  template <typename promise>
  void operator()(promise* p) const noexcept {
    using handle = std::coroutine_handle<promise>;
    handle::from_promise(*p).destroy();
  }
};

template <typename T>
using promise_ptr = std::unique_ptr<promise<T>,
                                    coro_deleter>;
```

```
template <typename T>
struct task
{
  using promise_type = promise

  auto operator co_await() const noexcept;
private:
  task(promise<T>* p) : m_promise(p) { }
  friend class promise<T>;
  promise_ptr<T> m_promise;
};
```

coroutines promise

```
template <typename T>
struct promise
{
  task<T> get_return_object() noexcept;
  std::suspend_never initial_suspend() noexcept;
  std::suspend_always final_suspend() noexcept;

  bool is_ready() const noexcept;
  T get();

  void unhandled_exception();
  template <typename U>
  std::suspend_always yield_value(U&& u);
  void return_void();

  std::coroutine_handle<> m_continuation;
  std::optional<T> m_value;
};
```

coroutines promise

```
template <typename T>
struct promise
{
  task<T> get_return_object() noexcept;
  std::suspend_never initial_suspend() noexcept;
  std::suspend_always final_suspend() noexcept;

  bool is_ready() const noexcept;
  T get();

  void unhandled_exception();
  template <typename U>
  std::suspend_always yield_value(U&& u);
  void return_void();

  std::coroutine_handle<> m_continuation;
  std::optional<T> m_value;
};
```

The function that creates the **`task<T>`** object

coroutines promise

```
template <typename T>
struct promise
{
  task<T> get_return_object() noexcept;
  std::suspend_never initial_suspend() noexcept;
  std::suspend_always final_suspend() noexcept;

  bool is_ready() const noexcept;
  T get();

  void unhandled_exception();
  template <typename U>
  std::suspend_always yield_value(U&& u);
  void return_void();

  std::coroutine_handle<> m_continuation;
  std::optional<T> m_value;
};
```

Behaviour at the beginning and end of the life of the coroutine

coroutines promise

```
template <typename T>
struct promise
{
  task<T> get_return_object() noexcept;
  std::suspend_never initial_suspend() noexcept;
  std::suspend_always final_suspend() noexcept;

  bool is_ready() const noexcept;
  T get();

  void unhandled_exception();
  template <typename U>
  std::suspend_always yield_val
  void return_void();

  std::coroutine_handle<> m_continuation;
  std::optional<T> m_value;
};
```

Utility functions for our own implementation

coroutines promise

```
template <typename T>
struct promise
{
  task<T> get_return_object() noexcept;
  std::suspend_never initial_suspend() noexcept;
  std::suspend_always final_suspend() noexcept;

  bool is_ready() const noexcept;
  T get();

  void unhandled_exception();
  template <typename U>
  std::suspend_always yield_value(U&& u);
  void return_void();

  std::coroutine_handle<> m_continuation;
  std::optional<T> m_value;
};
```

Handle to suspended coroutine

coroutines promise

```
template <typename T>
struct promise
{
  task<T> get_return_object() noexcept;
  std::suspend_never initial_suspend() noexcept;
  std::suspend_always final_suspend() noexcept;

  bool is_ready() const noexcept;
  T get();

  void unhandled_exception();
  template <typename U>
  std::suspend_always yield_value(U&& u){
    m_value.emplace(std::forward<U>(u));
    m_continuation.resume();
    return {};
  }
};
```

Resume execution of the coroutine that is suspended waiting for a value.

coroutine return object (task) cont.

```
template <typename T>
struct task
{
   auto operator co_await() const noexcept {
      struct awaitable {
         bool await_ready() const noexcept {
            return m_promise.is_ready();
         }
         void await_suspend(std::coroutine_handle<> next) const noexcept {
            m_promise.m_continuation = next;
         }
         T await_resume() const {
            return m_promise.get();
         }
         promise<T>& m_promise;
      };
      return awaitable{ *m_promise };
   }
};
```

coroutine return object (task) cont.

```
template <typename T>
struct task
{
   auto operator co_await() const noexcept {
      struct awaitable {
         bool await_ready() const noexcept {
            return m_promise.is_ready();
         }
         void await_suspend(std::corout                    noexcept {
            m_promise.m_continuation = r
         }
         T await_resume() const {
            return m_promise.get();
         }
         promise<T>& m_promise;
      };
      return awaitable{ *m_promise };
   }
};
```

Operator is called when code calls:
co_await task;

coroutine return object (task) cont.

```
template <typename T>
struct task
{
   auto operator co_await() const noexcept {
      struct awaitable {
         bool await_ready() const noe
            return m_promise.is_read
         }
         void await_suspend(std::cor                st noexcept {
            m_promise.m_continuation = next;
         }
         T await_resume() const {
            return m_promise.get();
         }
         promise<T>& m_promise;
      };
      return awaitable{ *m_promise };
   }
};
```

It returns an awaitable object that communicates with the **promise<T>**

coroutine return object (task) cont.

```
template <typename T>
struct task
{
   auto operator co_await() const {
      struct awaitable {
         bool await_ready() const noexcept {
            return m_promise.is_ready();
         }
         void await_suspend(std::coroutine_handle<> next) const noexcept {
            m_promise.m_continuation = next;
         }
         T await_resume() const {
            return m_promise.get();
         }
         promise<T>& m_promise;
      };
      return awaitable{ *m_promise };
   }
};
```

Check if the **promise<T>** holds a value

coroutine return object (task) cont.

```
template <typename T>
struct task
{
   auto operator co_await() const noexcept {
      struct awaitable {
         bool await_ready() const noexcept {
            return m_promise.is_ready();
         }
         void await_suspend(std::coroutine_handle<> next) const noexcept {
            m_promise.m_continuation = next;
         }
         T await_resume() const {
            return m_promise.get();
         }
         promise<T>& m_promise;
      };
      return awaitable{ *m_promise };
   }
};
```

Store the calling coroutine as the one to continue when the **promise<T>** gets a value

coroutine return object (task) cont.

```
template <typename T>
struct task
{
   auto operator co_await() const noexcept {
      struct awaitable {
         bool await_ready() const noexcept {
            return m_promise.is_ready();
         }
         void await_suspend(std::coroutine_handle<> next) const noexcept {
            m_promise.m_continuation = next;
         }
         T await_resume() const {
            return m_promise.get();
         }
         promise<T>& m_promis
      };
      return awaitable{ *m_pr
   }
};
```

And finally, on resume, get the value from the **promise<T>**, making it empty again.

coroutine return object (task) cont.

```
template <typename T>
struct task
{
   auto operator co_await() const noexcept {
      struct awaitable {
         bool await_ready() const noexcept {
            return m_promise.is_ready();
         }
         void await_suspend(std::coroutine_handle<> next) const noexcept {
            m_promise.m_continuation = next;
         }
         T await_resume() const {
            return m_promise.get();
         }
         promise<T>& m_promise;
      };
      return awaitable{ *m_promise };
   }
};
```

Live Demo!

Making a computation pipeline

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T>& in)
{
  for (;;) {
    auto v = co_await in;
    if (predicate(v)) {
      co_yield v;
    }
  }
}
```

Making a computation pipeline

```
template <typename T, typename P>
task<T> filter_in(P predicate, ta
{
  for (;;) {
    auto v = co_await in
    if (predicate(v)) {
      co_yield v;
    }
  }
}
```

Calls the **`yield_value()`** member function on the promise of the return type for the coroutine.

Making a computation pipeline

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T>& in)
{
  for (;;) {
    auto v = co_await in;
    if (predicate(v)) {
      co_yield v;
    }
  }
}
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto odd_values = filter_in(is_odd, incoming);
  auto printer = print_all(odd_values);
  for (int i = 0; i < 10; ++i) {
    incoming.get_promise().yield_value(i);
  }
}
```

Making a computation pipeline

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T>& in)
{
  for (;;) {
    auto v = co_await in;
    if (predicate(v)) {
      co_yield v;
    }
  }
}
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto odd_values = filter_in(is_odd, incoming);
  auto printer = print_all(odd_values);
  for (int i = 0; i < 10; ++i) {
    incoming.get_promise().yield_value(i);
  }
}
```

Making a computation pipeline

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T>& in)
{
  for (;;) {
    auto v = co_await in;
    if (predicate(v)) {
      co_yield v;
    }
  }
}
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto odd_values = filter_in(is_odd, incoming);
  auto printer = print_all(odd_values);
  for (int i = 0; i < 10; ++i) {
    incoming.get_promise().yield_value(i);
  }
}
```

Making a computation pipeline

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T>& in)
{
  for (;;) {
    auto v = co_await in;
    if (predicate(v)) {
      co_yield v;
    }
  }
}
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto odd_values = filter_in(is_odd, incoming);
  auto printer = print_all(odd_values);
  for (int i = 0; i < 10; ++i) {
    incoming.get_promise().yield_value(i);
  }
}
```

Making a computation pipeline

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T>& in)
{
  for (;;) {
    auto v = co_await in;
    if (predicate(v)) {
      co_yield v;
    }
  }
}
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto odd_values = filter_in(is_odd, incoming);
  auto printer = print_all(odd_values);
  for (int i = 0; i < 10; ++i) {
    incoming.get_promise().yield_value(i);
  }
}
```

Live Demo!

io_uring + coroutines =

io_uring + coroutines =

We’ve seen how:

io_uring + coroutines =

We've seen how:

- `io_uring` offers asynchronous data

io_uring + coroutines =

We've seen how:

- `io_uring` offers asynchronous data
- Calling `yield_value()` on a coroutine promise pushes data through the coroutine pipeline

io_uring + coroutines =

We've seen how:

- `io_uring` offers asynchronous data
- Calling `yield_value()` on a coroutine promise pushes data through the coroutine pipeline
- How to read values from an upstream coroutine with `co_await`

io_uring + coroutines =

We've seen how:

- `io_uring` offers asynchronous data
- Calling `yield_value()` on a coroutine promise pushes data through the coroutine pipeline
- How to read values from an upstream coroutine with `co_await`
- How to forward values downstream with `co_yield`

io_uring + coroutines =

We've seen how:

- `io_uring` offers asyn[illegible]ata
- Calling `yield_va`[illegible]e promise pushes data through t[illegible]peline
- How to read values from an upstream coroutine with `co_await`
- How to forward values downstream with `co_yield`

Let's put the pieces together

uring + coroutines

```
auto to_promise = [](auto& promise) {
  return [&](auto packet) {
    promise.yield_value(packet);
    return true;
  };
};
```

Convenient tool to generate a callback that writes to a promise, and thus drives a coroutine pipeline.

uring + coroutines

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
task<std::span<char>>
count_packet_data(uint64_t& packets,
                  uint64_t& bytes,
                  task<std::span<char>>& in)
{
  for (;;) {
    auto packet = co_await in;
    ++packets;
    bytes += packet.size();
    co_yield packet;
  }
}
```

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
task<std::string> to_string(task<std::span<char>>& in)
{
  for (;;) {
    auto packet = co_await in;
    co_yield std::string{packet.begin(), packet.end()};
  }
}
```

```
int main()
{
  uint64_t num_bytes =
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
task<std::string>
strip_trailing_newline(task<std::string>& in)
{
  for (;;) {
    auto s = co_await in;
    while (s.ends_with("\n")) {
      s.resize(s.length() - 1);
    }
    co_yield s;
  }
}
```

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
template <size_t line_length>
task<std::string> concatenate_to(task<std::string>& in)
{
  std::string current_line;
  for (;;) {
    auto next_piece = co_await in;
    if (current_line.length() + next_piece.length() + 1 > line_length) {
      co_yield std::exchange(current_line, next_piece);
    } else if (current_line.empty()) {
      current_line = next_piece;
    } else {
      current_line += " " + next_piece;
    }
  }
}
```

```
int
{
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
task<void> print_lines(std::ostream& os, task<std::string>& in)
{
  for (;;) {
    auto line = co_await in;
    os << ':' << line << ":\n";
  }
}
```

```
int main()
{
  uint64_t num_
  uint64_t num_
  bool done = f

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
...
  auto print_and_exit = [&](auto&&) {
    std::cout << "packets=" << num_packets << " bytes=" << num_bytes << '\n';
    done = true;
    return false;
  };

  ring r;
  auto port4000 = udp_socket("127.0.0.1", 4000);
  auto port4001 = udp_socket("127.0.0.1", 4001);
  r.add(port4000.fd(), to_promise(in_4000.get_promise()));
  r.add(port4001.fd(), print_and_exit);
  while (!done) {
    r.wait();
  }
}
```

Live Demo!

uring + coroutines

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

uring + coroutines

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

Keeping these coroutine objects alive as explicit objects is ugly, annoying and error prone.

uring + coroutines

Can we do better?

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  auto in_4000 = task<std::span<char>>::make();
  auto counted_packets = count_packet_data(num_packets, num_bytes, in_4000);
  auto strings = to_string(counted_packets);
  auto stripped_strings = strip_trailing_newline(strings);
  auto lines = concatenate_to<40>(stripped_strings);
  auto print = print_lines(std::cout, lines);
…
```

Keeping these coroutine objects alive as explicit objects is ugly, annoying and error prone.

pipeline as a value type

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T> in)
{
  for (;;) {
    auto v = co_await in;
    if (predicate(v)) {
      co_yield v;
    }
  }
}
```

pipeline as a value type

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T> in)
{
  for (;;) {
    auto v = co_await in;
    if (predicate(v)) {
      co_yield v;
    }
  }
}
```

By value

pipeline as a value type

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T> in)
{
  for (;;) {
    auto v = co_await in;
    if (predicate(v)) {
      co_yield v;
    }
  }
}
```

```
template <typename T>
task<void> print_all(task<T> in)
{
  for (;;) {
    auto v = co_await in;
    std::cout << v << '\n';
  }
}
```

pipeline as a value type

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T> in)
{
  for (;;) {
    auto v = co_await in;
    if (predicate(v)) {
      co_yield v;
    }
  }
}
```

```
template <typename T>
task<void> print_all(task<T> in)
{
  for (;;) {
    auto v = co
    std::cout
  }
}
```

By value

pipeline as a value type

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T> in)
{
  for (;;) {
    auto v = co_await in;
    if (pre
      co_yi
    }
  }
}
```

```
template <typename T>
task<void> print_all(task<T> in)
{
  for (;;) {
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto &promise = incoming.get_promise();
  auto pipeline = print_all(filter_in(is_odd,
                                      std::move(incoming));
  for (int i = 0; i < 10; ++i) {
    promise.yield_value(i);
  }
}
```

pipeline as a value type

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T> in)
{
  for (;;) {
    auto v = co_await in;
    if (pre
      co_yi
    }
  }
}
```

```
template <typename T>
task<void> print_all(task<T> in)
{
  for (;;) {
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto &promise = incoming.get_promise();
  auto pipeline = print_all(filter_in(is_odd,
                                      std::move(incoming)));

  for (int i = 0; i < 10; ++i) {
    promise.yield_value(i);
  }
}
```

pipeline as a value type

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T> in)
{
  for (;;) {
    auto v = co_await in;
    if (pre
      co_yi
    }
  }
}
```

```
template <typename T>
task<void> print_all(task<T> in)
{
  for (;;) {
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto &promise = incoming.get_promise();
  auto pipeline = print_all(filter_in(is_odd,
                                      std::move(incoming)));

  for (int i = 0; i < 10; ++i) {
    promise.yield_value(i);
  }
}
```

pipeline as a value type

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T> in)
{
  for (;;) {
    auto v = co_await in;
    if (pre
      co_yi
    }
  }
}
```

```
template <typename T>
task<void> print_all(task<T> in)
{
  for (;;) {
```

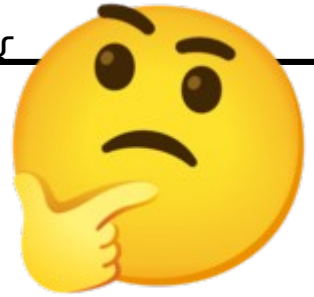
```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto &promise = incoming.get_promise();
  auto pipeline = print_all(filter_in(is_odd,
                                      std::move(incoming)));
  for (int i = 0; i < 10; ++i) {
    promise.yield_value(i);
  }
}
```

pipeline as a value type

But this is not very nice either, is it?

```
template <typename T, typename P>
task<T> filter_in(P predicate, task<T> in)
{
  for (;;) {
    auto v = co_await in;
    if (pre
      co_yi
    }
  }
}
```

```
template <typename T>
task<void> print_all(task<T> in)
{
  for (;;) {
```



```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto &promise = incoming.get_promise();
  auto pipeline = print_all(filter_in(is_odd,
                                      std::move(incoming)));
  for (int i = 0; i < 10; ++i) {
    promise.yield_value(i);
  }
}
```

Another level of indirection

```
auto filter_in = [](auto predicate)
{
   return [predicate]<typename T>(task<T> in) -> task<T>
   {
      for (;;) {
         auto v = co_await in;
         if (predicate(v)) {
            co_yield v;
         }
      }
   };
};
```

Another level of indirection

```
auto filter_in = [](auto predicate)
{
   return [predicate]<typename T>(task<T> in) -> task<T>
   {
      for (;;) {
         auto v = co_await in;
         if (predicate(v)) {
            co_yield v;
         }
      }
   };
};
```

By value

Another level of indirection

```
auto filter_in = [](auto predicate)
{
   return [predicate]<typename T>(task<T> in) -> task<T>
   {
      for (;;) {
         auto v = co_await in;
         if (predicate(v)) {
            co_yield v;
         }
      }
   };
};
```

By value

Another level of indirection

```
auto filter_in = [](auto predicate)
{
   return [predicate]<typename T>(task<T> in) -> task<T>
   {
      for (;;) {
         auto v = co_await in;
         if (predicate(v)) {
            co_yield v;
         }
      }
   };
};
```

Explicit return type

Another level of indirection

```
auto filter_in = [](auto predicate)
{
   return [predicate]<typename T>(task<T> in) -> task<T>
   {
      for (;;) {
         auto v = co_
         if (predicat
            co_yield v
         }
      }
   };
};
```

```
auto print_all = [](std::ostream& dest)
{
   return [&dest]<typename T>(task<T> in) -> task<void>
   {
      for (;;)
      {
         auto v = co_await in;
         dest << v << '\n' << std::flush;
      }
   };
};
```

Another level of indirection

```
auto filter_in = [](auto predicate)
{
   return [predicate]<typename T>(task<T> in) -> task<T>
   {
      for (;;) {
         auto v = co_
         if (predicat
            co_yield v
         }
      }
   };
};
```

```
auto print_all = [](std::ostream& dest)
{
   return [&dest]<typename T>(task<T> in) -> task<void>
   {
      for (;;)
      {
         auto v = co_await in;
         dest << v << '\n' << std::flush;
      }
   };
};
```

Capture the stream

Another level of indirection

```
auto filter_in = [](auto predicate)
{
   return [predicate]<typename T>(task<T> in) -> task<T>
   {
      for (;;) {
```

```
auto print_all = [](std::ostream& dest)
{
```

```
-> task<void>
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto &promise = incoming.get_promise();
  auto to_cout = print_all(std::cout);
  auto pipeline = to_cout(filter_in(is_odd));
  auto coro = pipeline(std::move(incoming));
  for (int i = 0; i < 10; ++i) {
    promise.yield_value(i);
  }
}
```

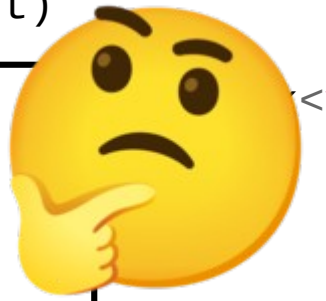
Another level of indirection

But this is still not very nice, is it?

```
auto filter_in = [](auto predicate)
{
   return [predicate]<typename T>(task<T> in) -> task<T>
   {
      for (;;) {
```

```
auto print_all = [](std::ostream& dest)
{
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto &promise = incoming.get_promise();
  auto to_cout = print_all(std::cout);
  auto pipeline = to_cout(filter_in(is_odd));
  auto coro = pipeline(std::move(incoming));
  for (int i = 0; i < 10; ++i) {
    promise.yield_value(i);
  }
}
```



A little helper

```
template <typename T>
struct coro_stage : T {
     coro_stage(T t) : T(std::move(t)) {}
};

template <typename T>
coro_stage(T) -> coro_stage<T>;

template <typename T, typename U>
auto operator|(coro_stage<T> lh, coro_stage<U> rh) {
   return coro_stage{
      [lh = std::move(lh), rh=std::move(rh)]
      <typename T>(task<T> in)
      {
         return rh(lh(std::move(in)));
      }
   };
}
```

A little helper

```
template <typename T>
struct coro_stage : T {
     coro_stage(T t) : T(std::move(t)) {}
};

template <typename T>
coro_stage(T) -> coro_stage<T>;

template <typename T, typename U>
auto operator|(coro_stage<T> lh, coro_stage<U> rh) {
   return coro_stage{
      [lh = std::move(lh), rh=std::move(rh)]
      <typename T>(task<T> in)
      {
         return rh(lh(std::move(in)));
      }
   };
}
```

An indirection wrapper that can be created from any class type.

A little helper

```
template <typename T>
struct coro_stage : T {
     coro_stage(T t) : T(std::move(t)) {}
};

template <typename T>
coro_stage(T) -> coro_stage<T>;

template <typename T, typename U>
auto operator|(coro_stage<T> lh, coro_stage<U> rh) {
   return coro_stage{
      [lh = std::move(lh), rh=std::move(rh)]
      <typename T>(task<T> in)
      {
         return rh(lh(std::move(in)));
      }
   };
}
```

Pipe operators are cool

A little helper

```
template <typename T>
struct coro_stage : T {
     coro_stage(T t) : T(std::move(t)) {}
};

template <typename T>
coro_stage(T) -> coro_stage<T>;

template <typename T, typename U>
auto operator|(coro_stage<T> lh, coro_stage<U> rh) {
   return coro_stage{
      [lh = std::move(lh), rh=std::move(rh)]
      <typename T>(task<T> in)
      {
         return rh(lh(std::move(in)));
      }
   };
}
```

By value

A little helper

```
template <typename T>
struct coro_stage : T {
     coro_stage(T t) : T(std::move(t)) {}
};

template <typename T>
coro_stage(T) -> coro_stage<T>;

template <typename T, typename U>
auto operator|(coro_stage<T> lh, coro_
   return coro_stage{
      [lh = std::move(lh), rh=std::m
      <typename T>(task<T> in)
      {
         return rh(lh(std::move(in)));
      }
   };
}
```

Call the right side with the result from calling the left side

A little helper

```
template <typename T>
struct coro_stage : T {
     coro_stage(T t) : T(std::move(t)) {}
};

template <typename T>
coro_stage(T) -> coro_stage<T>;

template <typename T, typename U>
auto operator|(coro_stage<T> lh, coro_stage<U> rh) {
   return coro_stage{
      [lh = std::move(lh), rh=std::move(rh)]
      <typename T>(task<T> in)
      {
         return rh(lh(std::move(in)));
      }
   };
}
```

Move these, in case they hold move-only data

A little helper

```
template <typename T>
struct coro_stage : T {
     coro_stage(T t) : T(std::move(t)) {}
};

template <typename T>
coro_stage(T) -> coro_stage<T>;

template <typename T, typename U>
auto operator|(coro_stage<T> lh, coro_stage<U> rh) {
   return coro_stage{
      [lh = std::move(lh), rh=std::move(rh)]
      <typename T>(task<T> in)
      {
         return rh(lh(std::move(in)));
      }
   };
}
```

And wrap this new stage for more pipes

Another level of indirection

```
auto filter_in = [](auto predicate)
{
  return coro_stage{
    [predicate]<typename T>(task<T> in) -> task<T>
    {
      for (;;) {
        auto v = co_await in;
        if (predicate(v)) {
          co_yield v;
        }
      }
    }
  };
};
```

Another level of indirection

```
auto filter_in = [](auto predicate)
{
  return coro_stage{
    [predicate]<typename T>(task<T> in) -> task<T>
    {
      for (;;) {
        auto v = co_await i
        if (predicate(v)) {
          co_yield v;
        }
      }
    }
  };
};
```

Use the wrapper

Another level of indirection

```
auto filter_in = [](auto predicate)
{
   return coro_stage{
     [predicate]<type
     {
       for (;;) {
         auto v = co_
         if (predicat
           co_yield v
         }
       }
     }
   };
};
```

```
auto print_all = [](std::ostream& dest)
{
   return coro_stage{
     [&dest]<typename T>(task<T> in) -> task<void>
     {
       for (;;)
       {
         auto v = co_await in;
         dest << v << '\n' << std::flush;
       }
     };
};
```

Use the wrapper

Another level of indirection

```
auto filter_in = [](auto predicate)
{
  return coro_stage{
    [predicate]<type
    {
```

```
auto print_all = [](std::ostream& dest)
{
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto &promise = incoming.get_promise();
  auto pipeline = filter_in(is_odd) | print_all(std::cout);
  auto coro = pipeline(std::move(incoming));
  for (int i = 0; i < 10; ++i) {
    promise.yield_value(i);
  }
}
```

Another level of indirection

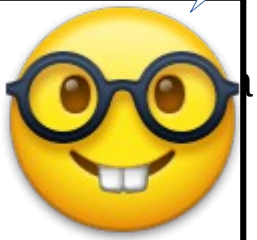
```
auto filter_in = [](auto predicate)
{
  return coro_stage{
    [predicate]<type
    {
```

```
auto print_all = [](std::ostream& dest)
{
```

```
int main()
{
  auto is_odd = [](auto v) { return v & 1;};
  auto incoming = task<int>::make();
  auto &promise = incoming.get_promise();
  auto pipeline = filter_in(is_odd) | print_all(std::cout);
  auto coro = pipeline(std::move(incoming));
  for (int i = 0; i < 10; ++i) {
    promise.yield_value(i);
  }
}
```

```
sk<void>
```

```
;
```



Now we're getting somewhere!

Live Demo!

Feeding the pipeline from io_uring

```
class poller {
public:
  using worker = std::function<void(std::span<char> data)>;
  void add(int fd, worker w) {
    fds_.push_back({fd, POLLIN, 0});
    cbs_.emplace(fd, std::move(w));
  }
  ...
};
```

Feeding the pipeline from io_uring

```
class poller {
public:
  using worker = std::function<void(std::span<char> data)>;
  void add(int fd, worker w) {
    fds_.push_back({fd, POLLIN, 0});
    cbs_.emplace(fd, std::move(w));
  }
  ...
};
```

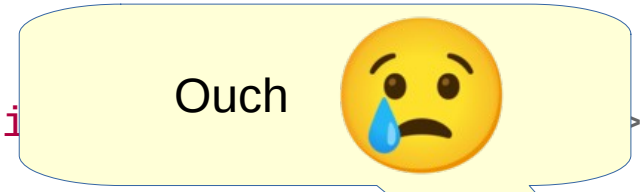
`std::function<F>`
requires F to be copyable

Feeding the pipeline from io_uring

```
class poller {
public:
  using worker = std::function<void(std::span<char> data)>;
  void add(int fd, worker w) {
    fds_.push_back({fd, POLLIN, 0});
    cbs_.emplace(fd, st
  }
  ...
};
```

Feeding the pipeline from io_uring

```
class poller {
public:
  using worker = std::function<voi                    >;
  void add(int fd, worker w) {
    fds_.push_back({fd, POLLIN, 0});
    cbs_.emplace(fd, st
  }
  ...
};
```



cppreference.com

Create account

Page Discussion

C++ Utilities library Function objects std::move_only_functi

std::move_only_function

Defined in header <functional>

```
template< class... >
class move_only_function; // not defined
```

(since C++23)

```
template< class R, class... Args >
class move_only_function<R(Args...)>;
```

Feeding the pipeline from io_uring

```
class poller {
public:
  using worker = std::function<void(std::span<char
  void add(int fd, worker w) {
    fds_.push_back({fd, POLLIN, 0});
    cbs_.emplace(fd, st
  }
  .
};
```

Rolled my own naïve version

```
template <typename>
class move_only_function;

template <typename R, typename ... Args>
class move_only_function<R(Args...)>
{
public:
     move_only_function() = default;
     move_only_function(move_only_function&& r)
     ...
};
```

cppreference.com

Create account

(since C++23)

Feeding the pipeline from io_uring

```
class poller {
public:
  using worker = move_only_function<void(std::span<char> data)>
  void add(int fd, worker w) {
    fds_.push_back({fd, POLLIN, 0});
    cbs_.emplace(fd, std::move(w));
  }
  ...
};
```

```
template <typename R, typename ... Args>
class move_only_function<R(Args...)>
{
public:
     move_only_function() = default;
     move_only_function(move_only_function&& r)
     ...
};
```

Feeding the pipeline from io_uring

How create something that is callable with `span<char>` that feeds into a pipeline via a `task<>`?

Feeding the pipeline from io_uring

```
template <typename task_type, typename pipeline>
auto init_task(pipeline pipe)
{
   auto input = task<task_type>::make();
   auto& promise = input.get_promise();
   using coro_type = decltype(pipe(std::move(input)));
   return [&promise,
           pipe = std::move(pipe),
           input = std::move(input),
           coro = std::unique_ptr<coro_type>{}](task_type value) mutable
   {
      if (!coro) coro = std::make_unique<coro_type>(pipe(std::move(input)));
      promise.yield_value(value);
      return true;
   };
}
```

Feeding the pipeline from i g

Create the task

```
template <typename task_type, typename pipeline>
auto init_task(pipeline pipe)
{
   auto input = task<task_type>::make();
   auto& promise = input.get_promise();
   using coro_type = decltype(pipe(std::move(input)));
   return [&promise,
           pipe = std::move(pipe),
           input = std::move(input),
           coro = std::unique_ptr<coro_type>{}](task_type value) mutable
   {
      if (!coro) coro = std::make_unique<coro_type>(pipe(std::move(input)));
      promise.yield_value(value);
      return true;
   };
}
```

Feeding the pipeline from io_uring

```
template <typename task_type, typename pipeline>
auto init_task(pipeline pipe)
{
   auto input = task<task_type>::make();
   auto& promise = input.get_promise();
   using coro_type = decltype(pipe(std::move(input)));
   return [&promise,
           pipe = std::move(pipe),
           input = std::move(input),
           coro = std::unique_ptr<coro_type>{}](task_type value) mutable
   {
      if (!coro) coro = std::make_unique<coro_type>(pipe(std::move(input)));
      promise.yield_value(value);
      return true;
   };
}
```

Get the promise

Feeding the pipeline from io_uring

```
template <typename task_type, typename pipeline>
auto init_task(pipeline pipe)
{
   auto input = task<task_type>::make();
   auto& promise = input.get_promise();
   using coro_type = decltype(pipe(std::move(input)));
   return [&promise,
           pipe = std::move(pipe),
           input = std::move(input),
           coro = std::unique_ptr<coro_type>{}](task_type value) mutable
   {
      if (!coro) coro = std::make_unique<coro_type>(pipe(std::move(input)));
      promise.yield_value(value);
      return true;
   };
}
```

Get the coroutine type

Feeding the pipeline from io_uring

```
template <typename task_type, typename pipeli
auto init_task(pipeline pipe)
{
   auto input = task<task_type>::make();
   auto& promise = input.get_promise();
   using coro_type = decltype(pipe(std::mov(input)));
   return [&promise,
           pipe = std::move(pipe),
           input = std::move(input),
           coro = std::unique_ptr<coro_type>{}](task_type value) mutable
   {
      if (!coro) coro = std::make_unique<coro_type>(pipe(std::move(input)));
      promise.yield_value(value);
      return true;
   };
}
```

Return something callable that initializes the coroutine return object on the first call.

Feeding the pipeline from io_uring

```
template <typename task_type, typename pipeline>
auto init_task(pipeline pipe)
{
   auto input = task<task_type>::make();
   auto& promise = input.get_promise();
   using coro_type = decltype(pipe(std::move(input)));
   return [&promise,
           pipe = std::move(pipe),
           input = std::move(input),
           coro = std::unique_ptr<coro_type>{}](task_type value) mutable
   {
      if (!coro) coro = std::make_unique<coro_type>(pipe(std::move(input)));
      promise.yield_value(value);
      return true;
   };
}
```

And yields the value to the promise on each call

Feeding the pipeline_uring

Not proud of this code

```
template <typename task_type, typename pipeline>
auto init_task(pipeline pipe)
{
   auto input = task<task_type>::make();
   auto& promise = input.get_promise();
   using coro_type = decltype(pipe(std::move(input)));
   return [&promise,
           pipe = std::move(pipe),
           input = std::move(input),
           coro = std::unique_ptr<coro_type>{}](task_type value) mutable
   {
      if (!coro) coro = std::make_unique<coro_type>(pipe(std::move(input)));
      promise.yield_value(value);
      return true;
   };
}
```

Feeding the pipeline from the uring

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  ring r;
  r.add(port4000.fd(),
        init_task<std::span<char>>(count_packet_data(num_packets, num_bytes)
                                   | to_string
                                   | strip_trailing_newline
                                   | concatenate_to(40)
                                   | print_lines(std::cout)
                                   ));
…
```

Feeding the pipeline fro g

But this is quite nice, isn't it?

```
int main()
{
  uint64_t num_bytes = 0;
  uint64_t num_packets = 0;
  bool done = false;

  ring r;
  r.add(port4000.fd(),
        init_task<std::span<char>>(count_packet_data(num_packets, num_bytes)
                                   | to_string
                                   | strip_trailing_newline
                                   | concatenate_to(40)
                                   | print_lines(std::cout)
                                   ));
…
```

Live Demo!

Takeaways

- Coroutines can be cool and powerful
- Lack of a good library is a major pain
- cancelling both coroutines and operations from io_uring can be tricky
- There are so many ways you can tweak coroutine behaviour, this was but one simplistic example
- We need a good library and we don’t have one
  - and writing one is **really** hard

Takeaways

Takeaways

- Testing and debugging is terrible, especially if you get the future<>/task<>/awaitable<> types wrong

Takeaways

```
template <size_t N>
struct str {
  constexpr str(const char* p) noexcept {
    std::copy_n(p, N, cstr);
  }
  friend std::ostream& operator<<(std::ostream& os, const str& s) {
    return os << s.cstr;
  }
  char cstr[N];
};

template <size_t N>
str(const char (&)[N]) -> str<N>;
```

y if

pes

Takeaways

```
template <size_t N>
struct str {
  constexpr str(const char* p) noexcept {
```

y if

```
template <typename T, str name>
struct promise
{
  task<T,name> get_return_object() noexcept {
    std::cerr << "task<T, " << name << ">::get_return_object()\n";
    return {this};
  }
…
};

template <str name>
task<void, "print_lines"> print_lines(std::ostream& os, task<std::string, name>& in)
{
…
```

Takeaways

```
template <size_t N>
struct str {
  constexpr str(const char* p) noexcept {
```

y if

```
template <typename T, str name>
struct promise
{
  task<T,name> get_return_object() noexcept {
    std::cerr << "task<T, " << name << ">::get_return_object()\n";
    return {this};
  }
…
};

template <str name>
task<void, "print_lines"> print_lines(std::ostream& os, task<std::string, name>& in)
{
…
```

Takeaways

```
template <size_t N>
struct str {
  constexpr str(const char* p) noexcept {
```

y if

```
template <typename T, str name>
struct promise
{
  task<T,name> get_return_object() noexcept {
    std::cerr << "task<T, " << name << ">::get_return_object()\n";
    return {this};
  }
…
};

template <str name>
task<void, "print_lines"> print_lines(std::ostream& os, task<std::string, name>& in)
{
…
```

Takeaways

```
template <size_t N>
struct str {
  constexpr str(const char* p) noexcept {
```

y if

```
template <typename T, str name>
struct promise
{
  task<T,name> get_return_object() noexcept {
    std::cerr << "task<T, " << name << ">::get_return_object()\n";
    return {this};
  }
…
};

template <str name>
task<void, "print_lines"> print_lines(std::ostream& os, task<std::string, name>& in)
{
…
```

Takeaways

- Testing and debugging is terrible, especially if you get the future<>/task<>/awaitable<> types wrong
- Writing asynchronous code as if they were local loops is **very** convenient

Takeaways

- Testing and debugging is terrible, especially if you get the future<>/task<>/awaitable<> types wrong
- Writing asynchronous code as if they were local loops is **very** convenient
- Connecting “pipelines” offers great support for generic utilities

Resources

Shuveb Hussain – “Lord of the io_uring”
https://unixism.net/loti/

Pavel Novikov – “Understanding coroutines by example”, C++London, Feb 2021
https://www.youtube.com/watch?v=7sKUAyWXNHA

Lewis Baker – “CppCoro”
https://github.com/lewissbaker/cppcoro

Facebook experimental – “libunifex”
https://github.com/facebookexperimental/libunifex

Asynchronous I/O and coroutines for smooth data streaming

Björn Fahller

bjorn@fahller.se 

@bjorn_fahller 

@rollbear