Futures from Scratch

A Guided Tour of Concurrency
in C++14 and Beyond
(Concurrency TS, N3865, more)

What’s our goal?

```
Out compute_expensive_sum(In a, In b)
{
    Out oa = expensive_computation(a);
    Out ob = expensive_computation(b);
    return oa + ob;
}
```

What’s our goal?

```
Out compute_expensive_sum(In a, In b)
{
    Out oa;
    std::thread t1([&oa]() {
        oa = expensive_computation(a);
    });
    Out ob = expensive_computation(b);
    t1.join();
    return oa + ob;
}
```

Manual thread management

- Just as bad as manual memory management
- Ugly: Hard to read and $\therefore$ reason about
- Asymmetrical:
  “main thread” versus “other threads”

What’s our goal?

```
Out compute_expensive_sum(In a, In b)
{
    auto oa = async(expensive_computation, a);
    auto ob = async(expensive_computation, b);
    return oa.get() + ob.get();
}
```

Part 1 of this talk will be about how to implement that `async()` thing.

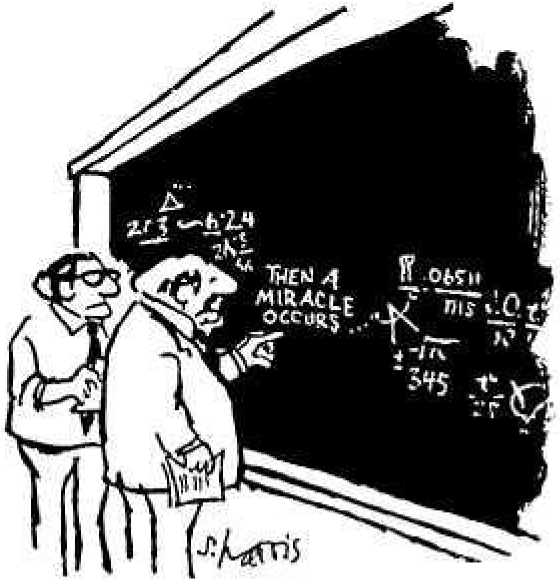
Suppose we have a task scheduler

```
class Scheduler {
    // details

  public:
    Scheduler(int num_worker_threads);

    // fire-and-forget a simple task
    void schedule(std::function<void()> unit);
};
```

Sean Parent's "Better Code: Concurrency" covers the implementation of `Scheduler`, so I won't cover it here.

Suppose we have a task scheduler

```
Scheduler s(10);

Out compute_expensive_sum(In a, In b) {
    Out oa, ob;
    s.schedule([&oa]() {
        oa = expensive_computation(a);
    });
    s.schedule([&ob]() {
        ob = expensive_computation(b);
    });
    // ... ????? ...
    return oa + ob;
}
```



"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

Metaphor time!

This is Pat.

Pat is going to deliver a letter.

This is Frosty.

Frosty is waiting for a letter.

Mailboxes, flags, and cymbals

- Frosty goes to sleep next to the mailbox
- Pat puts a letter in the mailbox
- Pat raises the flag
- Pat clashes her cymbals
- Frosty wakes up, sees the flag raised, and looks in the mailbox

Some minor details

- Frosty is lazy and wants to sleep as much as possible. But he is a light sleeper: we must handle *spurious wakeups*.

- Pat and Frosty can’t both be touching the mailbox (or even looking at it) at the same time. We’ll enforce this with a *mutex*.

The   pattern in code

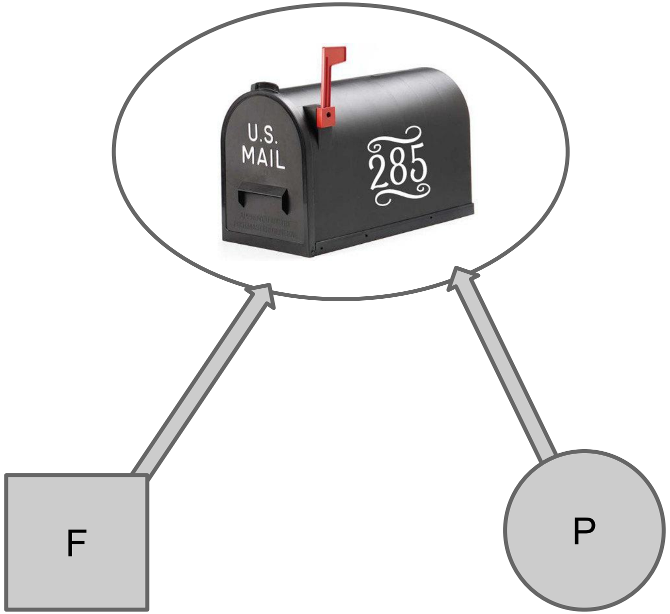

```
Out oa;
std::mutex mtx_a; std::condition_variable cv_a;
bool ready_a = false;

s.schedule([&]() {
    oa = expensive_computation(a); // put a letter in the box
    std::lock_guard<std::mutex> lock(mtx_a);
    ready_a = true;       // raise the flag
    cv_a.notify_one();    // clash the cymbals
});
std::unique_lock<std::mutex> lock(mtx_a);
while (!ready_a) cv_a.wait(lock);  // sleep by the mailbox
// look in the mailbox
```

The problem with mailboxes

The problem with mailboxes

- People move!
- Mailboxes aren't movable (because the mailman and the recipient both need to agree on the mailbox's location)
- We need a movable abstraction
- This is where **futures** come in!

The problem with mailboxes



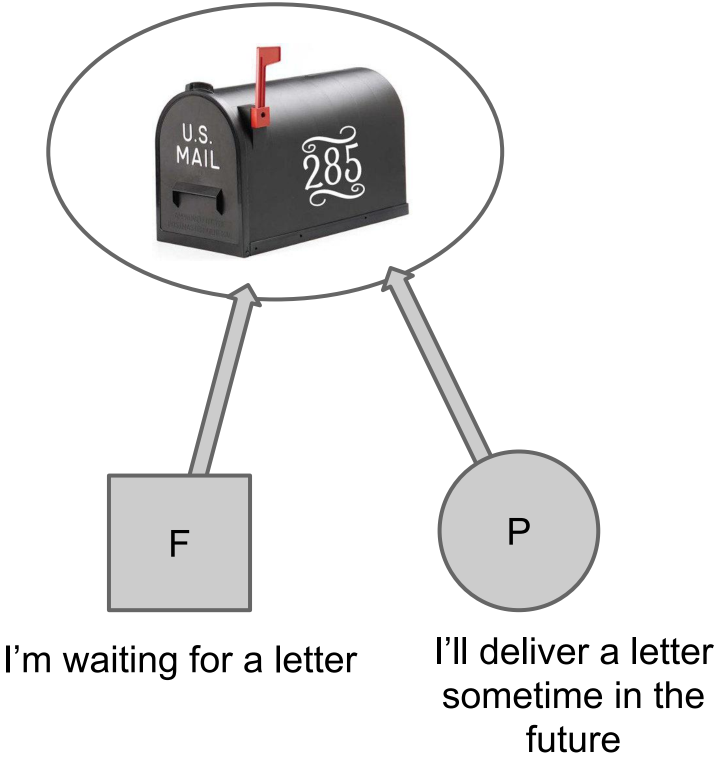
I’m waiting for a letter

I’ll deliver a letter sometime in the future

- We need a movable abstraction
- This is where **futures** come in!
- Mailbox lives on the heap
- Future and Promise each hold a ***pointer*** to the mailbox

These pointers are movable

The problem with mailboxes



- We need a movable abstraction
- This is where **futures** come in!
- Mailbox lives on the heap
- Future and Promise each hold a ***pointer*** to the mailbox

These pointers are movable

Preliminaries: SharedState

```
template<class R> struct Promise;
template<class R> struct Future;

template<class R>
struct SharedState {            // this is our mailbox class
    R value_;
    std::exception_ptr exception_;
    bool ready_ = false;                // the mailbox flag
    std::mutex mtx_;
    std::condition_variable cv_;            // the cymbals
};
```

The purple bit is basically an `expected<R>` (`Folly::Try<R>`); see www.hyc.io/boost/expected-proposal.pdf

Futures: basically this

```
template<class R> struct Promise {
    std::shared_ptr<SharedState<R>> state_;

    void set_value(R value) {
        std::lock_guard<std::mutex> lock(state_->mtx_);
        std::tie(state_->value_, state_->ready_) = std::make_tuple(r, true);
        state_->cv_.notify_all();
    }
};

template<class R> struct Future {
    std::shared_ptr<SharedState<R>> state_;

    R get() {
        std::unique_lock<std::mutex> lock(state_->mtx_);
        while (!state_->ready_) state_->cv_.wait(lock);
        return state_->value_;
    }
};
```

Up next: Not *really* this

Promise (the real deal)

```
template<class R> struct Promise {

    std::shared_ptr<SharedState<R>> state_;
    ...

    void set_value(R r) {
      if (!state_) throw "no_state";
      std::lock_guard<std::mutex> lock(state_->mtx_);
      if (state_->ready_) throw "promise_already_satisfied";
      state_->value_ = std::move(r);
      state_->ready_ = true;
      state_->cv_.notify_all();
    }

    ...
};
```

Promise (the real deal)

```
template<class R> struct Promise {

    std::shared_ptr<SharedState<R>> state_;
    ...

    void set_value(R r) {
      if (!state_) throw "no_state";
      std::lock_guard<std::mutex> lock(state_->mtx_);
      if (state_->ready_) throw "promise_already_satisfied";
      state_->value_ = std::move(r);
      state_->ready_ = true;
      state_->cv_.notify_all();
    }

    ...
};
```

I’m too lazy to write this:

```
throw std::future_error(std::future_errc::no_state);

throw std::future_error(std::future_errc::future_already_retrieved);

throw std::future_error(std::future_errc::promise_already_satisfied);
```

But not too lazy to write this:

```
throw "no_state";
```

http://en.cppreference.com/w/cpp/thread/future_errc

Promise (the real deal)

```
template<class R> struct Promise {

    std::shared_ptr<SharedState<R>> state_;
    ...

    void set_exception(std::exception_ptr p) {
      if (!state_) throw "no_state";
      std::lock_guard<std::mutex> lock(state_->mtx_);
      if (state_->ready_) throw "promise_already_satisfied";
      state_->exception_ = std::move(p);
      state_->ready_ = true;
      state_->cv_.notify_all();
    }

    ...
};
```

Up next: How do we create a future, anyway?

get_future() works only once

```
template<class R> struct Promise {

    std::shared_ptr<SharedState<R>> state_;
    bool future_already_retrieved_ = false;

    void set_value(R r) { ... }
    void set_exception(std::exception_ptr p) { ... }

    Future<R> get_future() {
        if (!state_) throw "no_state";
        if (future_already_retrieved_) throw "future_already_retrieved";
        future_already_retrieved_ = true;
        return Future<R>(state_);
    }
};
```

Up next: The future side of things

Future (the real deal)

```
template<class R> struct Future {

    std::shared_ptr<SharedState<R>> state_;

    void wait() const {
        if (!state_) throw "no_state";
        std::unique_lock<std::mutex> lock(state_->mtx_);
        while (!state_->ready_) state_->cv_.wait(lock);
    }

    R get() {
        wait();
        auto sp = std::move(state_);  // after this line, !this->valid()
        if (sp->exception_) {
            std::rethrow_exception(sp->exception_);
        }
        return std::move(sp->value_);
    }
};
```

Future (the real deal)

```
template<class R> struct Future {

    std::shared_ptr<SharedState<R>> state_;

    void wait() const;  // C++11 also provides wait_for(), wait_until()
    R get();

    bool valid() const { return (state_ != nullptr); }

    bool ready() const {        // N3721 "Improvements to std::future<T> and Related APIs"
        if (!state_) return false;
        std::lock_guard<std::mutex> lock(state_->mtx_);
        return state_->ready_;
    }
};
```

Up next: How do we use this thing, anyway?

Usage example: This works!

```
Scheduler s(10);

Out compute_expensive_sum(In a, In b)
{
    Promise<Out> pa, pb;
    Future<Out> oa = pa.get_future();
    Future<Out> ob = pb.get_future();

    s.schedule([&]() {
        pa.set_value( expensive_computation(a) );
    });
    s.schedule([&]() {
        pb.set_value( expensive_computation(b) );
    });

    return oa.get() + ob.get();
}
```

Up next: Move semantics, right?

Capture each promise “by move”

```
Scheduler s(10);

Out compute_expensive_sum(In a, In b)
{
    Promise<Out> pa, pb;
    Future<Out> oa = pa.get_future();
    Future<Out> ob = pb.get_future();

    s.schedule([pa = std::move(pa)]() mutable {
        pa.set_value( expensive_computation(a) );
    });
    s.schedule([pb = std::move(pb)]() mutable {
        pb.set_value( expensive_computation(b) );
    });

    return oa.get() + ob.get();
}
```

Up next: What if we destroy the promise without calling `set_value`?

But what if this happens?

```
Scheduler s(10);

Out compute_expensive_sum(In a, In b)
{
    Promise<Out> pa, pb;
    Future<Out> oa = pa.get_future();
    Future<Out> ob = pb.get_future();

    s.schedule([pa = std::move(pa)]() mutable {
        if (random()) pa.set_value( expensive_computation(a) );  // uh-oh!
    });
    s.schedule([pb = std::move(pb)]() mutable {
        pb.set_value( expensive_computation(b) );
    });

    return oa.get() + ob.get();
}
```

Broken promises

```
template<class R> struct Promise {

    void set_value(R r);
    void set_exception(std::exception_ptr p);
    Future<R> get_future();

    void abandon_state_() {
        if (!state_ || !future_already_retrieved_) return;
        std::lock_guard<std::mutex> lock(state_->mtx_);
        if (state_->ready_) return;
        state_->exception_ = std::make_exception_ptr("broken_promise");
        state_->ready_ = true;
        state_->cv_.notify_all();
    }

    ~Promise() { abandon_state_(); }
    Promise& operator=(Promise&& rhs) { ... abandon_state_(); ... }
};
```

This completes C++11 future/promise.

Problem: `s.schedule()` is fragile

- Fire-and-forget tasks require onerous bookkeeping
  (if you want to use the results, anyway)
- And it’s fragile: you have to catch your own exceptions.

```
Promise<Out> pa;
Future<Out> oa = pa.get_future();

s.schedule([pa = std::move(pa)]() mutable {
    try {
        pa.set_value( expensive_computation(a) );
    } catch (...) {
        pa.set_exception( std::current_exception() );
    }
});
```

Solution: PackagedTask

Analogous to std::function, but “delayed action”

```
// Template specialization, just like for std::function

template<class Signature> struct PackagedTask;

template<class R, class... A> struct PackagedTask<R(A...)> {

    ...

};
```

Solution: PackagedTask

```
template<class R, class... A> struct PackagedTask<R(A...)> {

    UniqueFunction<void(A...)> task_;        // N4543 "A polymorphic wrapper for..."
    Future<R> future_;
    bool promise_already_satisfied_ = false;

    template<class F> PackagedTask(F&& f) {
      Promise<R> pa;
      future_ = pa.get_future();
      task_ = [pa = std::move(pa), f = std::forward<F>(f)](A... args) mutable {
        try {
          pa.set_value( f(std::forward<A>(args)...) );
        } catch (...) {
          pa.set_exception( std::current_exception() );
        }
      };
    }
    ...
};
```

Solution: PackagedTask

```
template<class R, class... A> struct PackagedTask<R(A...)> {
    // ...
    template<class F> PackagedTask(F&& f) { ... }

    bool valid() const { return task_ != nullptr; }

    Future<R> get_future() {
        if (task_ == nullptr) throw "no_state";
        if (!future_.valid()) throw "future_already_retrieved";
        return std::move(future_);
    }

    void operator()(A... args) {
        if (task_ == nullptr) throw "no_state";
        if (promise_already_satisfied_) throw "promise_already_satisfied";
        promise_already_satisfied_ = true;
        task_(std::forward<A>(args)...);
    }
};
```

Conversions and operator()

```
template<class R, class... A> struct PackagedTask<R(A...)> {
    // ...
    template<class F> PackagedTask(F&& f) { ... }

    bool valid() const { ... }

    Future<R> get_future() { ... }

    void operator()(A... args) { ... }
};
```

```
UniqueFunction<int(char)> intfn = ...;

PackagedTask<int(char)> pt { std::move(intfn) };

UniqueFunction<void(char)> voidfn = std::move(pt);
```

Now, we can use PackagedTask to build a safer abstraction

The safer abstraction is async()

```
inline Scheduler& SystemScheduler() {
    static Scheduler sys(10);
    return sys;
}

template<typename Func, typename R = decltype(std::declval<Func>()())>
Future<R> async(Func func)
{
    PackagedTask<R()> task(std::move(func));
    Future<R> result = task.get_future();
    SystemScheduler().schedule(std::move(task)); // execute it on another
thread
    return result;
}
```

Up next: More than you wanted to know.

What was our goal again?

```
Out compute_expensive_sum(In a, In b)
{
    auto oa = async(expensive_computation, a);
    auto ob = async(expensive_computation, b);
    return oa.get() + ob.get();
}
```

What was our goal again?

```
Out compute_expensive_sum(In a, In b)
{
    auto oa = async(expensive_computation, a);
    auto ob = async(expensive_computation, b);
    return oa.get() + ob.get();
}
```

Technically, we're still missing the ability to pass arguments to `async()`. But we can fix that.

Capturing a pack “by move” is hard

```
template<typename Func, typename... Args>
auto async(Func func, Args... args)
     -> Future<decltype(func(std::move(args)...))>
{
    using R = decltype(func(std::move(args)...));
    PackagedTask<R(Args...)> task(std::move(func));
    Future<R> result = task.get_future();

    auto bound = [
        task = std::move(task),
        args... = std::move(args)...   // this line doesn’t compile
    ]() {
        task(std::move(args)...);
    };
    SystemScheduler().schedule(std::move(bound));
    return result;
}
```

Capturing a pack “by move” is hard

```
template<typename Func, typename... Args>
auto async(Func func, Args... args)
     -> Future<decltype(func(std::move(args)...))>
{
    using R = decltype(func(std::move(args)...));
    PackagedTask<R(Args...)> task(std::move(func));
    Future<R> result = task.get_future();

    auto bound = [
        task = std::move(task),
        argtuple = std::make_tuple(std::move(args)...)
    ]() {
        std::experimental::apply(task, std::move(argtuple));
    };
    SystemScheduler().schedule(std::move(bound));
    return result;
}
```

Capturing a pack “by move” is hard

```
template<typename Func, typename... Args>
auto async(Func func, Args... args)
    -> Future<decltype(func(std::move(args)...))>
{
    using R = decltype(func(std::move(args)...));
    PackagedTask<R(Args...)> task(std::move(func));
    Future<R> result = task.get_future();

    auto bound = std::bind(
        [](auto& task, Args&... args) { task(std::move(args)...); },
        std::move(task),
        std::move(args)...
    );

    SystemScheduler().schedule(std::move(bound));
    return result;
}
```

What was our goal again?

```
Out compute_expensive_sum(In a, In b)
{
    auto oa = async(expensive_computation, a);
    auto ob = async(expensive_computation, b);
    return oa.get() + ob.get();
}
```

Hey, we’ve just implemented that completely from scratch!

Let's make C++ look like Javascript

The technical content in the next part comes from N3784

"Improvements to `std::future<T>` and Related APIs"
(Gustafsson, Laksberg, Sutter, Mithani)

a.k.a. N3721, a.k.a. N3634

Some clever refactoring

```
template<class R> struct Promise {

    std::shared_ptr<SharedState<R>> state_;
    ...

    void set_value(R r) {
        if (!state_) throw "no_state";
        std::lock_guard<std::mutex> lock(state_->mtx_);
        if (state_->ready_) throw "promise_already_satisfied";
        state_->value_ = std::move(r);

        state_->ready_ = true;
        state_->cv_.notify_all();
    }

    ...
};
```

The (unique) `Promise` is the only possible ***writer*** of `state_->ready`, so it itself doesn't need to hold the lock while ***reading*** it.

Some clever refactoring

```
template<class R> struct Promise {

    std::shared_ptr<SharedState<R>> state_;
    ...

    void set_value(R r) {
        if (!state_) throw "no_state";

        if (state_->ready_) throw "promise_already_satisfied";
        state_->value_ = std::move(r);
        std::lock_guard<std::mutex> lock(state_->mtx_);
        state_->ready_ = true;
        state_->cv_.notify_all();
    }

    ...
};
```

The (unique) `Promise` is the only possible ***writer*** of `state_->ready`, so it itself doesn't need to hold the lock while ***reading*** it.

Some clever refactoring

```
template<class R> struct Promise {
    ...

    void set_value(R r) {
        if (!state_) throw "no_state";
        if (state_->ready_) throw "promise_already_satisfied";
        state_->value_ = std::move(r);
        set_ready();
    }

    void set_ready() {
        std::lock_guard<std::mutex> lock(state_->mtx_);
        state_->ready_ = true;
        state_->cv_.notify_all();
    }

    ...
};
```

Next up: Refactor `Promise::set_exception()` in the same way.

Some clever refactoring

```
template<class R> struct Promise {
    ...

    void set_value(R r) {
        if (!state_) throw "no_state";
        if (state_->ready_) throw "promise_already_satisfied";
        state_->value_ = std::move(r);
        set_ready();
    }

    void set_exception(std::exception_ptr p) {
        if (!state_) throw "no_state";
        if (state_->ready_) throw "promise_already_satisfied";
        state_->exception_ = std::move(p);
        set_ready();
    }

    ...
};
```

Next up: Let’s make `set_ready()` more complicated!

The next step: .then()

```
template<class R>
struct SharedState {
    R value_;
    std::exception_ptr exception_;
    std::list<UniqueFunction<void()>> continuations_;
    ...
};

void Promise<R>::set_ready() {
    std::lock_guard<std::mutex> lock(state_->mtx_);
    state_->ready_ = true;
    for (auto& task : state_->continuations_) {
        SystemScheduler().schedule(std::move(task));
    }
    state_->continuations_.clear();
    state_->cv_.notify_all();
}
```

Next up: How to populate continuations_

The next step: .then()

```
template<class F> auto Future<R>::then(F func)
{
    using R2 = decltype(func(move(*this)));
    if (state_ == nullptr) throw "no_state";
    auto sp = state_;  // after this line, !this->valid()
    PackagedTask<R2()> task([func = move(func), arg = move(*this)]() mutable {
        return func(move(arg));
    });
    Future<R2> result = task.get_future();
    std::lock_guard<std::mutex> lock(sp->mtx_);
    if (!sp->ready_) {
        sp->continuations_.emplace_back(move(task));
    } else {
        SystemScheduler().schedule(move(task));  // schedule it immediately
    }
    return result;
}
```

What’s our motivation?

```
Out compute_expensive_sum(In a, In b)
{
    auto oa = async(expensive1, a);
    auto ob = async(expensive1, b);
    return expensive2(oa.get())
         + expensive2(ob.get());
}
```

What’s our motivation?

```
Out compute_expensive_sum(In a, In b)
{
    Future<Mid> ma = async(expensive1, a);
    Future<Out> oa = ma.then([](auto x) {
        return expensive2(x.get());
    });
    // ...
    return oa.get() + ob.get();
}
```

Javascript programmers are cringing at this point...

Let’s make C++ look like ~~Javascript~~ Haskell?

This next part is largely due to Bartosz Milewski’s blog post “C++17: I See a Monad in Your Future!” http://bartoszmilewski.com/2014/02/26/c17-i-see-a-monad-in-your-future/

The technical content comes from N3865 “More improvements to `std::future<T>`” (Vicente J. Botet Escriba)

see also http://cplusplus.github.io/concurrency_ts/

What’s our motivation?

```
Out compute_expensive_sum(In a, In b)
{
    Future<Mid> ma = async(expensive1, a);
    Future<Out> oa = ma.then([](auto x) {
        return expensive2(x.get());
    });
    // ...
    return oa.get() + ob.get();
}
```

Javascript programmers are cringing at this point...

Convenience wrapper .next()

```
Out compute_expensive_sum(In a, In b)
{
    Future<Mid> ma = async(expensive1, a);
    Future<Out> oa = ma.next(expensive2);



    // ...
    return oa.get() + ob.get();
}
```

N3865 “More Improvements to `std::future<T>`”

Convenience wrapper .next()

```
Out compute_expensive_sum(In a, In b)
{
  auto oa = async(exp1, a).next(exp2);
  auto ob = async(exp1, b).next(exp2);
  return oa.get() + ob.get();
}
```

N3865 “More Improvements to `std::future<T>`”

Implementing .next() is easy

```
template<class F>
auto Future<R>::next(F func)
{
    return this->then(

        [func = std::move(func)](Future<R> x) {
            return func(x.get());
        }

    );
}
// As Travis Gockel showed on Tuesday, we can do it a lot
// more efficiently if we care to. I won't show that here.
```

.recover() is just like .next()

```
template<class F>
auto Future<R>::recover(F func) -> Future<R>
{
    return this->then(
        [func = std::move(func)](Future<R> x) {
            try {
                return x.get();
            } catch (...) {
                return func(std::current_exception());
            }
        }
    );
}
```

Now we have Javascript!

```
extern std::string build_string();

Future<double> fd =
    async(
        build_string
    ).next(
        [](auto s){ return std::stod(s); }
    ).recover(
        [](auto){ return std::nan(nullptr); }
    );

double d = fd.get();  // stod or NaN
```

.fallback_to() wraps .recover()

```
extern std::string build_string();

Future<double> fd =
    async(
        build_string
    ).next(
        [](auto s){ return std::stod(s); }
    ).fallback_to(
        std::nan(nullptr)
    );

double d = fd.get();  // stod or NaN
```

.fallback_to() wraps .recover()

```
template<class R>
auto Future<R>::fallback_to(R fallback_value) -> Future<R>
{
    return this->recover(

        [fbv = std::move(fallback_value)](std::exception_ptr) {
            return fbv;
        }

    );
}
```

Time for a quick recap of the interface we’ve built

Our current interface: Promise<R>

```
template<class R> struct Promise {

    Future<R> get_future();
    void set_value(R);
    void set_exception(std::exception_ptr);

  private:
    void set_ready();
    void abandon_state();
};
```

Things we won’t cover in this talk

```
template<class R> struct Promise {

    void set_value_at_thread_exit(R);
    void set_exception_at_thread_exit(std::exception_ptr);
};
```

These exploit the obscure C++11 feature

```
void std::notify_all_at_thread_exit(
    std::condition_variable&,
    std::unique_lock<std::mutex>
);
```

Our current interface: Future<R>

```
template<class R> struct Future {

    R get();
    void wait() const;
    bool valid() const;
    bool ready() const;                              // N3784 (TS)

    template<class F> auto then(F&&);                // N3784 (TS)
    template<class F> auto next(F&&);                // N3865
    template<class F> Future<R> recover(F&&);        // N3865
    Future<R> fallback_to(R);                        // N3865

};
```

Things we won’t cover in this talk

```
template<class R> struct Future {

    void wait_for(std::chrono::duration<...>) const;
    void wait_until(std::chrono::time_point<...>) const;
};
```

These exploit the C++11 features
`std::condition_variable::wait_for()` `std::condition_variable::wait_until()`

Things we won’t cover in this talk

```
template<class R> struct Future {

    SharedFuture<R> share();
};
```

SharedFuture is just like Future, except that it’s copyable.
SharedFuture<R>::get() returns a const R& instead of moving the R.
Neither .get() nor .then() invalidate a SharedFuture.

`when_any()` and `when_all()`

I have implementations of these in the GitHub repo, but I don’t have enough interesting stuff to say about them here to make it worth the time. Go check them out if you care.

Okay, one complaint about the current TS: `when_any()` on a list of zero items returns a ready future. This is weird, mathematically speaking.

“Work-dropping”
`.then()` with cancellation
(not currently proposed)

.then() with cancellation, take 1

```
template<class R, class... A>
template<class F> PackagedTask<R(A...)>::PackagedTask(F&& f)
{
  Promise<R> pa;
  future_ = pa.get_future();
  task_ = [pa = std::move(pa), f = std::forward<F>(f)](A... args) mutable {

      try {
        pa.set_value( f(std::forward<A>(args)...) );
      } catch (...) {
        pa.set_exception( std::current_exception() );
      }

  };
}
```

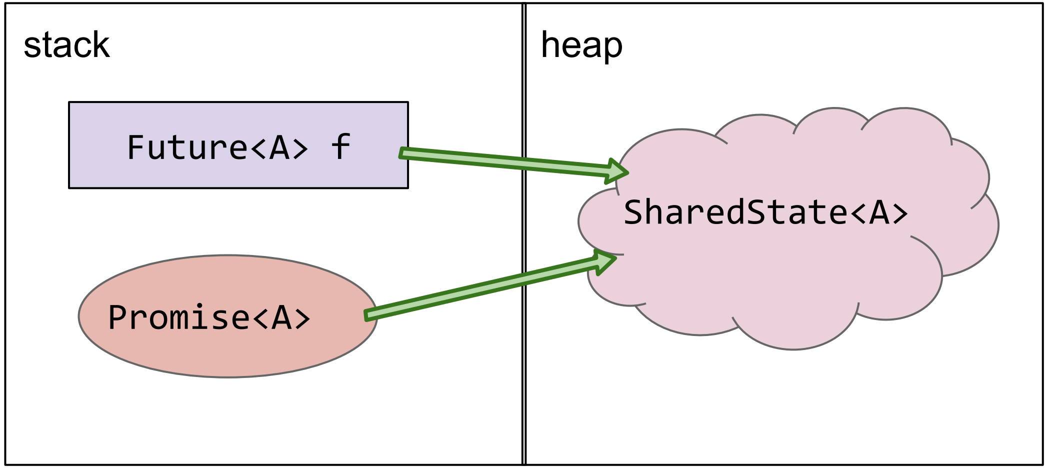
.then() with cancellation, take 1

```
template<class R, class... A>
template<class F> PackagedTask<R(A...)>::PackagedTask(F&& f)
{
  Promise<R> pa;
  future_ = pa.get_future();
  task_ = [pa = std::move(pa), f = std::forward<F>(f)](A... args) mutable {
    if (pa.has_extant_future()) {
      try {
        pa.set_value( f(std::forward<A>(args)...) );
      } catch (...) {
        pa.set_exception( std::current_exception() );
      }
    }
  };
}
```

```
bool Promise<R>::has_extant_future() const {
    return state_ && future_already_retrieved_ && !state_.unique();
}
```

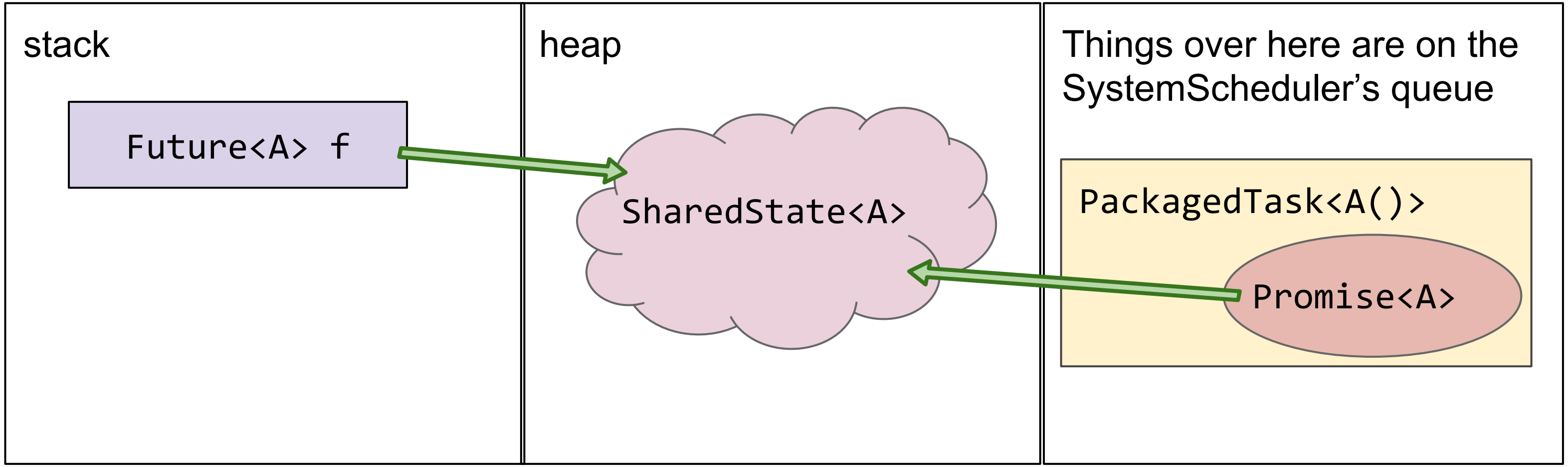
`.then()` with cancellation, take 1

```
Future<A> f = async(some_task);
```



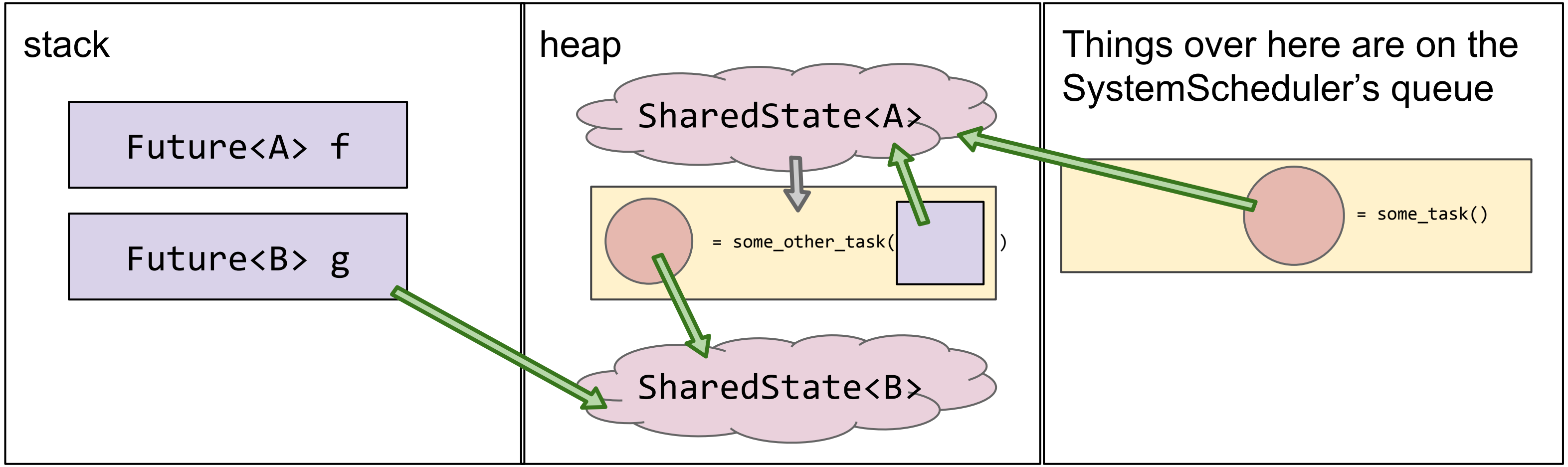
`.then()` with cancellation, take 1

```
Future<A> f = async(some_task);
```



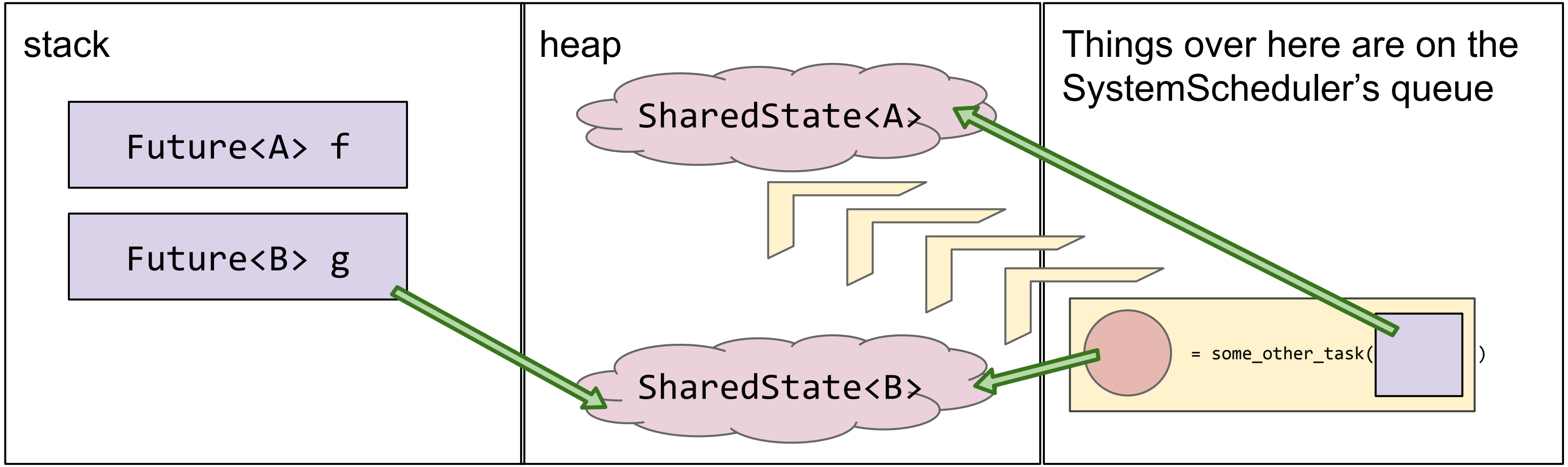
.then() with cancellation, take 1

```
Future<A> f = async(some_task);
Future<B> g = f.next(some_other_task);  // auto some_other_task(A) -> B;
```



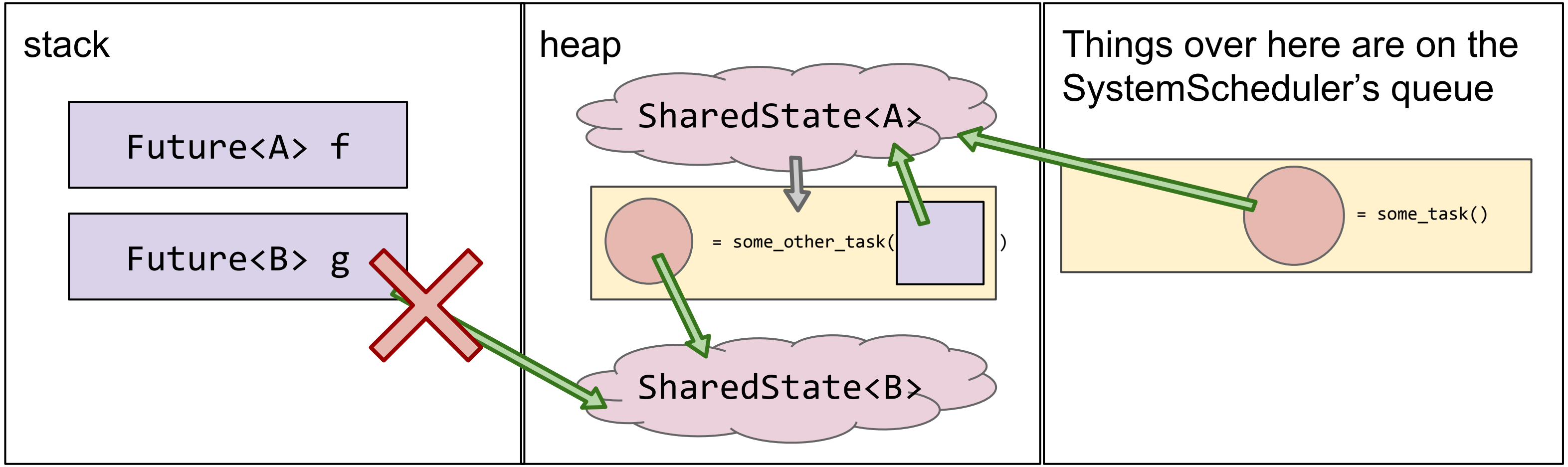
.then() with cancellation, take 1

```
Future<A> f = async(some_task);
Future<B> g = f.next(some_other_task);  // auto some_other_task(A) -> B;
```



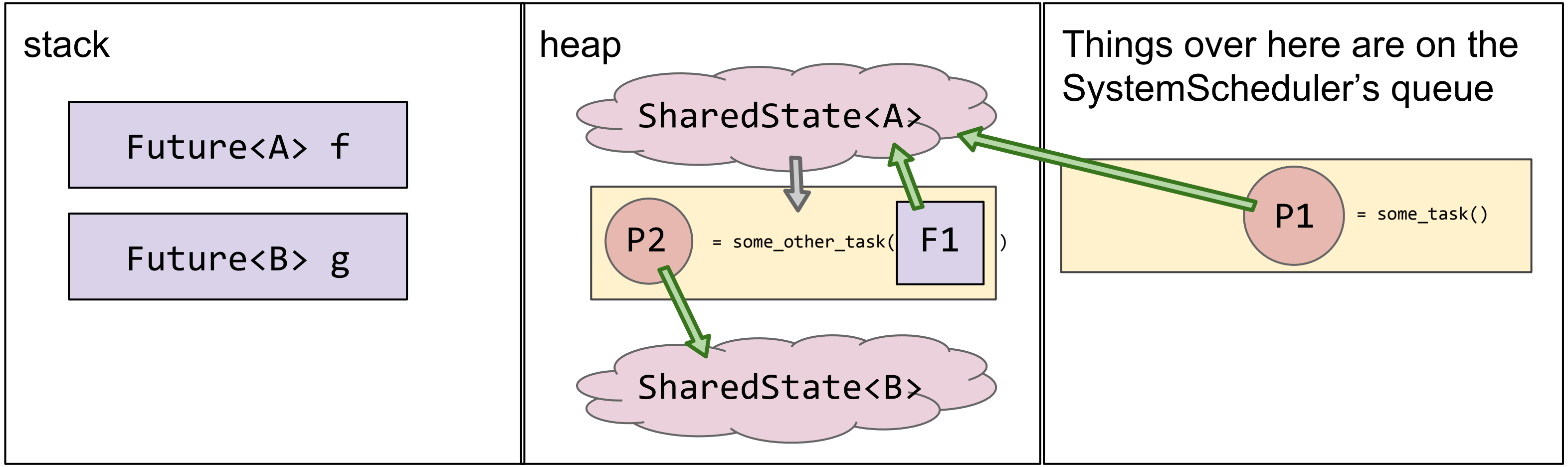
The problem is this...

```
Future<A> f = async(some_task);
Future<B> g = f.next(some_other_task);
g = Future<B>{};  // or g.reset(); — basically, drop it on the floor
```



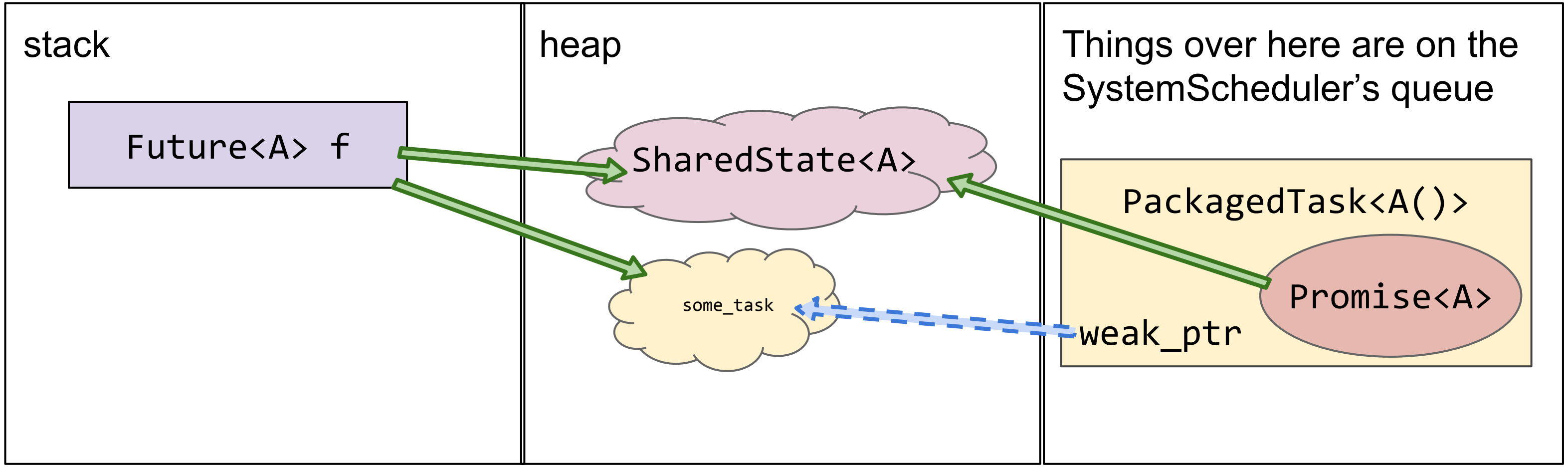
The problem is this...

Without g to keep it relevant, `some_other_task` will never be executed because `P2` has no extant future.
However, since `F1` exists (captured by the continuation attached to `P1->state_`),
`P1` ***does*** have an extant future and `some_task` will still be executed. Oops!



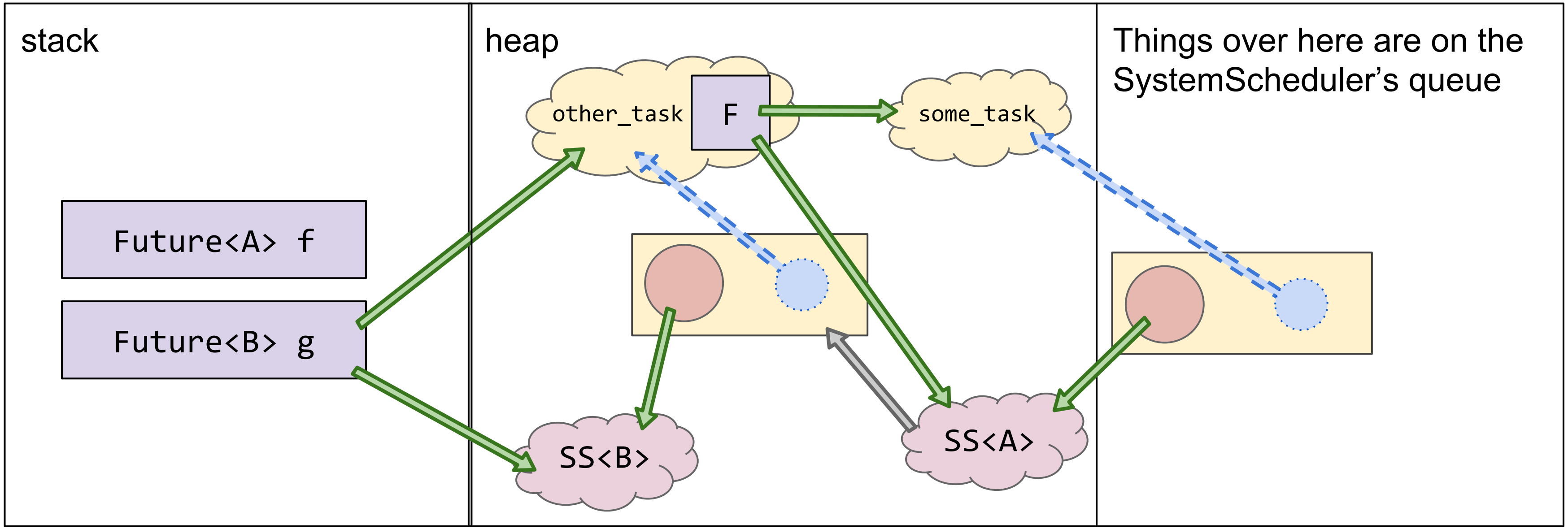
`.then()` with cancellation, take 2

```
Future<A> f = async(some_task);
```



.then() with cancellation, take 2

```
Future<A> f = async(some_task);
Future<B> g = f.next(other_task);  // auto other_task(A) -> B;
```



`.then()` with cancellation, take 2

```
template<class R>
struct Future {

    std::shared_ptr<SharedState<R>> state_;
    std::shared_ptr<void> cancellable_task_state_;

    void attach_cancellable_task_state(std::shared_ptr<void> p) {
        cancellable_task_state_ = std::move(p);
    }
};
```

.then() with cancellation, take 2

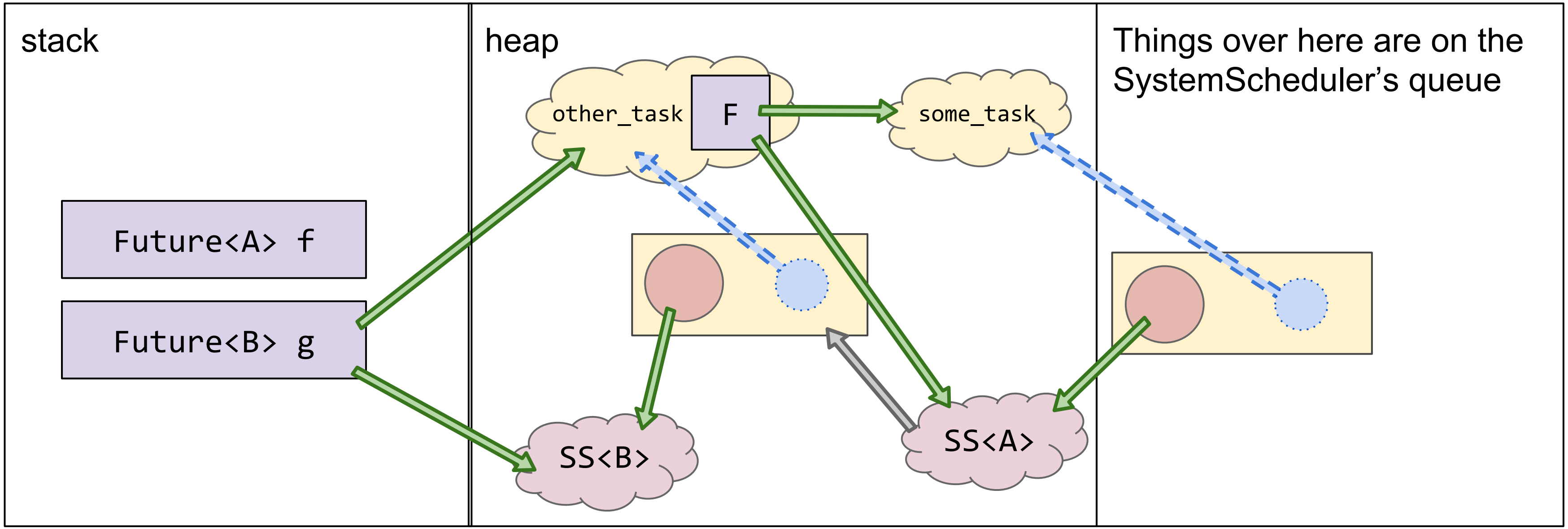
```
template<class F> PackagedTask<R(A...)>::PackagedTask(F&& f)
{


  Promise<R> pa; future_ = pa.get_future();

  task_ = [pa = std::move(pa), f = std::forward<F>(f)](A... args) mutable {

      try {
        pa.set_value( f(std::forward<A>(args)...) );
      } catch (...) {
        pa.set_exception( std::current_exception() );
      }

  };
}
```

.then() with cancellation, take 2

```
template<class F> PackagedTask<R(A...)>::PackagedTask(F&& f)
{
  using FF = std::remove_reference_t<decltype(F)>;
  auto sptr = std::make_shared<FF>(std::forward<F>(f));
  Promise<R> pa; future_ = pa.get_future();
  future_.attach_cancellable_task_state(sptr);
  task_ = [pa = std::move(pa), wptr = sptr.unlock()](A... args) mutable {
    if (auto sptr = wptr.lock()) {
      try {
        pa.set_value( (*sptr)(std::forward<A>(args)...) );
      } catch (...) {
        pa.set_exception( std::current_exception() );
      }
    }
  };
}
```

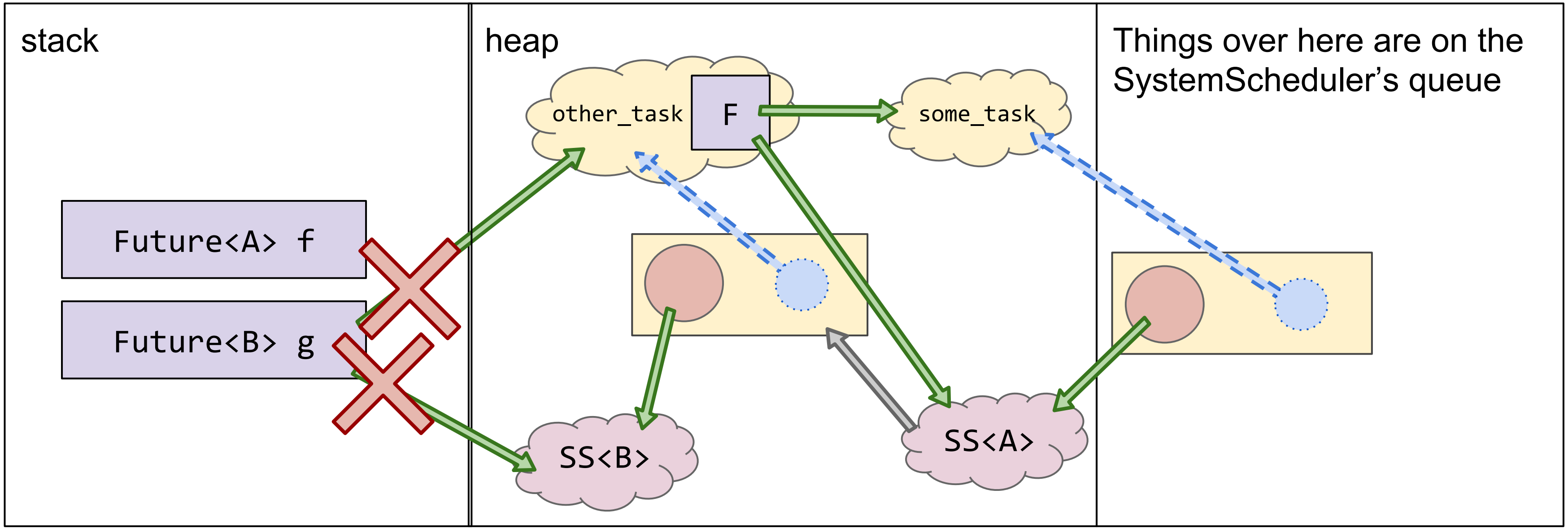
.then() with cancellation, take 2

```
Future<A> f = async(some_task);
Future<B> g = f.next(other_task);  // auto other_task(A) -> B;
```



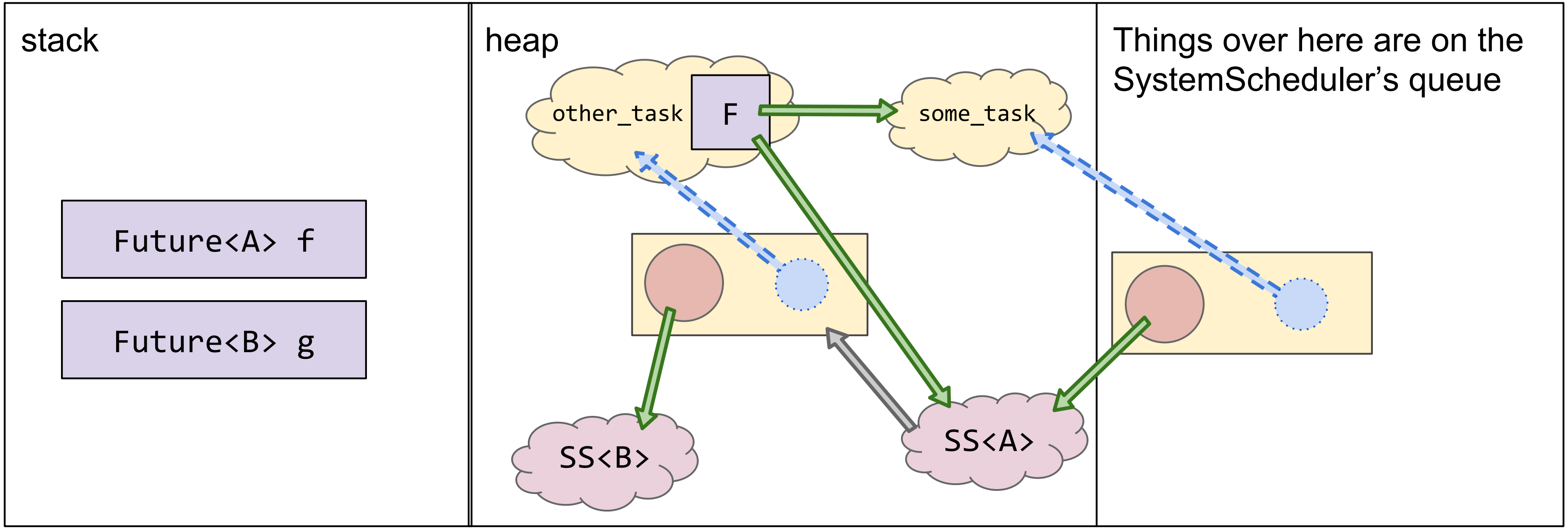
.then() with cancellation, take 2

```
Future<A> f = async(some_task);
Future<B> g = f.next(other_task);  // auto other_task(A) -> B;
g.reset();  // drop it on the floor
```



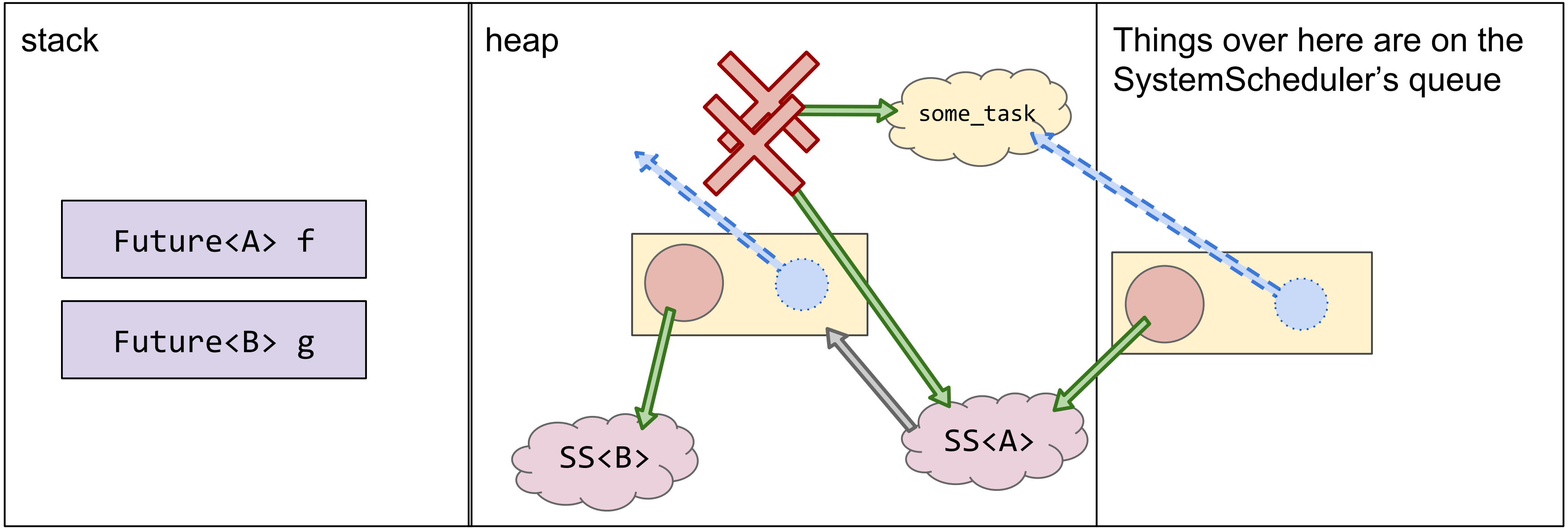
.then() with cancellation, take 2

```
Future<A> f = async(some_task);
Future<B> g = f.next(other_task);  // auto other_task(A) -> B;
g.reset();  // drop it on the floor
```



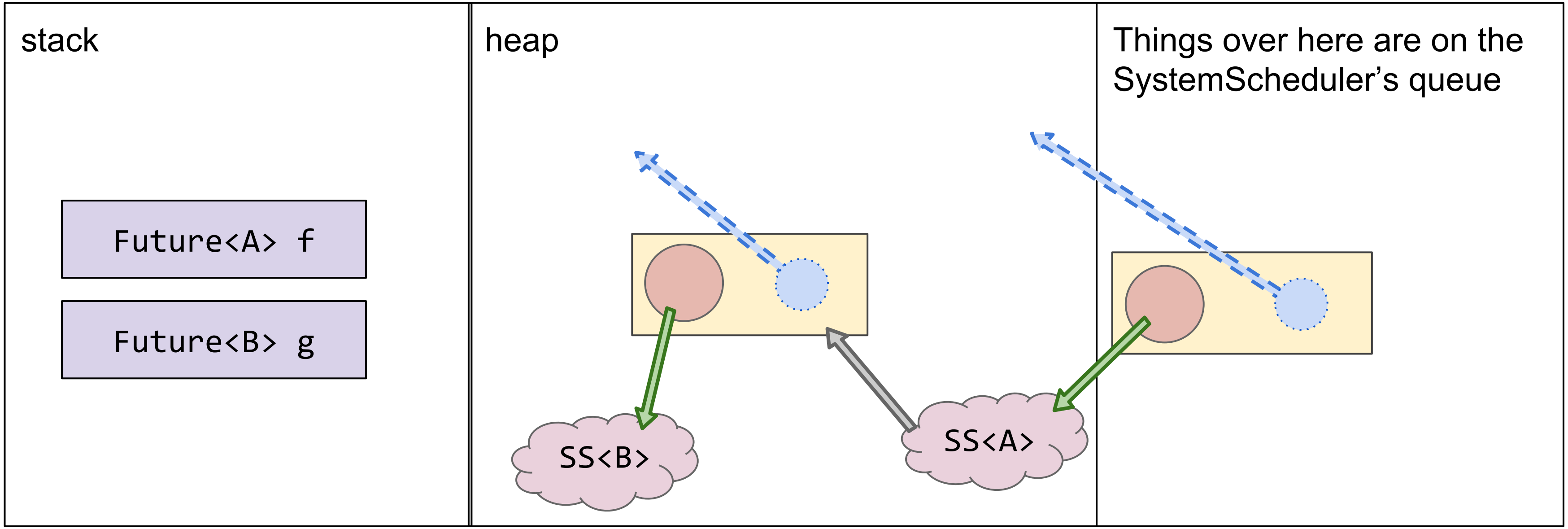
.then() with cancellation, take 2

```
Future<A> f = async(some_task);
Future<B> g = f.next(other_task);  // auto other_task(A) -> B;
g.reset();  // drop it on the floor
```



.then() with cancellation, take 2

```
Future<A> f = async(some_task);
Future<B> g = f.next(other_task);  // auto other_task(A) -> B;
g.reset();  // drop it on the floor
```




SO YOU HAVE QUESTIONS
HOW NICE
memegenerator.net