libprocess

a concurrent and asynchronous programming library

Benjamin Hindman - ACCU Bay Area - August 1st, 2018

libprocess

- Started as an actor library as part of UC Berkeley PhD research
- Created circa 2009, before `std::future`, `std::expected`, and others (some warts of the library are because it started before C++11, but we'd welcome contributions to clean those up)
- Cross-pollination while at Twitter from work on Finagle and Scala
- Lots of contributions from Mesos committers, lives inside of Mesos repository

Main Constructs of libprocess

1. Promises/Futures which are used to build …
2. HTTP abstractions, which make the foundation for …
3. Processes (aka Actors)

Processes (aka actors) are the probably most foreign of the concepts, but they are arguably the most critical part of the library (afterall the library is named after them); we organized this talk to walk through Promises/Futures, then HTTP, before Processes because the former two are prerequisites for the latter.

Part I: Promises/Futures

`Promise<T>` and `Future<T>`

A `Promise` is not copyable, but can be moved (at least in theory).

```
0: Promise<string> promise;
```

You get a `Future` from a `Promise`, or can construct a ready `Future` from a value, or an abandoned `Future` from the default constructor.

```
1: Future<string> future = promise.future();
2: Future<string> ready_future("hello world");
3: Future<string> abandoned_future;
```

A `Future` is copyable and assignable.

```
4: Future<string> future1 = future;
5: abandoned_future = future1; // Now references an unabandoned future.
```

`Promise<T>` and `Future<T>` States

A promise/future starts in the `PENDING` state and can then *transition* to any of the `READY`, `FAILED`, or `DISCARDED` states.

We typically refer to transitioning to `READY` as *completing the promise/future*.

Transition	`Promise::*()`	`Future::is*()`	`Future::on*()`
`READY`	`Promise::set(T)`	`Future::isReady()`	`Future::onReady(F&&)`
`FAILED`	`Promise::fail(const std::string&)`	`Future::isFailed()`	`Future::onFailed(F&&)`
`DISCARDED`	`Promise::discard()`	`Future::isDiscarded()`	`Future::onDiscarded(F&&)`

`Promise<T>` and `Future<T>` Example

```
0: Promise<int> promise;
1: Future<int> future = promise.future();
2: future.onReady([](int i) { cout << i << endl; });
3: promise.set(42); // Prints “42”.

---------------------------------

0: Promise<int> promise;
1: Future<int> future = promise.future();
2: future.onFailed([](const string& message) { cout << message << endl; });
3: promise.fail(“some failure”); // Prints “some failure”.
```

`onReady()`, `onFailed()`, etc, is not a composable pattern, more on that later, but first …
what about `onDiscarded()` ?

Standard Consideration

Discarding a Promise/Future (aka Cancellation)

You can ***try*** and cancel some asynchronous operation by discarding a `Future`.

```
0: Promise<string> promise;
1: Future<string> future = promise.future();
2: future.discard();
3: assert(future.hasDiscard()); // Not this is not isDiscarded()!
4: assert(future.isPending());
```

Doing a discard on a `Future` merely *signals* to the owner of the `Promise` that you no longer care about the result. You can watch for the signal using `Future::onDiscard()`.

```
5: future.onDiscard([&]() {
6:   // Try and cancel the asynchronous operation!
7:   promise.discard();
8: });
9: assert(future.isDiscarded());
```

Standard Consideration

Promise/Future Abandonment

It’s useful to also know when a Promise/Future will *never transition*. We call this abandonment.

```
0: Future<string> abandoned = Future<string>(); // Always pending!
1: assert(abandoned.isAbandoned());

2: abandoned = Promise<string>().future(); // Always pending!
3: assert(abandoned.isAbandoned());

4: auto promise = std::make_unique<Promise<string>>();
5: abandoned = promise->future();
6: assert(!abandoned.isAbandoned());
7: promise.reset();
8: assert(abandoned.isAbandoned());
```

The concept of abandonment was added late to the library so for backwards compatibility reasons we could not add a new state but instead needed to have it be a sub-state of `PENDING`.

Standard Consideration

Future Composition

Instead of using the `is*()` and `on*()` family of functions, use the composition family of functions instead:

Transition	`Future::*()`
`READY`	`Future::then(F&&)`
`FAILED`	`Future::repair(F&&)` and `Future::recover(F&&)`
`DISCARDED`	`Future::recover(F&&)`
Abandoned (`PENDING` and `Future::isAbandoned()`)	`Future::recover(F&&)`

Future<T> Composition Example

```
0: // Returns an instance of `Person` for the specified `name`.
1: Future<Person> find(const std::string& name);
2:
3: // Returns a parent (an instance of `Person`) of the specified `name`.
4: Future<Person> parent(const std::string& name)
5: {
6:   return find(name)
7:     .then([](const Person& person) {
8:       // Try to find the mother and if that fails try the father!
9:       return find(person.mother)
A:         .recover([=](const Future<Person>&) {
B:           return find(person.father);
C:         });
D:     });
E: }
```

Future<T> Composition Example

```
0: // Returns an instance of `Person` for the specified `name`.
1: Future<Person> find(const std::string& name);
2:
3: // Returns a parent (an instance of `Person`) of the specified `name`.
4: Future<Person> parent(const std::string& name)
5: {
6:   return find(name)
7:     .then([](const Person& person) {
8:       // Try to find the mother and if that fails try the father!
9:       return find(person.mother)
A:         .recover([=](const Future<Person>&) {
B:           return find(person.father);
C:         });
D:     });
E: }
```

More on callback semantics later!

Standard Consideration

Composition Complication #1

Infinite loops are easy to write but can cause a stack overflow.

```
0: Future<string> recv_all(Socket socket)
1: {
3:   auto helper = [socket](string&& buffer) {
4:     return socket.recv()
5:       .then([socket, buffer = std::move(buffer)](const string& data) mutable {
6:         if (data.empty()) { // EOF.
7:           return buffer;
8:         }
9:         buffer.append(data);
A:         return helper(std::move(buffer));
B:       };
C:   };
D:   return helper(std::string());
E: }
```

Even with tail call optimizations will leak memory (due to chain of futures, which is non-trivial to optimize).

Standard Consideration

`loop()`

Provides asynchronous looping semantics via an explicit construct.

```
0: Future<string> future = loop(
1:     [](...) {
2:       return iterate();
3:     },
4:     [](...) {
5:       return body();
6:     });
```

Both `iterate()` and `body()` can return a `Future`.

Standard Consideration

`loop()` Example

```
0: Future<string> recv_all(Socket socket)
1: {
2:   string buffer;
3:   return loop(
4:       [socket]() {
5:         return socket.recv();
6:       },
7:       [buffer = std::move(buffer)](const string& data) mutable {
8:         if (data.empty()) { // EOF.
9:           return Break(std::move(buffer));
A:         }
B:         buffer.append(data);
C:         return Continue();
D:       });
E: }
```

Standard Consideration

`loop()` Example

```
0: Future<string> recv_all(Socket socket)
1: {
2:   string buffer;
3:   return loop(
4:       [socket]() {
5:         return socket.recv();
6:       },
7:       [buffer = std::move(buffer)](const string& data) mutable {
8:         if (data.empty()) { // EOF.
9:           return Break(std::move(buffer));
A:         }
B:         buffer.append(data);
C:         return Continue();
D:       });
E: }
```

Standard Consideration

Composition Complication #2

Discarding needs to propagate *through* the composition.

```
0: Promise<int> promise;
1: Future<string> future = promise.future()
2:  .then([](int i) {
3:    return stringify(i);
4:  });
5:
6: future.discard();
7:
8: assert(future.hasDiscard());
9: assert(promise.future().hasDiscard());
```

Standard Consideration

Composition Complication #2

Discarding propagates *through* the composition.

```
0: Promise<int> promise;
1: Future<string> future = promise.future()
2:  .then([](int i) {
3:    return stringify(i);
4:  });
5:
6: future.discard();
7:
8: assert(future.hasDiscard());
9: assert(promise.future().hasDiscard());
```

Discarding propagates through `loop()` as well!

Standard Consideration

Composition Complication #2

Problem: how does the lambda used in the composition get the discard signal!?

```
0: Promise<int> promise;
1: Future<string> future = promise.future()
2:  .then([](int i) {
3:    return expensive(i);                // VERY EXPENSIVE SYNCHRONOUS FUNCTION!
4:  });
5:
6: future.discard();
```

Standard Consideration

Composition Complication #2

Problem: how does the lambda used in the composition get the discard signal!?

```
0: Promise<int> promise;
1: Future<string> future = promise.future()
2:  .then([](int i) {
      if (<???>.isDiscarded()) {
        Promise<string> p;
        p.discard();
        return p.future();
      }
3:    return expensive(i);                  // VERY EXPENSIVE SYNCHRONOUS FUNCTION!
4:  });
5:
6: future.discard();
```

Standard Consideration

Composition Complication #2

Solution: `Future::then()` does a *preemptive* discard.

```
0: Promise<int> promise;
1: Future<string> future = promise.future()
2:  .then([](int i) {
3:    return expensive(i);                // VERY EXPENSIVE SYNCHRONOUS FUNCTION!
4:  });
5:
6: future.discard();
7:
8: promise.set(42);
9:
A: assert(future.isDiscarded());          // EVEN THOUGH THE PROMISE COMPLETED!
```

Standard Consideration

Composition Complication #2

Current semantics are *not intuitive* for most people, what we wish we did was introduce a `discardable()` helper that folks could have used instead.

```
0: Promise<int> promise;
1: Future<string> future = promise.future()
2:  .then(discardable([](int i) {
3:    return expensive(i);                // VERY EXPENSIVE SYNCHRONOUS FUNCTION!
4:  }));
5:
6: future.discard();
7:
8: promise.set(42);
9:
A: assert(future.isDiscarded());          // Because of the `discardable()` helper.
```

Standard Consideration

Composition Complication #3

Problem: sometimes you don't want the discard to propagate!

```
0: Future<string> foo(int i)
1: {
2:   return initialize()                          // WE DON'T WANT TO PROPAGATE THE DISCARD!
3:     .then([i]() {
4:       return bar(i);
5:     });
6: }
7:
8: Future<string> future = foo();
9: future.discard();
```

Standard Consideration

Composition Complication #3

Solution: introduced an `undiscardable()` helper.

```
0: Future<string> foo(int i)
1: {
2:   return undiscardable(initialize())          // Always complete `initialize()`.
3:     .then([i]() {
4:       return bar(i);
5:     });
6: }
7:
8: Future<string> future = foo();
9: future.discard();
```

Composition Complication #3

Solution: introduced an `undiscardable()` helper.

```
0: Future<string> foo(int i)
1: {
2:   return undiscardable(initialize())          // Always complete `initialize()`.
3:     .then([i]() {
4:       return bar(i);
5:     });
6: }
7:
8: Future<string> future = foo();
9: future.discard();
```

Also have an `undiscardable(F&&)` variant that takes some callable `F`.

Standard Consideration

Callback Semantics

There are two possible ways in which the callbacks to the `Future::onReady()` family of functions as well as the composition functions like `Future::then()` get invoked:

1. By the caller of `Future::onReady()`, `Future::then()`, etc *if the future is no longer pending.*

2. By the caller of `Promise::set()`, `Promise::fail()`, etc *if callbacks have been set up while the future was still pending.*

Standard Consideration

Callback Semantics

1. By the caller of `Future::onReady()`, `Future::then()`, etc *if the future is no longer pending.*

```
0: Promise<string> promise;
1: Future<string> future = promise.future();
2:
3: promise.set(“hello world”);
4:
5: future
6:   .then([](const string& s) {        // Callback executed here!
7:     cout << s << endl;
8:   });
```

Standard Consideration

Callback Semantics

2. By the caller of `Promise::set()`, `Promise::fail()`, etc *if callbacks have been set up while the future was still pending.*

```
0: Promise<string> promise;
1: Future<string> future = promise.future();
2:
3: future
4:   .then([](const string& s) {
5:     cout << s << endl;
6:   });
7:
8: promise.set(“hello world”);          // Callback executed here!
```

Standard Consideration

Callback Semantics

There are two possible ways in which the callbacks to the `Future::onReady()` family of functions as well as the composition functions like `Future::then()` get invoked:

1. By the caller of `Future::onReady()`, `Future::then()`, etc *if the future is no longer pending.*

2. By the caller of `Promise::set()`, `Promise::fail()`, etc *if callbacks have been set up while the future was still pending.*

We call these callback semantics *synchronous* because the callbacks will be executed synchronously with respect to the caller of the functions in either 1. or 2. above.

This means that it is critical to consider the synchronization that might be necessary for your code given that multiple possible callers could execute your callbacks! It also means that best practices are that callbacks DO NOT BLOCK, otherwise you may cause some code to block completely unexpectedly (which is a debug nightmare).

Using `defer()` for Asynchronous Callbacks

We introduced `defer()` for asynchronously executing a callback on a pool of threads.

```
0: Promise<int> promise;
1: Future<string> future = promise.future()
2:  .then(discardable(defer([](int i) {
3:    return expensive(i);                // VERY EXPENSIVE SYNCHRONOUS FUNCTION!
4:  }))));
```

Unlike `std::async()`, you can easily compose `defer()` with other callables (it's more like a callable decorator).

Standard Consideration

Using `defer()` for Asynchronous Callbacks

We introduced `defer()` for asynchronously executing a callback on a pool of threads.

```
0: Promise<int> promise;
1: Future<string> future = promise.future()
2:  .then(discardable(defer([](int i) {
3:    return expensive(i);                // VERY EXPENSIVE SYNCHRONOUS FUNCTION!
4:  }))));
```

Unlike `std::async()`, you can easily compose `defer()` with other callables (it's more like a callable decorator).

More on defer later!

collect(), await()

```
0: list<Future<T>> futures = ...;
1:
2: collect(futures)
3:   .then([] (const list<T>& ts) {
4:     // Use each `T` that has been collected.
5:    });


6: await(futures)
7:   .then([] (const list<Future<T>>& futures) {
8:     // Check which state each future is in (none of them will be pending).
9:    });
```

Future::after()

```
0: list<Future<T>> futures = ...;
1:
2: collect(futures)
3:   .then([] (const list<T>& ts) {
4:     // Use each `T` that has been collected.
5:    });
```

Future::after()

```
0: list<Future<T>> futures = ...;
1:
2: collect(futures)
     .after(Milliseconds(100), defer(self(), [] (Future<list<T>> future) {
       future.discard();
       // Handle timeout.
     }))
3:   .then([] (const list<T>& ts) {
4:     // Use each `T` that has been collected.
5:    });
```

Part II: HTTP

HTTP Endpoints via `http::Route()`***

```
0: auto route = http::Route(
1:     “/foo/bar/baz”,
2:     None(),
3:     [](const http::Request& request) {
4:       return http::OK();
5:     });
```

HTTP Endpoints via `http::Route()`***

```
0: auto route = http::Route(
1:     “/foo/bar/baz”,
2:     None(),
3:     [](const http::Request& request) {
4:       return parse(request)
5:         .then([](T t) {
6:           if (invalid(t)) {
7:             return http::BadRequest(“Failed to parse request”);
8:           }
9:           …;
A:           return http::OK();
B:         });
C:     });
```

HTTP Clients

```
0: Future<http::Response> response = http::connect(network::Address(ip, port))
1:   .then([](http::Connection& connection) {
2:     http::Request request;
3:     request.method = “GET”;
4:     request.url.path = …;
5:     request.headers = …;
6:     return connection.send(request);
7:   });
8:
9: response = http::get(http::URL(...));
A: response = http::post(http::URL(...));
```

Part III: Processes (aka Actors)

Processes

Inspired by Erlang, an “actor” in libprocess is called a “process” (not to be confused by an operating system process).

A process receives ***events*** that it processes one at a time. Because a process is only processing one event at a time there's ***no need for synchronization within the process***.

Process Lifecycle

```
0: class FooProcess : public Process<FooProcess> {};
1:
2: FooProcess process;
3: spawn(process);
4: terminate(process);
5: wait(process);
```

Process Events

Each process has a queue of incoming `Event`'s that it processes one at a time. Other actor implementations often call this queue the "mailbox".

There are 5 different kinds of events that can be enqueued for a process:

- `MessageEvent`: a `Message` has been received.
- `DispatchEvent`: a method on the process has been "dispatched".
- `HttpEvent`: an `http::Request` has been received.
- `ExitedEvent`: another process which has been linked has terminated.
- `TerminateEvent`: the process has been requested to terminate.

DispatchEvent

```
0: class FooProcess : public Process<FooProcess>
1: {
2:   string foo(int i)
3:   {
4:     return stringify(i);
5:   }
6: };
7:
8: FooProcess process;
9: spawn(process);
A:
B: Future<string> future = dispatch(process, &FooProcess::foo, 42);
C:
D: terminate(process);
E: wait(process);
```

HttpEvent

```
0: class FooProcess : public Process<FooProcess>
1: {
2:   void initialize() override
3:   {
4:     route(
5:         “/foo/bar/baz”,
6:         None(),
7:         [](const http::Request& request) {
8:           return http::OK();
9:         });
A:   }
B: };
```

Processes and the Pimpl Pattern

```
0: class FooProcess : public Process<FooProcess> { string foo(int i); };
1:
2: class Foo {
3: public:
4:   Foo() { spawn(process); }
5:   ~Foo() { terminate(process); wait(process); }
6:   Future<string> foo(int i) { return dispatch(process, &FooProcess::foo, i); }
7: private:
8:   FooProcess process;
9: };
A:
B: Foo foo;
C: Future<string> future = foo.foo(42);
```

Processes Provide an Execution Context

Problem: remember the callback semantics and `defer()`? Still tricky to do synchronization!

```
0: auto object = std::make_shared<Object>();
1: auto route = http::Route(
2:     "/foo/bar/baz",
3:     None(),
4:     [=](const http::Request& request) {
5:       return process(request)
6:         .then([=](T t) {
7:           object->foo();                  // Need to synchronize access to `object`.
8:           return http::OK();
9:         });
A:     });
```

Solution #1: Asynchronous Synchronization

```
0: Mutex mutex;
1: auto object = std::make_shared<Object>();
2: auto route = http::Route(
3:     "/foo/bar/baz",
4:     None(),
5:     [=](const http::Request& request) {
6:       return process(request)
7:         .then([=](T t) {
8:           return mutex.lock()
9:             .then([]() {
A:               object->foo();
B:               return http::OK();
C:             })
D:             .onAny([]() {
E:               mutex.unlock();
F:             });
G:         });
H:     });
```

Solution #2: Processes

```
0: class FooProcess : public Process<FooProcess>
1: {
2:   void initialize() override
3:   {
4:     route(
5:         “/foo/bar/baz”,
6:         None(),
7:         [=](const http::Request& request) {
8:           return process(request)
9:             .then(defer(self(), [=](T t) {
A:               object.foo();
B:               return http::OK();
C:             });
D:         });
E:   }
F:   Object object;
G: };
```

Actor Programming Model Review

- Not OOP, more composition than inheritance (if ever inheritance)
- Easy to reason about, basically every actor is a server that handles requests and returns responses
- Makes it easy to do *integration* like tests in unit tests
- Not a panacea, can become a performance bottleneck if you aren’t careful to put too much in a single actor

Part IV: Misc

Owned<T>, Shared<T>

Similar to `std::unique_ptr` and `std::shared_ptr` except `Shared<T>` forces only `const T` uses of T and it provides an `own()` member that enables asynchronous recovery of memory.

```
0: Owned<Matrix> matrix(new Matrix());
1: matrix->put(0, 0, value);
2: Shared<Matrix> shared = matrix.share();
3: // shared->put(0, 0, value);            <--- Does not compile.


4: for (size_t i = 0; i < THREADS; i++) {
5:   std::thread([shared]() {
6:     process(shared->get(..., ...));
7:   });
8: }
9:
A: Future<Owned<string>> owned = shared.own();
```

Clock

Questions, Code Examples ...