MANAGING ASYNCHRONY

An Application Perspective

TYPES OF ASYNCHRONY

DETERMINISTIC ORDER.

(I/O)

NON-DETERMINISTIC ORDER.

(USER EVENTS)

THESE ARE DIFFERENT.

Because both are "events",
we reach for the same tools
when building abstractions.

APPLICATION CODE SHOULD USUALLY NOT BE ASYNC.

In some cases, apps will define their own abstractions for the asynchrony. In a few cases (chat servers), it may be appropriate. In most cases, we can abstract away callbacks from application logic through abstraction.

CONTEXT.

ASYNC CODE.

```
fs.readFile("/etc/passwd",
  function(err, data) {
    if (err) { throw err; }
    console.log(data);
  });
```

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

SCHEDULER.

```
var value;
callbacks = new Map;
callbacks.set(function() { return true; }, program);

while(callbacks.length) {
  callbacks.forEach(function(pollStatus, callback) {
    if (value = pollStatus()) { callback(value); }
  });
  sleep(0.1);
}
```

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

SCHEDULER.

callbacks

SCHEDULER.

callbacks

poll, callback

poll, callback

poll, callback

poll, callback

poll, callback

SCHEDULER.

ASYNC CODE.

```
fs.readFile("/etc/passwd",
  function(err, data) {
    if (err) { throw err; }
    console.log(data);
  });
```

true, program

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

ASYNC CODE.

```
fs.readFile("/etc/passwd",
  function(err, data) {
    if (err) { throw err; }
    console.log(data);
  });
```

fileReady, callback

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

ASYNC CODE.

```
fs.readFile("/etc/passwd",
  function(err, data) {
    if (err) { throw err; }
    console.log(data);
  });
```

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

PRIMITIVE STATUS.

```
$ man select

<snip>
Select() examines the I/O descriptor sets
whose addresses are passed in readfds,
writefds, and errorfds to see if some of
their descriptors are ready for
reading, ...
<snip>
```

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

PRIMITIVE VALUE.

```
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
read(fd, buffer, 100);
```

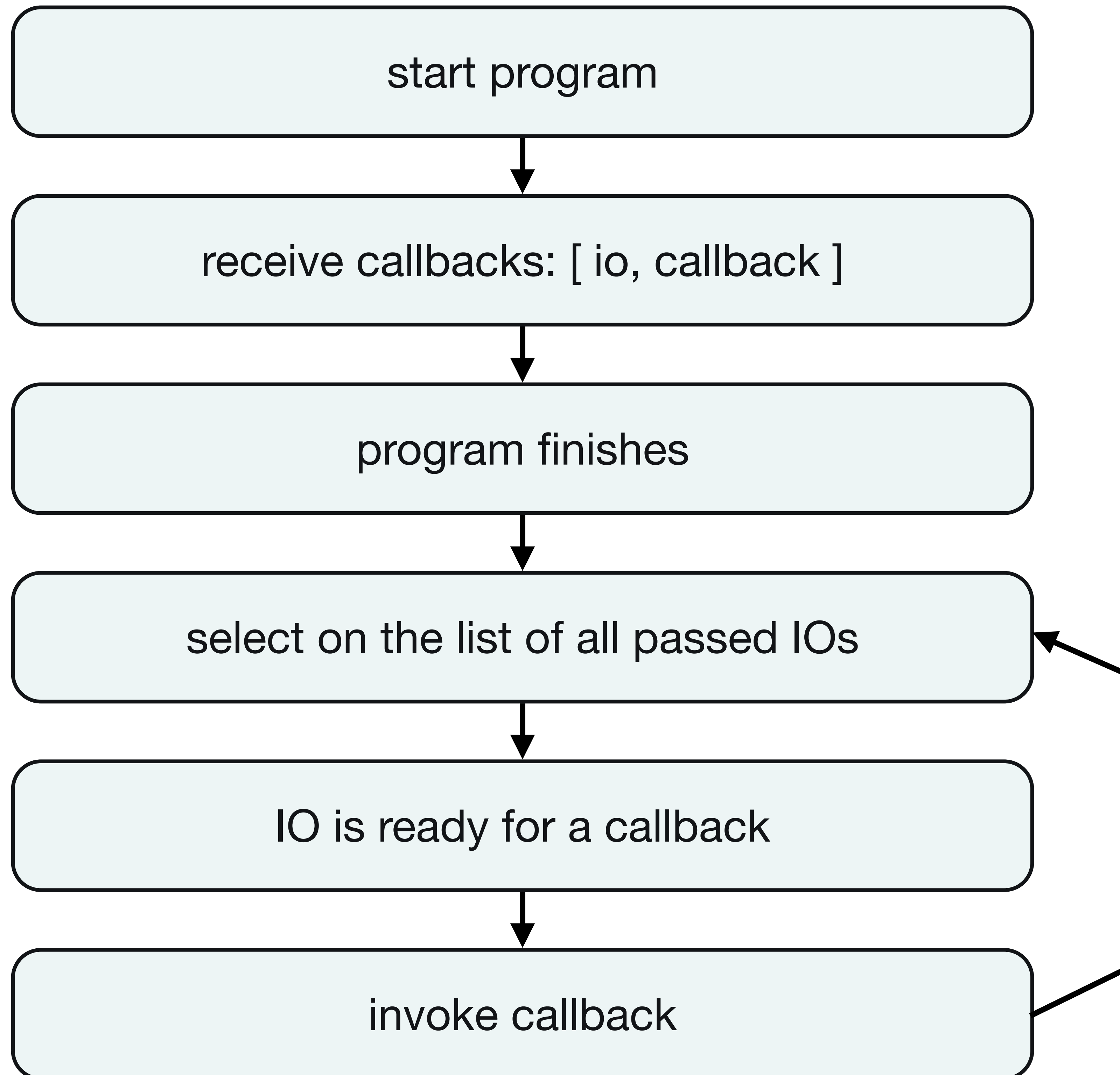
Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

RESULT.

```
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
error = read(fd, buffer, 100);

if (error === -1) {
  callback(errorFrom(errno));
} else {
  callback(null, buffer);
}
```

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

SCHEDULER.



WHEN ORDER IS DETERMINISTIC, WE WANT A SEQUENTIAL ABSTRACTION.

THREADS.

ASYNC.

```
fs.readFile("/etc/passwd",
  function(err, data) {
    console.log(data);
  });

fs.readFile("/etc/sudoers",
  function(err, data) {
    console.log(data);
  });
```

CALLBACK.

function body

THREADS.

```
new Thread(function() {
  var data = fs.readFile("/etc/passwd");
  console.log(data);
});

new Thread(function() {
  var data = fs.readFile("/etc/sudoers");
  console.log(data);
});

sleep();
```

THREADS.

```
new Thread(function() {
  var data = fs.readFile("/etc/passwd");
  console.log(data);
});

new Thread(function() {
  var data = fs.readFile("/etc/sudoers");
  console.log(data);
});

sleep();
```

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

THREADS.

```
new Thread(function() {
  var data = fs.readFile("/etc/passwd");
  console.log(data);
});

new Thread(function() {
  var data = fs.readFile("/etc/sudoers");
  console.log(data);
});

sleep();
```

The scheduler does essentially the same thing when using threads.

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

THREADS.

```
new Thread(function() {
  var data = fs.readFile("/etc/passwd");
  console.log(data);
});

new Thread(function() {
  var data = fs.readFile("/etc/sudoers");
  console.log(data);
});

sleep();
```

Same with initial program.

- Scheduler
- **Initial Program**
- Primitive Status
- Primitive Value
- Error Condition
- Application Callback

STACK.

entry stack frame

stack frame

stack frame

current stack frame

stack frame

local variable values
+ current position

callback

resume thread

The main difference with threads is that the callback structure is more complicated.

SELECT + READ.

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

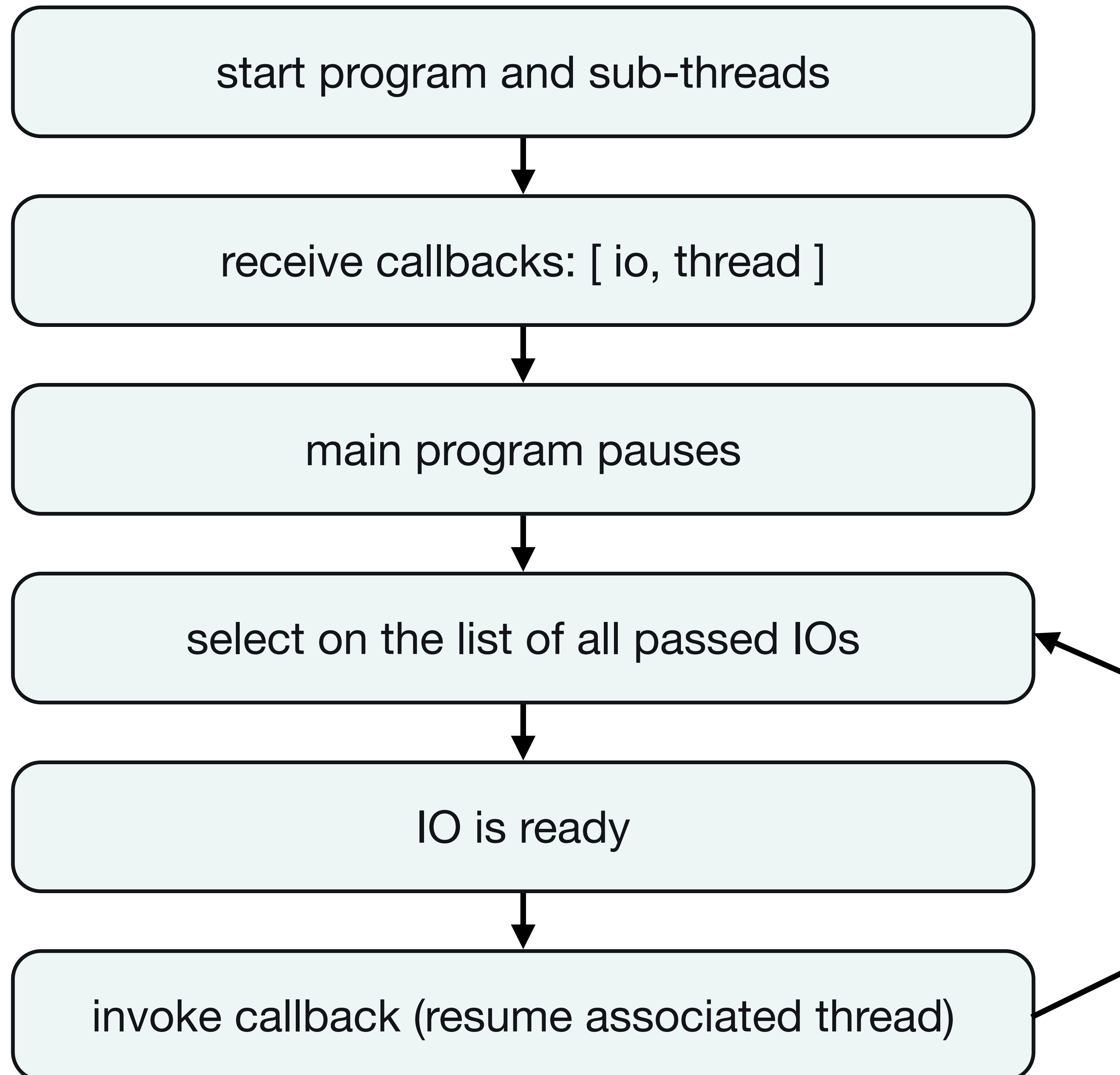
THREADS.

```
new Thread(function() {
  var data = fs.readFile("/etc/passwd");
  console.log(data);
});

new Thread(function() {
  var data = fs.readFile("/etc/sudoers");
  console.log(data);
});

sleep();
```

When data is ready, resume the associated thread.

- Scheduler
- Initial Program
- Primitive Status
- Primitive Value
- Error Condition
- **Application Callback**

SCHEDULER.



DIFFERENCES.

- Possible simultaneous code
  - Can eliminate if desired via a GIL
- Unexpected interleaved code and context switching overhead
  - Can eliminate if desired by disabling pre-emptive scheduling
- More memory required for callback structure

The thread abstraction, which is useful to manage asynchronous events with deterministic order, has varying implementation-defined characteristics. It will always require more memory.

ASYNC.

```
fs.readFile("/etc/passwd",
  function(err, data) {
    console.log(data);
  });

fs.readFile("/etc/sudoers",
  function(err, data) {
    console.log(data);
  });
```

Async code can and usually does still have global state and interleaved execution.

CONFUSION.

"FIBERS ARE LIKE THREADS WITHOUT THE PROBLEMS"

FIBERS.

```
new Fiber(function() {
  var data = fs.readFile("/etc/passwd");
  console.log(data);
});

new Fiber(function() {
  var data = fs.readFile("/etc/sudoers");
  console.log(data);
});

sleep();
```

Same with initial program.

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

IMPLEMENTATION.

```
fs.readFile = function(filename) {
  var fiber = Fiber.current;

  fs.readAsync(filename,
    function(err, data) {
      fiber.resume(data);
    });

  return Fiber.yield();
};
```

Fibers implement the status and value parts of the scheduler in the language, but fundamentally have the same data structures as threads.

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

CALLBACK.

entry stack frame

stack frame

stack frame

current stack frame

stack frame

local variable values
+ current position

callback

resume thread

The fact that fibers are "lighter" is an implementation detail. Having to implement manual yielding is a pain.

> Threads are useful when working with asynchronous events that arrive in a deterministic order.

TO RECAP.

WHEN TO USE CALLBACKS.

APPROACHES.

```
var Stats = Ember.View.extend({
  templateName: 'stats',
  didInsertElement: function() {
    this.$().flot();
  }
});

Stats.create().append();

// vs.

var Stats = Ember.View.extend({
  templateName: 'stats'
});

var view = yield Stats.create().append();
view.$().flot();
```

PROBLEMS.

- Encapsulation: The caller needs to know about the call to `flot()`
- Resiliance to Errors: All callers needs to remember to call `flot()`
- Composability: The framework can no longer simply ask for a view and render it as needed

LIFECYCLE HOOKS SHOULD BE USED TO AID ENCAPSULATION.

JAVASCRIPT.

INVARIANT: TWO ADJACENT STATEMENTS MUST RUN TOGETHER.

This is a core guarantee of the JavaScript programming model and cannot be changed without breaking existing code.

PROBLEM.

```
new Thread(function() {
  var data = fs.readFile("/etc/passwd");
  console.log(data);
});
```

In this case, readFile implicitly halts execution and allows other code to run. This violates guarantees made by JS.

BUT SEQUENTIAL ABSTRACTIONS ARE STILL USEFUL!

"GENERATORS"

YIELDING.

```
var task = function*() {
  var json = yield jQuery.getJSON(url);
  var element = $(template(json)).appendTo('body');

  yield requestAnimationFrame();
  element.fadeIn();
};

var scheduler = new Scheduler;
scheduler.schedule(task);
```

YIELDING.

```
var task = function*() {
  var json = yield jQuery.getJSON(url);
  var element = $(template(json)).appendTo('body');

  yield requestAnimationFrame();
  element.fadeIn();
};

var scheduler = new Scheduler;
scheduler.schedule(task);
```

Here, the task is yielding an object that knows how to be resumed.

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

STATUS AND VALUE.

Thus far, we have limited status and value to built-in primitives that the VM knows how to figure out.

In order for generators to be useful, we will need to expose those concepts to userland.

PROMISES.

PROMISES.

```
file.read = function(filename) {
  var promise = new Promise();

  fs.waitRead(filename, function(err, file) {
    if (err) { promise.error(err); }
    else {
      promise.resolve(file.read());
    }
  });

  return promise;
}
```

Here, we are still allowing the VM to let us know when the file is ready, but we control the callback manually.

- Scheduler
- Initial Program
- Primitive Status
- **Primitive Value**
- Error Condition
- Application Callback

BETTER PROMPT.

```
var prompt = function() {
  var promise = new Promise();

  $("input#confirm")
    .show()
    .one('keypress', function(e) {
      if (e.which === 13) {
        promise.resolve(this.value);
      }
    });

  return promise();
};

spawn(function*() {
  var entry = yield prompt();
  console.log(entry);
});
```

In this case, we are in control of both the status information and the value of the status.

- Scheduler
- Initial Program
- **Primitive Status**
- **Primitive Value**
- Error Condition
- Application Callback

Promises provide a shared implementation of status/value that a scheduler can use. In JavaScript, generators + promises give us a way to use a sequential abstraction for asynchronous events with deterministic order.

In a UI app, a small part of the total interface may have deterministic order, and we can use this abstraction on demand.

PROMISES ARE A PRIMITIVE.

In general, I would prefer to use promises together with a higher level sequential code abstraction than use promises directly in application code.

PSEUDOCODE.

```
var spawn = function(generator, value) {
  try {
    promise = generator.send(value);
    promise.success(function(value) {
      spawn(generator, value);
    })
    promise.fail(function(err) {
      generator.throw(err);
    })
  } catch(e) {
    if (!isStopIteration(e)) {
      generator.throw(err);
    }
  }
};
```

Scheduler
Initial Program
Primitive Status
Primitive Value
Error Condition
Application Callback

GENERATORS ARE EXPLICIT AND SHALLOW.

Because generators are explicit and shallow, they don't have a parent stack to store, so their memory usage is more like callbacks. Generators can be explicitly chained though.

OTHER LANGUAGES?

BETTER PRIMITIVES.

```
class File
  def read
    promise = Promise.new
    async_read do |string|
      promise.resolve(string)
    end
    Thread.yield promise
  end
end
```

In languages that already have threads, promises can be used to provide more power in the existing scheduler. This can be implemented in terms of sync primitives.

EXTERNAL EVENTS.

These are events that are external to the application or application framework.

They are typically handled as asynchronous events. If they have deterministic order, the above techniques may work.

EXTERNAL EVENTS.

However, you may not want to wait to do anything until the first Ajax request is returned, so you typically will not try to use sequential abstractions even for network I/O.

TWO TYPES.

- External events
  - I/O responses (IndexedDB, Ajax)
  - setTimeout and setInterval
  - DOM Mutation Observers
- User Interaction
  - click
  - swipe
  - unload

INTERNAL EVENTS.

These are events generated by the app or app framework for another part of the app or app framework.

EVENTED MODELS.

```
var EventedObject = function(properties) {
  for (var property in properties) {
    this[property] = properties[property];
  }
};

Object.prototype.set = function(key, value) {
  EventEmitter.fire(this,
    key + ':will-change');

  this[key] = value;

  EventEmitter.fire(this,
    key + ':did-change');
};
```

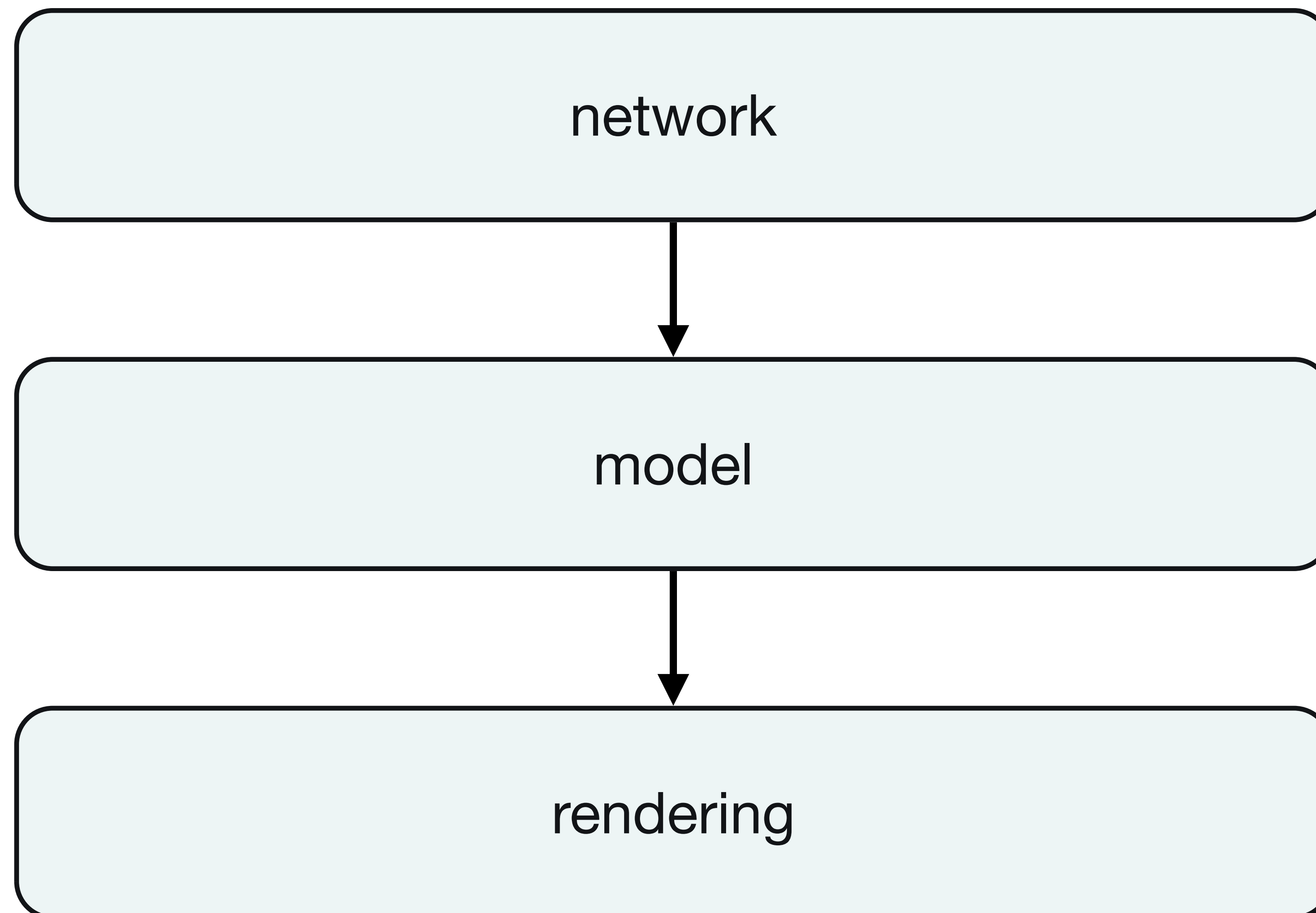
DECOUPLING.

```
var person = new EventedObject({
  firstName: "Yehuda",
  lastName: "Katz"
});

$("#person").html("<p><span>" +
  person.firstName + '</span><span>' +
  person.lastName + "</span></p>");

EventEmitter.on(person, 'firstName:did-change',
  function() {
    $("#person span:first").html(person.firstName);
  });

EventEmitter.on(person, 'lastName:did-change',
  function() {
    $("#person span:last").html(person.lastName);
  });
```

NETWORK.

```
$.getJSON("/person/me", function(json) {
  person.set('firstName', json.firstName);
  person.set('lastName', json.lastName);
});
```

DECOUPLING.



PROBLEMS.

```
var person = new EventedObject({
  firstName: "Yehuda", lastName: "Katz",
  fullName: function() {
    return [this.firstName, this.lastName].join(' ');
  }
});

$("#person").html("<p>" + person.fullName() + "</p>");

EventEmitter.on(person, 'firstName:did-change',
  function() {
    $("#person p").html(person.fullName());
  });

EventEmitter.on(person, 'lastName:did-change',
  function() {
    $("#person p").html(person.fullName());
  });
```

IF WE DO THIS:

```
$.getJSON("/person/me", function(json) {
  person.set('firstName', json.firstName);
  person.set('lastName', json.lastName);
});
```

DOUBLE RENDER!

TRANSACTIONS.

COALESCING.

```
var person = new EventedObject({
  firstName: "Yehuda", lastName: "Katz",
  fullName: function() {
    return [this.firstName, this.lastName].join(' ');
  }
});

EventEmitter.on(person, 'firstName:did-change',
  function() {
    UniqueEmitter.fire(person, 'fullName:did-change');
  });

EventEmitter.on(person, 'lastName:did-change',
  function() {
    UniqueEmitter.fire(person, 'fullName:did-change');
  });

UniqueEmitter.on(person, 'fullName:did-change',
  function() {
    $("#person p").html(person.fullName());
  });
```

NETWORK.

```
$.getJSON("/person/me", function(json) {
  UniqueEmitter.begin();
  person.set('firstName', json.firstName);
  person.set('lastName', json.lastName);
  UniqueEmitter.commit();
});
```

GETS COMPLICATED.

This type of solution is not a very good user-facing abstraction, but it is a good primitive.

DATA FLOW.

What we want is an abstraction that describes the data flow via data bindings.

DECLARATIVE.

```
var person = Ember.Object.extend({
  firstName: "Yehuda",
  lastName: "Katz",

  fullName: function() {
    return [this.get('firstName'),
      this.get('lastName')].join(' ');
  }.property('firstName', 'lastName')
});

$("#person").html("<p>" +
  person.get('fullName') + "</p>");

person.addObserver('fullName', function() {
  $("#person p").html(person.get('fullName'));
});
```

The .property is the way we describe that changes to firstName and lastName affect a single output.

NETWORK.

```
$.getJSON("/person/me", function(json) {
  person.set('firstName', json.firstName);
  person.set('lastName', json.lastName);
});
```

Behind the scenes, Ember defers the propagation of the changes until the turn of the browser's event loop.

Because we know the dependencies of fullName, we can also coalesce the changes to firstName and lastName and only trigger the fullName observer once.

BOUNDARIES.

A good data binding system can provide an abstraction for data flow for objects that implement observability.

For external objects and events, you start with asynchronous observers (either out from the binding system or in from the outside world).

BETTER ABSTRACTIONS.

For common cases, we can do better.

EXTENDED REACH.

```
var person = Ember.Object.create({
  firstName: "Yehuda", lastName: "Katz",

  fullName: function() {
    return [this.get('firstName'),
      this.get('lastName')].join(' ');
  }.property('firstName', 'lastName')
});

var personView = Ember.Object.create({
  person: person,
  fullNameBinding: 'person.fullName',
  template: compile("<p>{{fullName}}</p>")
});

person.append();

$.getJSON("/person/me", function(json) {
  person.set('firstName', json.firstName);
  person.set('lastName', json.lastName);
});
```

For common boundary cases, like DOM, we can wrap the external objects in an API that understands data-binding.

This means that we won't need to write async code to deal with that boundary.

SAME SEMANTICS.

If you extend an async abstraction to an external system, make sure that the abstraction has the same semantics when dealing with the outside world.

In Ember's case, the same coalescing guarantees apply to DOM bindings.

CONCLUSION.

In general, applications should not have large amounts of async code.

In many cases, an application will want to expose abstractions for its own async concerns that allow the bulk of the application code to proceed without worrying about it.

LIMITED ASYNC CODE IN APPLICATIONS.

One common pattern is using an async reactor for open connections but threads for the actual work performed for a particular open connection.

DIFFERENT KINDS OF ASYNC CODE.

Asynchronous code that arrives in a strict deterministic order can make use of different abstractions that async code that arrives in non-deterministic order.

Internal events can make use of different abstractions than external events.

LAYERS.

Promises are a nice abstraction on top of async code, but they're not the end of the story.

Building task.js on top of promises gives us three levels of abstraction for appropriate use as needed.

THANK YOU.

QUESTIONS?