Porting Node
The Journey from Lake JavaScript to The Strait of Lua
We Want to Swap Engines

As of node.js v0.6.x, node is just V8 bindings to libuv right?

So let’s just swap V8 with another engine. LuaJit looks nice.

This should be quick and easy. Let’s port the boat!

Tuesday, July 3, 12
Boating in Lake JavaScript

C Libraries

- **libuv** - Provides non-blocking, callback based I/O and timers.
- **http_parser** - Fast event-based HTTP protocol parser.
- **openssl** - Provides crypto primitives.
- **zlib** - Provides compression and decompression.

Scripting Language Virtual Machine

- **Google V8** - Runs JavaScript code.
Navigating The Strait of Lua

C Libraries

- **libuv** - Provides non-blocking, callback based I/O and timers.
- **http_parser** - Fast event-based HTTP protocol parser.
- **openssl** - Provides crypto primitives.
- **zlib** - Provides compression and decompression.

Scripting Language Virtual Machine

- **LuaJit** - Runs Lua code.
Why Bother Porting the Canoe?

- LuaJit is much lighter than V8 in embedded situations.
- I don’t like C++ for addons, I prefer straight C.
- Lua has coroutines! (an alternative to callbacks)
- LuaJit has really fast FFI built-in
- I wanted to make other things than websites.
Begin the Journey
Learning libUV

- Cloned the repo at https://github.com/joyent/libuv.git
- Read include/uv.h
- Joined #libuv on freenode IRC.
- Work with @piscisaureus and @bnoordhuis.
int main() {
    /* Initialize the tcp server handle */
    uv_tcp_t* server = malloc(sizeof(uv_tcp_t));
    uv_tcp_init(uv_default_loop(), server);

    /* Bind to port 8080 on "0.0.0.0" */
    printf("Binding to port 8080\n");
    if (uv_tcp_bind(server, uv_ip4_addr("0.0.0.0", 8080))) {
        error("bind");
    }

    /* Listen for connections */
    printf("Listening for connections\n");
    if (uv_listen((uv_stream_t*)server, 128, on_connection)) {
        error("listen");
    }

    /* Block in the main loop */
    uv_run(uv_default_loop());
    return 0;
}

/* Callback for new tcp client connections */
static void on_connection(uv_stream_t* server, int status) {
    uv_tcp_t* client;

    if (status) error("on_connection");
    client = malloc(sizeof(uv_tcp_t));
    uv_tcp_init(uv_default_loop(), client);

    /* Accept the client */
    if (uv_accept((uv_stream_t*)server, (uv_stream_t*)client)) {
        error("accept");
    }
    printf("connected\n");

    /* Start reading data from the client */
    uv_read_start((uv_stream_t*)client, on_alloc, on_read);
}
TCP Server Continued

/* Helper for exiting on errors */
static void error(const char* name) {
    uv_err_t err = uv_last_error(uv_default_loop());
    fprintf(stderr, "%s: %s\n", name, uv_strerror(err));
    exit(-1);
}

/* Hook to allocate data for read events */
static uv_buf_t on_alloc(uv_handle_t* handle, size_t suggested_size) {
    return uv_buf_init(malloc(suggested_size), suggested_size);
}

static void on_close(uv_handle_t* handle) {
    free(handle);
    printf("disconnected\n");
}

/* Callback on data chunk from client */
static void on_read(uv_stream_t* stream, ssize_t nread, uv_buf_t buf) {
    if (nread >= 0) {
        printf("chunk: \%.*s", (int)nread, buf.base);
    } else {
        uv_err_t err = uv_last_error(uv_default_loop());
        if (err.code == UV_EOF) {
            printf("eof\n");
            uv_close((uv_handle_t*)stream, on_close);
        } else {
            fprintf(stderr, "read: \%s\n", uv_strerror(err));
            exit(-1);
        }
    }
    free(buf.base);
}
Learning Lua

- Bought *Programming in Lua 2nd Edition*.
- Read it all!
- Joined the lua-l mailing list
- Joined #lua on freenode irc
- Bookmarked the HTML manual [http://www.lua.org/manual/5.1/](http://www.lua.org/manual/5.1/)
Prototype Some APIs

- Can **node-style** APIs work in the **Lua** language?
- **Lua tables** are similar to **JS objects**, but not the same.
  - There is **no inheritance**. But there are **metatables**.
  - **Anything** can be used as **keys**, including other tables or functions.
  - Tables may not contain **nil** for keys or values.
  - There is no **this**, but there is function calling sugar with **self**.
local Object = {}
Object.meta = {__index = Object}

-- Creates a new instance.
function Object:new(...)
    local obj = setmetatable({}, self.meta)
    if obj.initialize then
        obj:initialize(...)
    end
    return obj
end

-- Creates a new sub-class
function Object:extend()
    local sub = setmetatable({}, self.meta)
    sub.meta = {__index = sub}
    return sub
end

-- Create a class using Object
local Rect = Object:extend()
function Rect:initialize(w, h)
    self.w = w
    self.h = h
end

function Rect:getArea()
    return self.w * self.h
end

-- Create an instance it.
local rect = Rect:new(4, 5)
print(rect:getArea())
local Emitter = Object:extend()

-- Register an event listener
function Emitter:on(name, callback)
  -- Lazy create event types table.
  if not self.handlers then
    self.handlers = {}
  end
  local handlers = self.handlers

  -- Lazy create table for callbacks.
  if not handlers[name] then
    handlers[name] = {}
  end

  -- Store the callback as a key
  handlers[name][callback] = true
end

-- Remove an event listener
function Emitter:off(name, callback)
  -- Get the list of callbacks.
  local list = self.handlers and self.handlers[name]
  if not list then return end

  -- Delete the key by setting to nil.
  list[callback] = nil
end

-- Emit a named event.
function Emitter:emit(name, ...
  -- Get the list of callbacks.
  local list = self.handlers and self.handlers[name]
  if not list then return end

  -- Call each one with the args.
  for callback in pairs(list) do
    callback(...)
  end
end
```javascript
-- Load the net module
var net = require("net");

// Create a tcp server
net.createServer(function (client) {
    console.log("connected");

    client.on("data", function (chunk) {
        process.stdout.write(chunk);
    });

    client.on("end", function () {
        console.log("eof");
    });

    client.on("close", function () {
        console.log("disconnected");
    });
}).listen(8080, function () {
    console.log("Listening on port 8080");
});
```

```cpp
-- Load the net module
local net = require "net"

-- Create a tcp server
net.createServer(function (client)
    print "connected"
    client:on("data", function (chunk)
        process.stdout:write(chunk)
        end)
    end)
    client:on("end", function ()
        print "eof"
        end)
    client:on("close", function ()
        print "disconnected"
        end)
end):listen(8080, function ()
    print "Listening on port 8080"
    end)
```
Faux Blocking in Luvit

```lua
fiber.new(function (wrap)
  -- Wrap some functions for sync-style calling
  local sleep = wrap(require('timer').setTimeout)
  -- Can wrap modules too
  local fs = wrap(require('fs'), true) -- true means to auto-handle errors

  local fd = fs.open(__filename, 'r', "0644")
  local stat = fs.fstat(fd)
  local offset = 0
  repeat
    local chunk, length = fs.read(fd, offset, 40)
    offset = offset + length
  until length == 0
  sleep(1000)
  fs.close(fd)
  return fd, stat, offset
end, callback)
```
LuaJit Bindings

There are two ways to call C libraries from scripts in LuaJit:

- One is the Application Program Interface.
- Write special lua-callable C functions that wrap C libraries.
- Works the same in LuaJit and stock Lua
- Two is using LuaJit’s FFI interface.
- Write bindings in Lua, no compile step needed.
/* uvffi.h */
enum { MAX_TITLE_LENGTH = 8192 }
struct uv_err_s { int code; int sys_errno_; }
typedef struct uv_err_s uv_err_t;
uv_err_t uv_get_process_title(char* buffer, size_t size);
uv_err_t uv_set_process_title(const char* title);
const char* uv_strerror(uv_err_t err);
const char* uv_err_name(uv_err_t err);
ffi.cdef(fs.readFileSync("uvffi.h"))
local C = ffi.C
local uv = {}

local function uvCheck(err)
    if err.code == 0 then return end
    local name = ffi.string(C.uv_err_name(err))
    local message = ffi.string(C.uv_strerror(err))
    error(name .. "": " .. message)
end

function uv.getProcessTitle()
    local buffer = ffi.new("char[MAX_TITLE_LENGTH]"
    uvCheck(C.uv_get_process_title(buffer, C.MAX_TITLE_LENGTH))
    return ffi.string(buffer)
end

function uv.setProcessTitle(title)
    uvCheck(C.uv_set_process_title_title(title))
end
int luv_get_process_title(lua_State* L) {
    char title[8192];
    uv_err_t err = uv_get_process_title(title, 8192);
    if (err.code) {
        return lual_error(L, "%s: %s", uv_err_name(err), uv_strerror(err));
    }
    lua_pushstring(L, title);
    return 1;
}

LUALIB_API int luaopen_uv_native (lua_State* L) {
    lua_newtable (L);
    lua_pushcfunction(L, luv_get_process_title);
    lua_setfield(L, -1, "getProcessTitle");
    return 1;
}
When I started, LuaJit FFI didn’t support callbacks.

Even after callbacks were added, they didn’t support passing struct values (only references).

Rackspace was using stock Lua for their Agent program.
typedef uv_buf_t (*uv_alloc_cb)(uv_handle_t* handle, size_t suggested_size);
typedef void (*uv_read_cb)(uv_stream_t* stream, ssize_t nread, uv_buf_t buf);
typedef void (*uv_read2_cb)(uv_pipe_t* pipe, ssize_t nread, uv_buf_t buf,
                            uv_handle_type pending);
typedef void (*uv_write_cb)(uv_write_t* req, int status);
typedef void (*uv_connect_cb)(uv_connect_t* req, int status);
typedef void (*uv_shutdown_cb)(uv_shutdown_t* req, int status);
typedef void (*uv_connection_cb)(uv_stream_t* server, int status);
typedef void (*uv_close_cb)(uv_handle_t* handle);
typedef void (*uv_timer_cb)(uv_timer_t* handle, int status);
typedef void (*uv_exit_cb)(uv_process_t*, int exit_status, int term_signal);
typedef void (*uv_fs_cb)(uv_fs_t* req);
Through the Thicket
Bind all the other UV Functions

**FS** - Has open, close, read, write, unlink, mkdir, rmdir, readdir, stat, fstat, rename, fsync, fdatasync, ftruncate, sendfile, chmod, utime, futime, lstat, link, symlink, readlink, fchmod, chown, and fchown.

**UV** - Has run, ref,_unref, updateElapsedTime, now, hrtime, getFreeMemory, getTotalMemory, loadavg, uptime, cpuInfo, interfaceAddresses, execpath, getProcessTitle, setProcessTitle, handleType and activateSignalHandler.
Bind all the libUV Types

- **Handle** - Has **close**.
- **Stream** - Has **shutdown**, **listen**, **accept**, **readStart**, **readStop**, and **write**.
- **Tcp** - Has **nodelay**, **keepalive**, **bind**, **getSockName**, **getPeerName**, **connect**, and **writeQueueSize**.
- **Udp** - Has **bind**, **setMembership**, **getsockname**, **send**, **recvStart** and **recvStop**.
- **Pipe** - Has **open**, **bind**, and **connect**.
- **Tty** - Has **setMode**, **getWinsize** and **resetMode**.
- **Timer** - Has **start**, **stop**, **again**, **setRepeat**, **getRepeat** and **getActive**.
- **Process** - Has **spawn**, **kill** and **getPid**.
- **Watcher** - has **newFsWatcher**.
Callbacks are Hard (in C)

- Unlike JavaScript, Lua, and Candor, C has **no anonymous functions**.
- It also does **not have closures**.
- Since callbacks are **global** and shared, state needs to be **stored** somewhere.
- The `uv_handle_t` and `uv_req_t` base types have a `void*` **data** property.
- Using this we can store anything in custom **structs**.
Memory Management is Hard

- In C of course we have to manage everything manually. That’s expected.
- LibUV does **not assume much** and stays out of the way.
- However, sometimes a pending C **callback** will need a script object that has **no roots** in any script.
- Doing this **properly** without being too **greedy** or too **loose** is tricky.
- The recent libUV **refcount** refactor helps.
Status Check

- Just binding libuv to Lua is over **12,000** lines of C code!
- We still need to write bindings for **http-parser**, **c-ares**, **openssl** and **zlib**.
- As you might have noticed, **libuv** is a very different API than **node**.
- We need a sugar layer written in Lua. (over **5,000** lines of Lua code)
- Building this **cross-platform** is a real pain.
- Lua does not have **JSON** built in, nor does it fit **100%** with the language.
Rackspace wanted to build something like Lua for their monitoring agent. So we joined forces.

Others joined as well from all over the world. (even as far as Russia and New Zealand)

The community helped finish out the project.

- Maintain a windows build.
- Write bindings for c-ares, openssl, zlib, and fix my udp bindings.
- Co-Designed the object system and much of the lua sugar layer.
The Destination is in Sight

- In initial testing Luvit was 4x faster than node.js in http hello world.
- Luvit also used 20x less ram than node in the same test.
- Most the basic functionality was in place and APIs were starting to converge with node’s.
- We made a webpage http://luvit.io, a mailing list luvit @ google groups, an irc room #luvit @ freenode.
- We made a few releases with pre-built binaries for many platforms.
Lessons Learned

- Lua as a language is **compatible** with Node.js **API** patterns.
- LuaJit uses a lot **less ram** than V8.
- Calling C code in luajit is much **faster** than in V8.
- Though as the **script** code grows, V8 tends to be faster.
- Coroutines allow **faux-blocking** using the native language features.
- Streaming **JSON** is really cool. But **LTIN** is more natural for Lua!
Lessons Learned

- Small codebases make developers happy. Luvit would build in 4 seconds on my desktop.
- Node’s `require` system and especially path resolving algorithm is worth copying.
- JavaScript is hard to beat for web development. It’s a requirement for the browser, so it’s natural for the server.
- Lua rocks for other types of development, especially with FFI.
Lessons Learned

Open Source collaboration makes it possible to build anything.
Final Destination

Now where do we go from here?
Do it Again!

Choose your pet scripting language and **bind** it to libUV.

**Explore** the strengths of the new platform.

**Share** what you learned.

**Contribute** back.
Candor.IO

- Candor is a new programming language by Fedor Indutny.
- It was called Dot, but I proposed the new name Candor.
- Candor is to JavaScript as C is to C++.
- Except it’s implemented in C++.
- Candor.IO is libUV bindings to the language.
new = (prototype, args...) {
    obj = clone prototype
    obj:initialize(args...)
    return obj
}

Rectangle = {}
Rectangle.getArea = (self) {
    return self.w * self.h
}
Rectangle.initialize = (self, w, h) {
    self.w = w
    self.h = h
}
p("Rectangle", Rectangle)

Square = clone Rectangle
Square.initialize = (self, s) {
    self.w = s
    self.h = s
}
p("Square", Square)

rect = new(Rectangle, 3, 5)
p("rect", rect)
print("Rectangle 3x5 = " + rect:getArea())

square = new(Square, 4)
p("square", square)
print("Square 4x4 = " + square:getArea())
using namespace candor;

static Value* luv_timer_stop(uint32_t argc, Value* argv[]) {
    assert(argc == 1);
    Object* obj = argv[0]->As<Object>();
    uv_timer_t* handle = UVData::ObjectTo<uv_timer_t>(obj);
    int status = uv_timer_stop(handle);
    return Number::NewIntegral(status);
}

static Handle<Object> module;
Object* uv_timer_module() {
    if (!module.IsEmpty()) return *module;
    module.Wrap(Object::New());
    module->Set("create", Function::New(luv_create_timer));
    return *module;
}
// Load the libraries we need
prettyprint = global.prettyprint
require = global.require
exit = global.exit
lastError = require('uv').lastError
Timer = require('timer')

// Helper to check uv return values.
check = (status) {
  if (!status) return
  err = lastError()
  prettyprint(err)
  exit(err.code)
}

// setTimeout, then stop and close

// set Timeout, then stop and close

Calling libuv from Candor

Calling libuv from Candor
Porting Node.js to SpiderMonkey.

This means JavaScript 1.8.5 vs EcmaScript 5 in V8.

Generators are already implemented.

VM competition for Node.js is a good thing.
static JSBool luv_run(JSContext *cx, unsigned argc, jsval *vp) {
    uv_run(uv_default_loop());
    JS_SET_RVAL(cx, vp, JSVAL_VOID);
    return JS_TRUE;
}

static JSFunctionSpec luv_functions[] = {
    JS_FS("run", luv_run, 0, JSPROP_ENUMERATE),
    JS_FS_END
};

JSBool luv_init(JSContext *cx, unsigned argc, jsval *vp) {
    JSObject* uv = JS_NewObject(cx, NULL, NULL, NULL);
    if (!JS_DefineFunctions(cx, uv, luv_functions)) {
        return JS_FALSE;
    }
    JS_SET_RVAL(cx, vp, OBJECT_TO_JSVAL(uv));
    return JS_TRUE;
}
Thank You for the Journey.