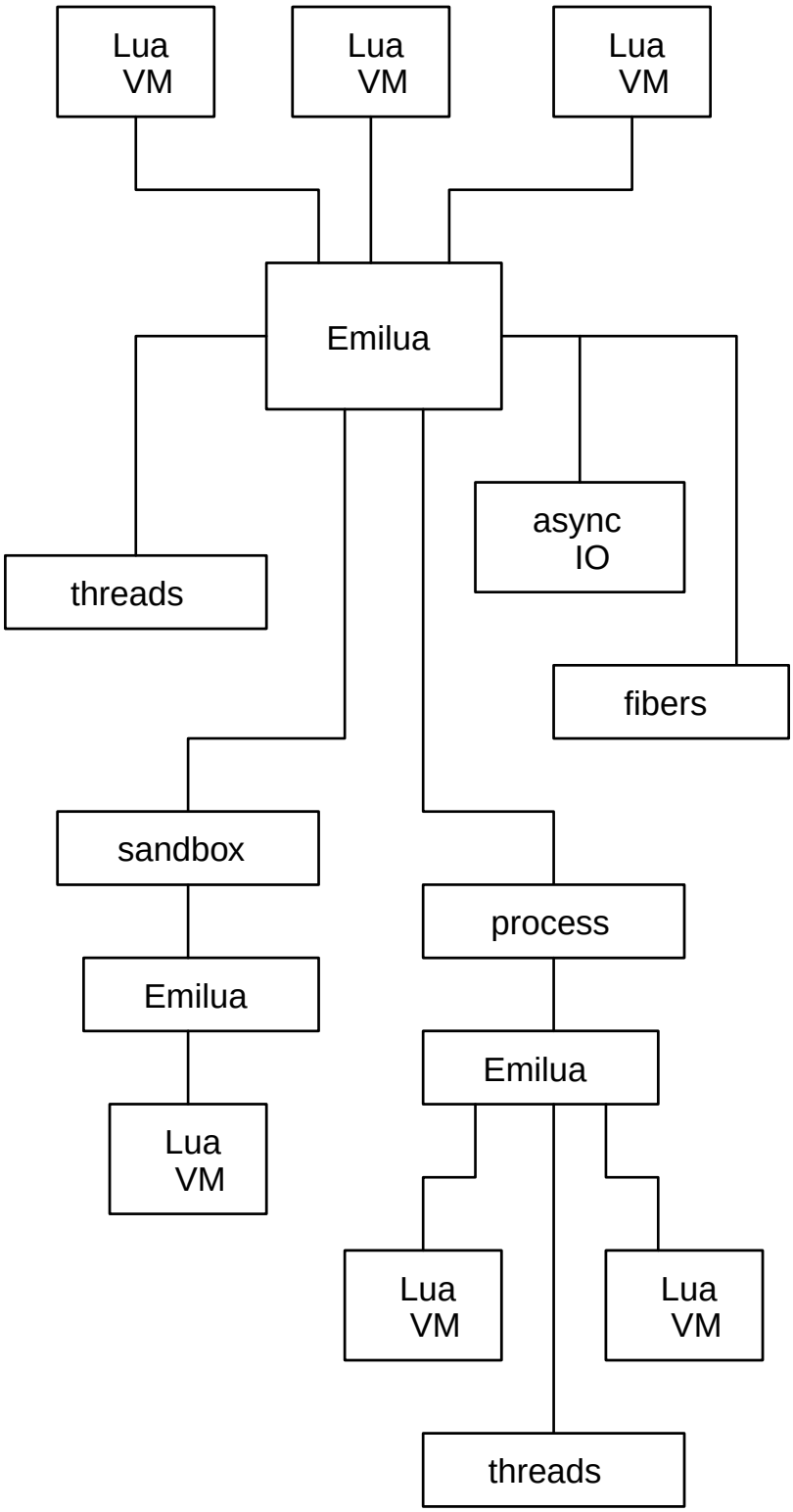
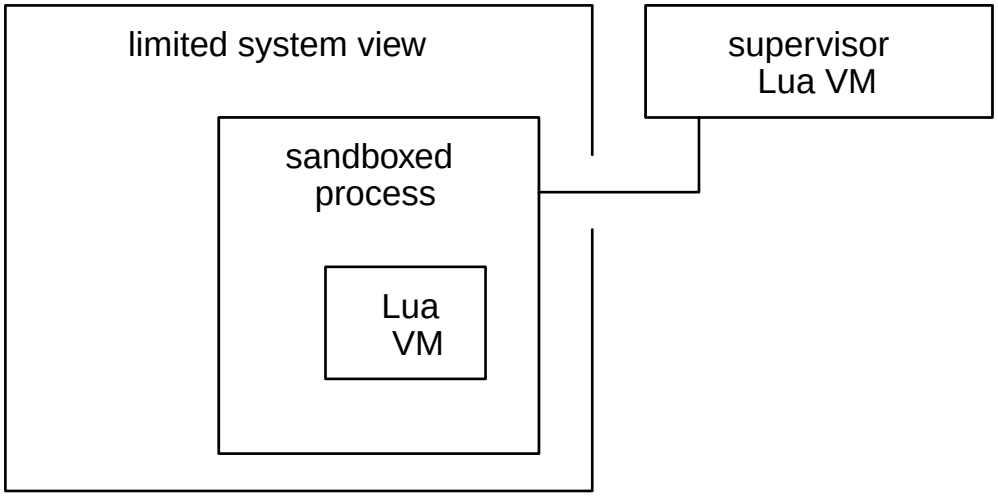
Emilua 0.6 reference documentation

Preface

Emilua

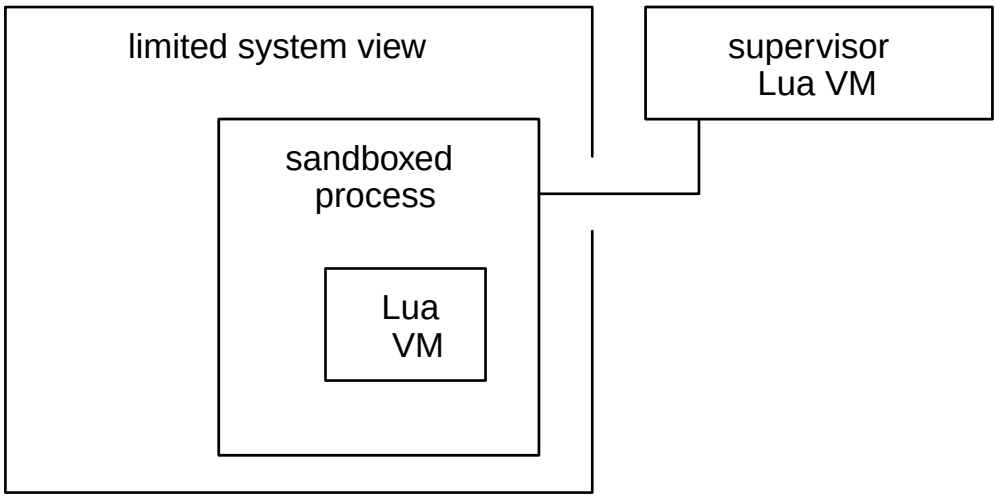


Emilua is an execution engine. As a runtime for your Lua programs, it'll orchestrate concurrent systems by providing proper primitives you can build upon.



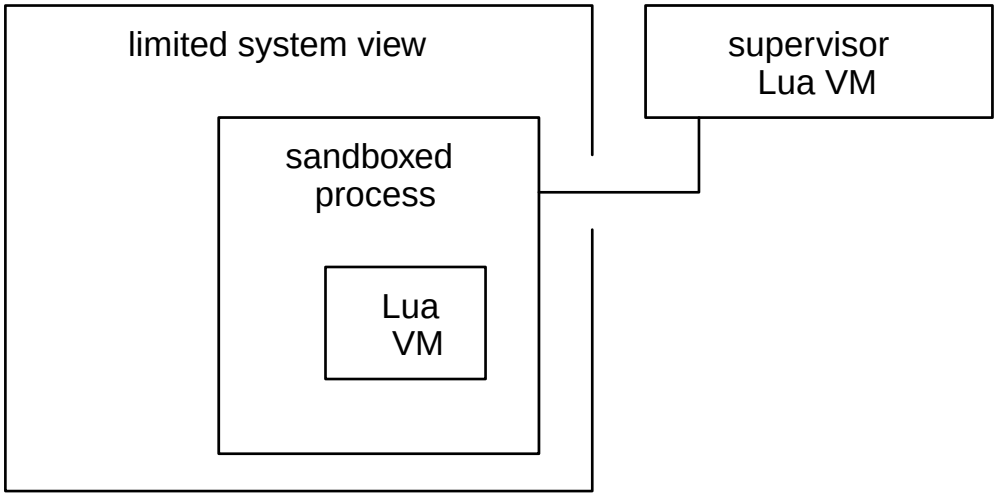
Emilua is not a framework. You don't design the structure of your software by extending a complex concurrency framework. On the contrary, you start **simple** and only makes use of primitives your application needs. Should you only have the need for simple serial programs, you'll have access to plenty of IO abstractions that work across a broad range of platforms.

Fibers



When your software grows and the need to increase the concurrency level a notch arises, just spawn fibers. The same IO abstractions that work on serial programs will work on concurrent programs as well. You don't need to pay an extra huge cost by completely refactoring your program during this transition[1].

Sandboxes



Emilua has first-class support for modern sandboxing technologies.

- Linux namespaces.
- Linux's Landlock.
- FreeBSD's jails.
- FreeBSD's Capsicum.

Mitigate risks by creating disposable cheap sandboxes to parse untrusted input data.

Sandboxing support on Emilua is based around capabilities and elegantly integrates with the same machinery that is used to implement the actor model.

> Compartmentalised application development is, of necessity, distributed application development, with software components running in different processes and communicating via message passing.
>
> — Capsicum: practical capabilities for UNIX, Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway

The only resource a sandbox starts with is `inbox` and its only method: `receive()`. In this initial state, a sandbox *can't even ask* for new resources (i.e. it's a push model). The Lua VM on the host system can then selectively choose which resources are safe to hand over (e.g. read-only access to a file and a pipe).

Cross-platform

- Windows.

- Linux.
- FreeBSD.

Emilua is powered by the battle-tested and scar-accumulating Boost.Asio library to drive IO and it'll make use of native APIs in a long list of supported platforms. However processor ISA compatibility will be limited by LuaJIT availability.

Network IO

- TCP.
- UDP.
- TLS.
- Address/service forward/reverse name resolution.
- IPv6 support (and mostly transparent).
- Cancellable operations transparently integrated into the fiber interruption API.
- Several generic algorithms.

IPC

- UNIX domain sockets (stream, datagram, and seqpacket).
- `SCM_RIGHTS` fd-passing.
- Pipes.
- UNIX signals.
- Ctty job control (and basic pty support).

Filesystem API

- It easily abstracts path manipulation for different platforms (e.g. POSIX & Windows).
- Transparently translates to UTF-8 while retaining the native representation for the underlying system under the hood.
- Directory iterators (flat and recursive).
- APIs to query attributes, manipulate permissions, and the like.
- Lots of algorithms (e.g. symlink-resolving path canonization, subtrees copying, etc).
- It focuses on cross-platform support, so not all operations are supported yet, but some platform-specific extensions are already available (e.g. non-Windows umask(3p)).

Misc features

- Complete fiber API (sync primitives, interruption API, clean-up handlers, fiber local storage, assert-like scheduling constraints, …).

- Integrates with Lua builtins (i.e. you can mix up fibers and coroutines, modules, ...).
- AWK-inspired scanner to parse textual streams easily.
- Clocks & timers.
- File IO (for proactors only[2], so the main thread never blocks).
- Serial ports.
- A basic regex module.
- Native JSON module.
- Portable error code comparison.
- And much more.

[1] Emilua doesn't suffer from Bob Nystrom' two colors problem.

[2] Right now, Windows' IOCP, and Linux's io_uring.

Conventions

Type annotations

Lua syntax is extended to document expected types.

Parameter types

Colon punctuation is used to denote the start of some type annotation after some variable name.

```
function some_function(arg1: number, arg2: string)
    -- ...
end
```

Return type

The characters `->` are used to denote the return type of a function.

```
function some_function() -> number
    -- ...
end

function another_function() -> string, number
    -- ...
end
```

Recognized types

- `nil`
- `boolean`
- `number`
- `integer`
- `string`
- `table`
- `function`

`value` may be used when we don't want to specify the return type for a function.

```
function yet_another_function() -> value
    -- ...
end
```

`unspecified` may be used to denote special values for which the actual type might change among Emilua versions. The user should avoid making any assumptions about the concrete type of such objects.

```
null: unspecified
```

Composite types

Type	Syntax	Example
Union type	`TYPE_1 | TYPE_2`	`file_descriptor | file.stream`
Array	`VALUE_TYPE[]`	`string[]`
Dictionary	`{ [KEY_TYPE]: VALUE_TYPE }`	`{ [string]: number }`

Literals

Literals may be used when only a subset of values are acceptable for some parameter.

```
function some_function(a: 0|1|2, b: "stdin"|file_descriptor)
    -- ...
end

function another_function(c: string) -> { foo: string, bar: number }
    -- ...
end
```

Optional parameters

Brackets may be used to denote optional parameters.

```
function a_function(required: string[, optional1: integer, optional2: boolean])
    -- ...
end

function send_file(
    self,
    file: file.random_access,
    offset: integer,
    size_in_bytes: integer,
    n_number_of_bytes_per_send: integer
    [, head: byte_span[, tail: byte_span]]
) -> integer
    -- ...
end

function another_function([foo: number]) -> string[]|byte_span[]
```

```
    -- ...
end
```

For this syntax, it's not necessary to specify `nil` as an optional accepted type.

Varargs

```
function fun(...: byte_span|string)
    -- ...
end

function fun2(command: string[, ...])
    -- ...
end

function fun3(n: integer) -> ip.address...
    -- ...
end
```

Overloads

If a function requires different explanations for each overload, code callouts are used to specify a overload.

```
function foo(file.stream)        ①
function foo(file.random_access) ②
```

① Lorem ipsum dolor sit amet, consectetur adipiscing elit

② sed do eiusmod tempor incididunt ut labore et dolore magna

Similar functions

Similar functions that take the same arguments may be documented together.

```
ip.tcp.get_address_info()
ip.tcp.get_address_v4_info()
ip.tcp.get_address_v6_info()
ip.udp.get_address_info()
ip.udp.get_address_v4_info()
ip.udp.get_address_v6_info()

function(host: string|ip.address, service: string|integer[, flags: integer]) -> table
```

Brace expansion as in BASH may appear in section titles to denote the functions that are similar and documented together. However the full name for each function will still appear at the start of the body for these sections.

1. `this_fiber.{disable,restore}_interruption()`

```
this_fiber.disable_interruption()
this_fiber.restore_interruption()
```

Check the interruption tutorial to see what it does.

Named parameters

For complex functions that accept too many options a table argument is used to emulate named parameters. The parameters will then be defined in the text that follows.

`parameter_a: string`

Lorem ipsum

If a parameter is optional, then `nil` will be OR'ed among the valid types.

`parameter_b: string|nil`

Lorem ipsum

Another way to specify an optional parameter is to give it a default value. If a default value exists, it'll be used instead of `nil`. In this case, `nil` may be omitted. The default value follows an equals sign.

`parameter_c: boolean = false`

Lorem ipsum

`parameter_d: number = unspecified`

Lorem ipsum

If a parameter might accept different types, nested definition lists in the text may be used to define the behavior for each type.

`parameter_e: string|number`

`string`

Lorem ipsum

`number`

dolor sit amet

If nested parameters exist, we'll omit the `table` specification for the nested parameters, and directly document each submember using a dot-notation.

`parameter_f.foo: string`

Lorem ipsum

`parameter_f.bar: number`

dolor sit amet

`self`

It's safe to assume that any function that takes `self` as the first argument is not available as a free function in the module. These functions can only be accessed through the `__index`'s metamethod on the given object.

If a function is also available as a free function in the module, an explicit overload will be documented.

```
function append(self, ...: byte_span|string|nil) -> byte_span ①
function append(...: byte_span|string|nil) -> byte_span       ②
```

When only the free function is available in that module, the term `self` won't be used.

```
function append(o: byte_span[, ...])
    -- ...
end
```

ChangeLog

0.6

- Add FreeBSD's jails support.
- Add function `format()` to format strings. The implementation uses C++'s libfmt.
- Convert decomposition functions from filesystem.path to properties: `root_name`, `root_directory`, `root_path`, `relative_path`, `parent_path`, `filename`, `stem`, `extension`.
- Convert some filesystem.path properties to string: `root_name`, `root_directory`, `filename`, `stem`, `extension`.
- filesystem.path.iterator() will return strings at each iteration now.
- Add more functions to the module filesystem: `exists()`, `is_block_file()`, `is_character_file()`, `is_directory()`, `is_fifo()`, `is_other()`, `is_regular_file()`, `is_socket()`, `is_symlink()`, `mode()`. It was already possible to query for these attributes. These functions were added as an extra convenience.
- Add yet more functions to the module filesystem: `mkfifo()`, `mknod()`, `makedev()`.
- New UNIX socket options to retrieve security labels and credentials from the remote process.
- Remove HTTP & WebSocket classes. They should be offered as separate plugins.
- `file_descriptor` implemented for Windows pipes and `file.stream`.
- Many improvements to Windows version of `system.spawn()`.

0.5

- Add `mutex.try_lock()`.
- Add module `recursive_mutex`.
- Add module `future`.
- Add `filesystem.chown()`.
- Enable IPC-based actors on all UNIX systems.
- spawn_vm() performs the same module path resolution from require() now. That means it's possible to use root-imports from spawn_vm().
- spawn_vm() parameters refactored (API break).
- Add Linux Landlock support.
- Add FreeBSD Capsicum support.

0.4

- A new `byte_span` type akin to Go slices is used for IO ops.
- Actor channels now can transceive file descriptors.

- Support for Linux namespaces. Now you can set up sandboxes and run isolated actors (or just the well-known containers).
- Upgrade to C++20. The motivating feature for the upgrade was `std::atomic<std::weak_ptr<T>>`. However, other C++20 features are being used as well.
- Removed `println()`.
- Removed `sleep_for`. Its functionality has been replaced by the module `time`.
- Moved `steady_timer` to the new module `time`.
- Removed `ip.tcp.resolver`. Its functionality has been replaced by `ip.get_address_info()`.
- `tls.ctx` renamed to `tls.context`.
- Modules `ip` and `tls` grew a lot. The API for sockets now supports IO ops on `byte_span` instances, and plenty of new functions and classes (including UDP) were added.
- `inbox.recv()` renamed to `inbox.receive()`
- Module `cond` renamed to `condition_variable`.
- `error_code.cat` renamed to `error_code.category`.
- `spawn_ctx_threads()` renamed to `spawn_context_threads()`.
- `inherit_ctx` renamed to `inherit_context` in `spawn_vm()`.
- New modules.
  - `time`: clocks and timers.
  - `pipe`.
  - `unix`: UNIX domain sockets.
  - `serial_port`: serial ports.
  - `system`: UNIX signals, CLI args, env vars, process credentials, and much more.
  - `file`: file IO. Only available on systems with proactors (e.g. Windows with IOCP, and Linux with io_uring). BSD can still be supported later (with kqueue + POSIX AIO).
  - `filesystem`: portable path-manipulation, and plenty of filesystem operations & algorithms.
  - `stream`: AWK-inspired scanner and common stream algorithms.
  - `regex`: Basic regex functions. The interface has been inspired by C++, Python and AWK.
  - `generic_error`: portable error comparison for filesystem, sockets, and much more.
  - `asio_error`: errors thrown by the asio layer.
  - `websocket`.
- Lua programs can define their own error categories now.
- Several new OS-specific APIs (e.g. Linux capabilities, and Windows' `TransmitFile()`).
- Add `http.request.upgrade_desired()`.
- `http.socket` can work on top of UNIX domain stream sockets now.
- Now Emilua is less liberal on accepted values for env var `EMILUA_COLORS`.
- Finer-grained cancellation of IO ops.

- Locales are set at application startup.
- Bug fixes.
- The build system now makes use of Meson's wrap system.
- Documentation can now be installed as manpages.
- Support for io_uring.

0.3

- HTTP request and response objects now use read-write locks and there is some limited sharing that you can do with them without stumbling upon EBUSY errors.
- Improvements to the module system (that's the main feature for this release). You should be able to use guix as the package manager for your emilua projects.
- EMILUA_PATH environment variable.
- Native plugins API (it can be disabled at build configure time).
- Add logging module.
- Add manpage.
- `--version` CLI arg.
- Build configure options to disable threading.
- Use fmtlib from host system.

0.2

- Fix build when compiler is GCC.
- Refactor module system. The new module system is incompatible with the previous one. Please refer to the documentation.
- Add HTTP query function: `http.request.continue_required()`.
- Remove `failed_to_load_module` error code. Now you should see `"iostream error"` or other more informative error reasons upon a failed module load.
- Numeric values for error codes changed.

Tutorials

Getting started

Perhaps Lua's best-known feature is its portability. Its reference implementation from PUC-Rio is written in plain ANSI C and it's very easy to embed in any larger program.

However limiting Lua to ANSI C has a high toll attached. Any useful program interacts with the external world (i.e. it must perform IO operations), and approaching portability by limiting oneself to ANSI C has consequences:

- Many useful IO operations don't belong to ANSI C's scope (you can't even perform socket operations).
- Not every operation will use the most efficient approach for the underlying system.
- There aren't even APIs to create threads, nor to multiplex IO requests in the same thread, so at most you can handle half-duplex protocols.

Another approach to portability — the one chosen by Emilua — is to have a different implementation for every OS. So your Lua program can make use of portable interfaces that require different underlying implementations. That also seems to be the approach taken by luapower[1].

Furthermore, if efficient operations exist to deal with patterns specific to some OSes, they are available when your Lua program runs in them (as long as they don't conflict with the proactor model[2]). For instance, you can make use of `TransmitFile()` when your program runs in Windows. It's expected that more of these interfaces will appear in future Emilua releases.

Hello World

```
print("Hello World")
```

Or, using the streams API:

```
local system = require "system"
local stream = require "stream"

stream.write_all(system.out, "Hello World\n")
```

Emilua doesn't expose native handles (e.g. file descriptors, or Windows `HANDLE` objects) for the underlying system directly. Instead they're wrapped into IO objects that expose a portable & safe interface (they'd also be type-safe in statically typed languages). You can't accept connections on a pipe handle, and Emilua doesn't worry about such impossible use cases.

Many of the interfaces used in Emilua are inspired by Douglas C. Schmidt's work in Pattern-Oriented Software Architecture.

The standard stream handles — `stdin`, `stdout`, and `stderr` — are available in the module `"system"`.

They model the interface for streams. The module `"stream"` contains useful functions to manipulate these objects.

Many other types modeling streams exist in Emilua such as files, pipes, serial ports, TCP and TLS connections.

A stream can be further broken down into read streams and write streams. `system.out` models a write stream. Write streams contain the following method:

`write_some(self, buffer: byte_span) → integer`

Writes `buffer` into the stream and returns the number of bytes written.

On errors, an exception containing the error code generated by the OS is raised.

Writes are not atomic (unless guaranteed by the underlying system under certain scenarios). To portably write the whole buffer into the stream, we must keep calling `write_some()` until the buffer is fully drained (Emilua won't automatically and inappropriately buffer data behind your back). That's what `stream.write_all()` does. Another boilerplate taken care of by `stream.write_all()` is creating a network buffer out of a string object.

Async IO

In truly async IO APIs, the network buffer must stay alive until the operation completes. So — for network buffers — Emilua uses a type independent of the Lua VM lifetime. If you call `system.exit()` to kill the calling VM, the network buffers participating in outstanding IO operations will stay alive until the respective operations finish (but killing the VM will also send a signal to cancel such associated outstanding IO operations).

`byte_span` is modeled after Golang slices, but many more algorithms (mostly string-related) are available as well.

The initiating function (such as `read_some()`) signals to the operating system that it should start an asynchronous operation, but the operation itself hardly involves the CPU at all. So if there's nothing else to execute, the CPU would idle until notified of external events. Keeping the CPU spinning will not make the IO happen faster. Making more CPU cores spin won't make the IO operation run faster. Once the request is sent to the kernel (and then further forwarded to the controller), the CPU is free to perform other tasks.

That's what async IO means. The IO operation happens asynchronously to the program execution. However signaling that the IO operation has completed (the IO completion event) doesn't need to be asynchronous.

> Delay not, Caesar. Read it instantly.
>
> — Shakespeare, Julius Caesar, 3, I

> Here is a letter, read it at your leisure.

> — Shakespeare, Merchant of Venice, 5, I
>
> — Quoted in "VMS Internals and Data Structures", V4.4, when referring to I/O system services

There is a lot more to this topic. However, for the Lua programmer, the topic ends here (pretty boring, huh?).

Concurrent IO

The initiating function blocks the current fiber until the operation finishes. However, as we saw earlier, this would be the perfect moment to perform other tasks and schedule more IO operations.

A trend we see in modern times is that of lazy frameworks to solve the async IO problem first and foremost. Only then when their authors stumble on the problem of concurrent programming[3] they're forced to do something about it, and they keep ignoring it by offering lame ad-hoc tooling around it[4]. Emilua is different. The first versions of Emilua were all focused on offering a solid execution engine for concurrent programming. And once this foundation was solid, a new version was released with plenty of IO operations integrated.

Emilua — as the execution engine — will schedule fibers and actors in a cooperative multitasking fashion. Once the initiating function forwards the request to the kernel, Emilua will choose the next ready task to run and schedule it (be it a fiber, be it an actor).

Emilua is focused on scalability and throughput. A solution for latency-oriented problems could be offered as well, but as of this writing it doesn't exist.

So, if you want to perform background tasks while the IO operation is in progress, just schedule a new task before you call the initiating function.

Spawning new fibers

Just call `spawn()` passing the start function and a new fiber will be scheduled for near execution.

```
local system = require "system"
local stream = require "stream"
local sleep = require "time".sleep

spawn(function()
    -- WARNING: Please, do not ever use timers to synchronize
    -- tasks in your programs. This is just an example.
    sleep(1)

    stream.write_all(system.out, " World\n")
end):detach()

stream.write_all(system.out, "Hello")
```

Spawning new actors

Just call `spawn_vm()` passing the start module and a new Lua VM will be created and scheduled for near execution.

```
local system = require "system"
local stream = require "stream"

if _CONTEXT == 'main' then
    spawn_vm('.')
    stream.write_all(system.out, "Hello")
else assert(_CONTEXT == 'worker')
    require "time".sleep(1)
    stream.write_all(system.out, " World\n")
end
```

Choosing between fibers and actors

Fibers share memory, and failing to handle errors in certain well-defined scenarios will bring down the whole Lua VM. If you need a slightly higher degree of protection against dirty code, spawn actors.

Lua VMs represent actors in Emilua. Different actors share no memory. That has an associated cost, and it's also inconvenient for certain common patterns. If you aren't certain which model to choose, go with fibers.

If you saturated your single-core performance already, an easy way to extract more performance of the underlying system is most likely to spawn new threads. Lua isn't a thread-safe language, so you need to spawn more Lua VMs (i.e. actors), and a few threads as well.

You can also mix both approaches.

Hello sleepsort

One really useful algorithm to quickly showcase a good deal of design for execution engines is sleepsort. In a nutshell, sleepsort sorts numbers by waiting N units of time before printing N, and this process is executed concurrently for each item in the list.

```
local sleep = require('time').sleep

local numbers = {8, 42, 38, 111, 2, 39, 1}

for _, n in pairs(numbers) do
    spawn(function()
        sleep(n / 100)
        print(n)
    end)
```

```
end
```

The above program will print the numbers in sorted order.

Cancellable operations

IO operations might never complete, so serious execution engines will expose some way to cancel them. There's a huge tutorial just on this topic and you're encouraged to read it: emilua-interruption(7).

Adding a timeout argument for each operation is a lame way to solve this problem[5], and Emilua wants no part in this trend. However, if that's how you really want to solve your problems, here's one way to do it:

```
local sleep = require('time').sleep

function op_with_timeout(op, timeout)
    local f_op = spawn(op)
    local f_timer = spawn(function()
        sleep(timeout)
        f_op:interrupt()
    end)

    local ret = {f_op:join()}
    f_timer:interrupt()
    return unpack(ret)
end

-- USAGE EXAMPLE

local ip = require 'ip'

local acceptor = ip.tcp.acceptor.new()
acceptor:open('v4')
acceptor:set_option('reuse_address', true)
if not pcall(function() acceptor:bind(ip.address.loopback_v4(), 8080) end) then
    acceptor:bind(ip.address.loopback_v4(), 0)
end
print('Listening on ' .. ip.tostring(acceptor.local_address, acceptor.local_port))
acceptor:listen()

local sock = op_with_timeout(function() return acceptor:accept() end, 5000)
print(getmetatable(sock))
```

Final notes

That's the gist of using Emilua. The interfaces mimic their counterpart in the non-async world, and it's usually obvious what the program is doing even when there's a huge theoretical background

behind it all.

We try to follow the principle of no-surprises. One operation in Emilua is roughly equivalent to one syscall in the underlying OS, and we just pass the original error (if any) unmodified for the caller to handle instead of trying to do anything funny on the user's back.

If you don't need multitasking support, the program you write in Emilua won't look much different from a program written for an abstraction layer that just exposes small shims over the real syscalls. If you can write programs for blocking APIs, you can write programs for Emilua.

When you do need multitasking, Emilua is perhaps the most flexible solution for Lua programs. However, why is that so — how to make good use of all the tools, and what it's really being offered beyond the trivial — will be a topic of other tutorials.

Many of the topics barely scratched above could be further expanded into tutorials of their own. Browse the documentation pages to see what topics catch your attention.

[1] https://luapower.com/

[2] The exception to this rule are filesystem operations. Filesystem operations are available in Emilua regardless of whether the underlying system offers them as part of a proactor.

[3] Managing state, event notifications, wasteful pooling, forward progress, fairness, …

[4] Exceptions to this trend include Java's LOOM, Erlang, and Golang.

[5] Latency-oriented frameworks are not part of this criticism. They have a good excuse for it.

Working with streams

Streams are one of the fundamental concepts one has to deal with when working on IO. Streams represent channels where data flows as slices of bytes respecting certain properties (e.g. ordering).

Emilua exposes two concepts to work with streams. Write streams are objects that implement the method `write_some()`:

`write_some(self, buffer: byte_span) -> integer`

Writes `buffer` into the stream and returns the number of bytes written.

Similarly, read streams are objects that implement the method `read_some()`:

`read_some(self, buffer: byte_span) -> integer`

Reads into `buffer` and returns the number of bytes read.

Exceptions are used to communicate errors.

When the type of the stream is not informed (i.e. read or write), it's safe to assume the stream object implements both interfaces. Pipes are unidirectional, and separate classes exist to deal with each. On the other hand, TCP sockets are bidirectional and data can flow from any direction. Furthermore, many sockets allow one to shutdown one communication end so they can work unidirectionally as well.

Short reads and short writes

Streams represent streams of bytes, *with no implied message boundaries*.

Each operation on a stream roughly maps to a single syscall[1], and it may transfer fewer bytes than requested. This is referred to as a short read or short write.

Reasons why short writes occur include out of buffer space in kernels that don't expose proactors. The rationale for short reads is more obvious, and it should stay as an exercise for the reader (no pun intended).

To recover from short reads and short writes, one just has to try the operation again adjusting the buffer offsets. For instance, to fully drain the buffer for a write operation:

```
while #buffer > 0 do
    local nwritten = stream:write_some(buffer)
    buffer = buffer:slice(1 + nwritten)
end
```

The module `stream` already contains many of such algorithms. You may come up with your own algorithms as well taking the business rules of your application into consideration (e.g. combining newly arrived data into the next calls to `write_some()`). Alternatively, if you don't need portable code, and the underlying system offers extra guarantees, you may do away with some of this complexity.

Layering

Streams of bytes by themselves are hardly useful for application developers. Many patterns exist to have structured data on top:

- Fixed-length records (binary protocols).
- Fixed-length header + variably-sized data payload (binary protocols).
- Records delimited by certain character sequences (textual protocols).
- Combinations of the above (e.g. HTTP starts with a textual protocol of CRLF-delimited fields, and it might change to a fixed-length payload to read the body, and maybe change yet again to a textual protocol to extract the resulting JSON data).

Given a single protocol might require multiple strategies, it's important to offer algorithms that don't monopolize the stream object to themselves. The algorithms should be composable. The algorithms found in the module `stream` follow this guideline.

This composition of algorithms naturally build layers:

- Raw IO. The IO interfaces exposed by the OS. There's no interface for peeking data or putting data back. Once the data is extracted out of the stream, it's your responsibility to save it until needed.
- Buffered IO. Just as short reads might happen, so can "long" reads. Upon dispatching the message for processing that includes data until the delimiter, you must be careful to not discard extra data that represents the start of the next message. Buffered IO is built on top of raw IO by managing an user-space buffer (and an associated index for the current message) alongside with the IO object.
- Formatted IO. Built on top of buffered IO integrating a parser (for input), and/or a generator (for output). Now the user is no longer interacting with slices of bytes, but properly structured data and messages.

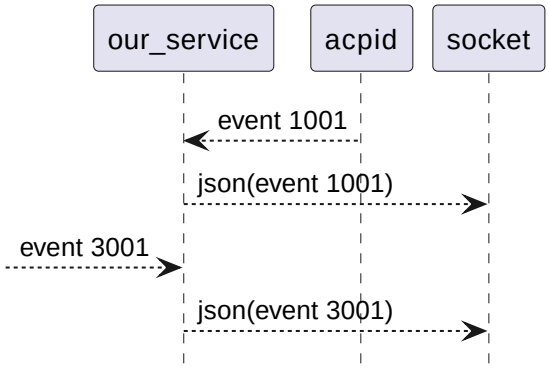
It's always easier to work with high-level formatted IO than low-level raw IO. However, when an implementation for the target protocol doesn't exist, you may have no other choice.

Emilua offers stream.scanner(3em) for generic formatted textual input.

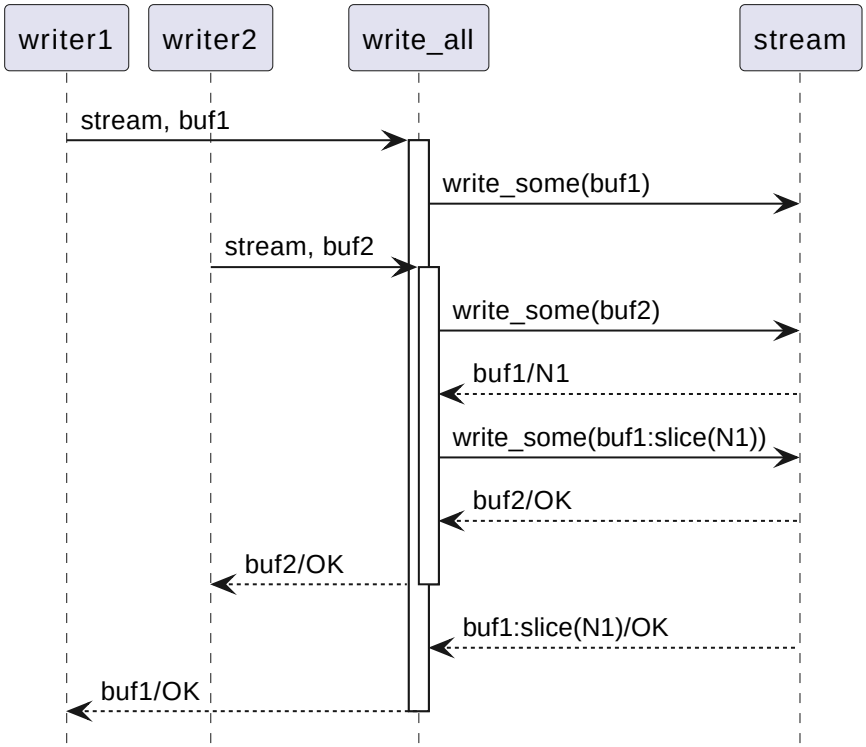
Composed operations

As it may already be clear by now, many algorithms are compositions of raw IO operations. Unless the IO object synchronizes access on its own (and explicitly says so), you should be careful to not initiate extra IO operations that might affect the already in-flight operations for that object.

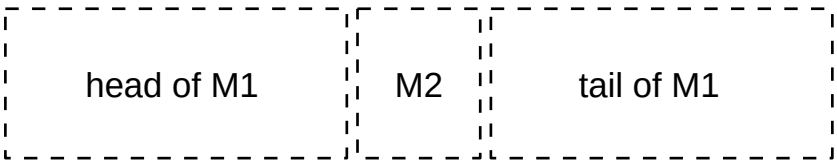
Concurrent writers operating on the same IO object is a common gotcha that causes corrupt streams during high-load scenarios (if "atomic" writes are not guaranteed by the underlying system). Suppose you're generating line-delimited JSON objects on a UNIX stream socket. You're collecting info from various system services (e.g. `"/run/acpid.socket"`), and for each event, you generate a new JSON object.



In other words, you're multiplexing information from assorted sources. The same can happen on the web when you're orchestrating microservices and dumping information on a WebSocket channel. Now, back to our example, if a short write happens, you might end up in the following state:



In other words, one of the messages didn't fit in the kernel buffer, then `stream.write_all()` retried the operation to drain the buffer. However there was already another in-flight write operation, and it was scheduled first than `buf1:slice(N1))`. The end result will be a stream where the second message is inserted in the middle of another message (a corrupt stream):



This problem is not exclusive to async IO frameworks. The same behavior can be observed if you code for blocking APIs making use of threads to achieve concurrency.

To solve this problem, you should create a mutex to protect the write end of the stream:

```
scope(function()
    stream_write_mtx:lock()
    scope_cleanup_push(function() stream_write_mtx:unlock() end)
    stream.write_all(stream, event_json)
end)
```

Other network frameworks for scripting languages try to solve the problem transparently by making use of an unbounded write buffer under the hood. However that's solving the issue in the wrong layer. If a write buffer is always used, the network framework can no longer appropriately communicate which user-issued write operation caused an error. The way such frameworks implement this solution is actually way worse as they face back-pressure issues as well, and have to band-aid patch the API all over.

Emilua will not inappropriately entangle all IO layers — raw IO, buffered IO, formatted IO — together. When you do want to make use of shared write buffers, you can write your own socket + the buffer (and mutex) to abstract this pattern in a way that won't cause problems to your application.

Do notice that such problems don't exist when composed operations use operations that don't overlap each other. For instance, you can use `stream.read_all()` and `stream.write_all()` on the same object with no synchronization because such use won't perform concurrent `write_some()` calls nor concurrent `read_some()` calls.

Why EOF is an error

Same rationale as Boost.Asio[2]:

- The end of a stream can cause stream.read_all(3em), stream.read_at_least(3em), and other composed operations to violate their contract (e.g. a read of N bytes may finish early due to EOF).
- An EOF error may be used to distinguish the end of a stream from a successful read of size 0.

See also

- https://techspot.zzzeek.org/2015/02/15/asynchronous-python-and-databases/
- https://sourceforge.net/p/asio/mailman/asio-users/thread/5357B16C.6070508%40mail1.stofanet.dk/

[1] That applies to IO objects that expose system resources (e.g. TCP sockets). Higher-level abstractions built in user-space (e.g. TLS sockets) don't apply.

[2] https://www.boost.org/doc/libs/1_81_0/doc/html/boost_asio/overview/core/streams.html

Alternative projects

Table 1. General concurrency models

	Fibers	Threads	Local actors	Distributed actors	Sandboxed actors[1]
cqueues[2]		✔			
Tarantool[3]	✔				
Effil[4]		✔			
Lanes[5]		✔			
Löve[6]		✔			
ConcurrentLua[7]			✔	✔	
luaproc[8]		✔			
Emilua	✔	✔	✔		✔

Do notice that the table won't go into many details. For instance, several projects allow you to use threads, but only Emilua is flexible enough that it actually allows you to create heterogeneous thread pools where some thread may be pinned to a single Lua VM while another thread is shared among several Lua VMs, and some work-stealing thread pool takes care of the rest. Too many tables would be needed to explore all the other differences.

Integrated IO engine also belongs to the comparison of concurrency models, but a separate table solely focused on them will be presented later (only mentioning the projects that do have one).

Table 2. NodeJS wannabes

	Fibers	Threads	Local actors	Sandboxed actors
Luvit[9]		✔		
LuaNode[10]				
nodish[11]				
Emilua (not a NodeJS wannabe)	✔	✔	✔	✔

When you create a project that tries to bring together the best of two worlds, you're also actually bringing together the worst of two worlds. This sums up most of the attempts to mirror NodeJS API:

- If everything is implemented correctly, it can only achieve being as bad as NodeJS is.
- Horrible back-pressure.

Table 3. IO engines

	Linux (epoll)	Linux (io_uring)	BSD (kqueue)	Windows
cqueues	✔		✔	
Tarantool	✔		✔	
Luvit	✔	✔	✔	✔
LuaNode	✔	✔	✔	✔
nodish	✔		✔	ugly[12]
Emilua	✔	✔	✔	✔

This document deliberately left some projects out of the comparison tables. The underlying reason is that it focuses on one problem space: the traditional userspace-in-a-modern-OS-box. Projects such as eLua[13], NodeMCU[14], XDPLua[15], and Snabb[16] will always have a space in the market. And the reason is quite simple: it's not possible to cater for very specific needs and be general at the same time. For instance, if you're trying to run something on the kernel side, there are specific restrictions and concerns that will further contaminate every dependant project down the line. It's not merely a question of porting the same API over. The mindset behind the whole program would need to change as well.

Emilua is young and there are plans to explore part of use cases that stretch just a little over the traditional userspace-in-a-modern-OS-box. However it still is a general cross-platform solution for an execution engine. It's still not possible to tackle very specific use cases and be general at the same time.

OpenResty

Most of the languages are not designed to make the programmer worry about memory allocation failing. Lua is no different. If you want to deal with resource exhaustion, C and C++ are the only good choices.

A web server written in lua exposed directly to the web is rarely a good idea as it can't properly handle allocation failures or do proper resource management in a few other areas.

OpenResty's core is a C application (nginx). The lua application that can be written on top is hosted by this C runtime that is well aware of the connections, the process resources and its relationships to each lua-written handler. The runtime then can perform proper resource management. Lua is a mere slave of this runtime, it doesn't really own anything.

This architecture works quite well while gives productivity to the web application developer. Emilua can't just compete with OpenResty. Go for OpenResty if you're doing an app exposed to the wide web.

Emilua can perform better for client apps that you deliver to customers. For instance, you might develop a torrent client with Emilua and it would work better than OpenResty. Emilua HTTP

interface is also designed more like a gateway interface, so we can, in the future, implement this interface as an OpenResty lib to easily allow porting apps over.

- Emilua can also be used behind a proper server.
- Emilua can be used to quickly prototype the architecture of apps to be written later in C++ using an API that resembles Boost.Asio a lot (and IOFiber will bring them even closer).
- In the future, Emilua will be able to make use of native plug-ins so you can offload much of the resource management.
- Emilua apps can do some level of resource (under)management by restricting the number of connections/fibers/…
- Emilua won't be that bad given its defaults (active async style, no implicit write buffer to deal with concurrent writes, many abstractions designed with back-pressure in mind, …).
- The actor model opens up some possibilities for Emilua's future (e.g. partition your app resources among multiple VMs and feel free to kill the bad VMs).

[1] Linux namespaces, Landlock, or Capsicum

[2] https://github.com/wahern/cqueues: Designed "to be unintrusive, composable, and embeddable within existing applications" [sic].

[3] https://www.tarantool.io/en/doc/2.1/reference/reference_lua/fiber/

[4] https://github.com/effil/effil

[5] http://lualanes.github.io/lanes/

[6] https://love2d.org/wiki/love.thread: Focused on game development.

[7] https://github.com/lefcha/concurrentlua: You could rewrite ConcurrentLua on top of Emilua, but you couldn't rewrite Emilua on top of ConcurrentLua.

[8] http://www.inf.puc-rio.br/~roberto/docs/ry08-05.pdf: It has a primitive model of what could become a full local actor system.

[9] https://luvit.io/

[10] https://github.com/ignacio/LuaNode

[11] https://github.com/lipp/nodish

[12] http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod#WIN32_PLATFORM_LIMITATIONS_AND_WORKA

[13] https://eluaproject.net/

[14] https://nodemcu.readthedocs.io/

[15] https://victornogueirario.github.io/xdplua/

[16] https://github.com/snabbco/snabb

Internals

The target public for this document are C++ programmers who want to delve into the project's code, not lua users. Native plug-in authors should also read this page.

The intent of this page is not to detail every internal of the project, but just to give an overview of the architecture. Details change quickly and documentation would lag behind, so they're avoided.

Once you read it, you should be familiar with the assumptions made thoroughly the project, and how to interact with the native code.

We assume that you already have some familiarity with the lua C API and Boost.Asio.

Multiple lua VMs

The project allows multiple OS threads to call `asio::io_context::run()`, so lua VMs can jump from one thread to another freely, but they will always refer to the same `asio::io_context` and each will be protected by its own ASIO strand.

```
-- Instantiates a new lua VM that shares
-- the caller's `asio::io_context`
spawn_vm(module)

-- Instantiates a new lua VM in a new
-- thread with its own `asio::io_context`
spawn_vm{ module=module, inherit_context=false }
```

You must specify a lua module name to run in the new VM, not a function. The module will be loaded and run in the new VM.

The only way for two different lua VMs to communicate is message passing. The channels are given when you instantiate the extra VMs. The channels accept a range of different values and will deep-copy them. You can also send references to IO objects, but the original references will be rendered unusable (their metatables are unset). Do pay attention to not let objects that have pending operations to be sent over (`EBUSY`, but do create an error code just for that).

Nor synchronization primitives (such as `mutex`) nor fiber handles can be sent over the channels and by implication can't be used to synchronize (or send cancellation requests to) fibers running in different lua VMs.

You can also send a channel over a channel. This will only send the channel "address" over and will allow complex routing among the lua VMs. If you send a channel's rx-end, the other side will receive a tx-channel anyway. On the C++-side, we need to implement a MPSC strand-based channel.

These characteristics should be enough to implement actor patterns. And it is not the job of emilua to enforce good patterns on applications. The patterns can be configured purely in the lua side of coding.

```
-- Spawn extra threads to the
-- caller's `asio::io_context`
spawn_context_threads(count)
```

Leaving the actor model aside for a moment, it's now easy to have threads with work-stealing (e.g. 8 lua VMs sharing the same `asio::io_context` running on 4 threads) so you don't have to worry about load-balancing.

Inside a single lua VM

When you issue some IO operation (including `chan:receive()`), the calling fiber will suspend, but other fibers from the same lua VM are allowed to kick in (cooperative multitasking). Fibers can share state with each other safely (and free from contention problems) as-if the program was single-threaded.

```
-- Spawn a new fiber on this lua VM
spawn(fn)
```

You can use the fiber handle just like you'd use a thread handle. There is `join()`, `detach()` and `interrupt()`.

All sync primitives obey some characteristics thanks to the restrictions we've laid out:

- They always live in the same strand. They never migrate strands.
- They don't synchronize with fibers from other strands (except for channels, but that's another story).

Given these conditions, it's now easier to implement and reason about the C++ code.

Only the C++ code that suspended the fiber can resume it back. If the operation should be cancellable, the async op should set an interrupter before suspending the fiber. No other code from the runtime will wake this fiber up. Once the interrupter is called, it'll be cleared automatically to prevent further complications on the async op implementation. The completion handler should also clear the interrupter to make sure it won't be (wrongly) reused for other operations.

A good level of serialization can be done by exploring these properties and simplify the implementation a lot. For once, you know no other code will wake the fiber up, so you can just as well call `io_obj.cancel()` on the interrupter and map `asio::error::operation_aborted` to `errc::interrupted` on the completion handler. A single handler (and no other) will take care of waking the fiber. There is no race to deal with here or anything alike.

A lot of the boilerplate is handled already on the prologue/epilogue functions from `vm_context`.

Userdata practices

Besides the common practices to create custom objects through userdata, Emilua (IO) objects will

also:

- Hide the metatable. By doing that, user code is prevented from changing the metatable (the metatable is just an usual table after all) that native code relies on.
- Assume `lua_setmetatable()` is an indivisible operation for userdata (i.e. if it fails, it doesn't set a metatable nor any `__gc` metamethod). This assumption is important to simplify object management by getting away with all pre-initialization tricks teached on Roberto's manuals and associated complexities.
- Assume `lua_setmetatable()` reports errors through exceptions (i.e. it always returns `1`). This is a superset of the previous point and it is *guaranteed* by the VM[1]. We don't really care as much about this point, but as *it is* guaranteed, the assumption described in the previous point (which we *do* care about) is covered as well.

C++ async operations

Let's begin with `require()`.

`require()`'ing a module is also an async operation which will suspend the caller fiber. Every module has its own isolated environment (i.e. a new lua thread is created for every module and that thread's environment is configured to use a separate lua table) sharing the same lua VM. The module's entry point is an user-provided source code evaluated to prepare the environment with the names that should be exported to the caller fiber. But this preparatory step may not be immediately ready and may need to call other async operations. The rule we define to mark a module as loaded and ready is when its main fiber finishes (synchronization code similar to `fiber:join()`).

To further enforce a more manageable project layout, it is only allowed to import new modules from the main fiber. This may introduce a "slow" startup in some project layouts, but:

- It is simpler to reason about the relationship of exported/imported names if we restrict them to the same main fiber. One such use we do of this feature is detecting whether the `inbox` module was loaded and close it if not.
- We are explicitly not aiming for remote modules (e.g. JS running on a web browser), so we don't need to care about slow startup happening in this event.
- In the cases where some module startup is indeed slow, the module programmer himself can adopt lazy loading techniques within his module's functions to have a quick startup with respect to the rest of the application.

Modules evaluate only once and are cached. We never unload them. We keep a reference to their lua thread for as long as the lua VM is active.

Loading a module forms a loader-loaded relationship. This relationship builds a chain that must be checked when a new module is `require()`d (so we can for instance prevent cyclic imports). But each module will have its own environment. This means the C++ function that implements `require()` needs to check lua-hidden state associated with the caller lua function (not a global one). That's the module system state per-module.

Rule

The per-module state is stored by using the module's main thread as a key in the fibers table. The fibers table is strong, but this isn't a problem because the module shall never be unloaded anyway. Code that unrefs fiber coroutines shall check whether the lua thread represents a module and skip removing it from the fibers table if so.

We can't store the module system data directly at the thread environment because lua code can change the thread environment by calling `setfenv(0, table)`.

We've already gone through the trickiest parts and added the most important restrictions to the table (no lua-related pun intended), so the remaining rules should be quick'n'easy to catch.

When you initiate an async operation, the C++ side will copy the `lua_State*` to handle the completion (or cancellation) later. However, any `LUA_ERRMEM` will trigger an emilua-call to `lua_close()` and `L` may then be invalid when we later try to resume it. So the completion handler need to check whether the vm is still valid before accessing it and this is the purpose of the `vm_context` structure (also protected by the same strand as the vm).

`this_fiber`

As long as lua code is executing, there is a current fiber and this property stays unchanged for as long as control doesn't return to host.

transparent, adj.

Being or pertaining to an existing, nontangible object.

> It's there, but you can't see it
>
> — IBM System/360 announcement, 1964

virtual, adj.

Being or pertaining to a tangible, nonexistent object.

> I can see it, but it's not there.
>
> — Lady Macbeth

This property is mostly transparent to lua code. Which is to say that the programmer is aware of this property, but there isn't a tangible object that it can track back to `this_fiber`. This is **mostly** true, but there is a quite tangible `this_fiber` lua global object that the user can inspect — exposed at the beginning of the first thread execution.

However, `this_fiber` being a global is shared among all the fibers, so it can't point to a single fiber. Instead, it will query which fiber is current and do operations on it.

C++ async ops will always store which fiber is current to know how to resume it back. And before a

fiber is resumed, this info is stored at a know lua registry's index so future async ops will get to know about it too. The reason why we can't rely on the `L` argument passed to C functions registered at the VM and the current fiber needs to be remembered is because there will be a `L` that points to the wrong lua thread as soon as the user wraps some function in a coroutine.

This design works well because we don't mix responsibilities of the scheduler with user code (as is the case for `Fiber#resume` in Ruby which would be better suited by a `Fiber#spawn()` that accepts `post`/`dispatch` execution policies and would avoid the (un-)parking unsound ideas altogether).

Asynchronous event notification

Some events are intrusive and will be generated even when no thread/fiber asked for them. The classical example are UNIX signals. A sighandler must be registered to handle them, but that begs the question: from which thread are these functions called? In the C world there are multiple answers:

`SIGEV_SIGNAL`
: The handler will be called asynchronously from any thread. That means a lot of restrictions to what a sighandler can do.

`SIGEV_THREAD`
: The handler will be called from an unspecified thread. Now we have way less restrictions, but some still exist (e.g. unsafe thread-local variables and thread cancelability state).

`SIGEV_KEVENT`
: The golden standard for event multiplexing in the C world.

Generally the need for asynchronous events spurs from bad design and should be avoided. However when integrating lua code to existing libraries we must deal with asynchronous events now and then. Emilua reserves a lua coroutine/thread for which no suspension is ever allowed and that will give the lua user a mix between `SIGEV_SIGNAL` and `SIGEV_THREAD` restrictions. From the handler the user can notify a condition variable to achieve friction-less handling from a different fiber similar to what `SIGEV_KEVENT` enables.

From the C++ side, one just needs to get the asynchronous event (lua) thread and rely on `lua_pcall()` (no need for complex `lua_resume()` handling, nor fiber APIs).

LUA_ERRMEM

Lua code cannot recover from allocation failures. As an example (and single-VM only):

```
my_mutex:lock()
scope_cleanup_push(function() my_mutex:unlock() end)
```

If the VM fails to allocate the closure passed to `scope_cleanup_push()`, `my_mutex` will be kept locked and the lua code inside that VM will be in an unrecoverable state. There's no pattern or ordering to make resource management work here as allocation failures can happen almost anywhere and we

then inherit some constraints and reasoning from preemptive scheduling. The only option (and this applies to **any** allocation failure reported by the lua VM when running arbitrary user code) is to terminate the VM from the C++-side.

When `lua_close()` is called, there is no guarantee pending operations will be canceled as they might hold strong references to the underlying IO object preventing its destructor from getting called. Therefore, the `vm_context` structure also holds an intrusive container of polymorphic elements which are destroyed after `lua_close()` is called and can be used to register cleanup code to avoid such leaks. If the operation finishes, the IO object is free to reclaim their own objects from this container and use them for other purposes.

`lua_CFunction` objects should never call `lua_close()`. If they detect `LUA_ERRMEM` all they have to do is to mark the flags field from `vm_context` and suspend the fiber. The host will take care of closing `lua_State*` and extra cleanup when it recovers control of the thread.

The other side of the coin is to *detect* `LUA_ERRMEM`. All interactions with the VM from the C API happens through the virtual stack, so naturally that's the first concern. You must not push anything on the stack if there's no extra free stack slot available. To check for such slot space, there's `lua_checkstack()`.

The usual C function signature is not enough to convey all the semantics required by the Lua C API. On the Functions and Types section from the manual, we verify the following information:

> Here we list all functions and types from the C API in alphabetical order. Each function has an indicator like this: `[-o, +p, x]`
>
> [...] The third field, `x`, tells whether the function may throw errors: '`-`' means the function never throws any error; '`m`' means the function may throw an error only due to not enough memory; '`e`' means the function may throw other kinds of errors; '`v`' means the function may throw an error on purpose.

The 5.1's signature for `lua_checkstack()` is:

```
int lua_checkstack(lua_State *L, int extra); // [-0, +0, m]
```

That's obviously bogus. If `lua_checkstack()` can throw on `ENOMEM` that means there is no possible safe interaction with the VM. That's — plain and simple — a bug. This bug was fixed in Lua 5.2 when the signature changed to:

```
int lua_checkstack(lua_State *L, int extra); // [-0, +0, ]
```

Lua 5.2 received a few other improvements concerning `ENOMEM` such as obsoleting `lua_cpcall()` by introducing light C functions. API-wise, Lua 5.2 was a great release as it fixed many shortcomings.

You don't *always* need to call `lua_checkstack()` before doing anything thanks to at least `LUA_MINSTACK` free stack slots being guaranteed for you when the VM calls into your `lua_CFunction` objects. And here's where things start to get tricky. Consider the following Lua code:

```
coroutine.wrap(function()
    spawn(function()
        print('Hello World')
    end)
end)()
```

The underlying C function implementing `spawn()` is exposed to 3 different `lua_State*` handles:

Current fiber

`get_vm_context(L).current_fiber()`. The one that calls `coroutine.wrap()`.

Inner coroutine

The `L` parameter from `lua_CFunction`. The one that calls `spawn()`.

New fiber

`lua_newthread(L)` return value. The one to print "*Hello World*".

If `lua_error()` is called on `L`, the stack for `L` will be in a completely deterministic state. Anything this `lua_CFunction` object pushed on the stack will be popped and the whole `pcall()`-chain on the state `L` will be respected too. However `lua_error()` might be called indirectly through other API functions. That's the signature for `lua_newtable()`:

```
void lua_newtable(lua_State *L); // [-0, +1, m]
```

As we've seen previously:

> '`m`' means the function may throw an error only due to not enough memory

"Throw" here means sorts of a call to `lua_error()` (`LUAI_THROW` to be more accurate). That's the `pcall()`-chain and each `lua_State` has its own (this property won't change even if you compile the Lua VM as C++ code). This independent `pcall()`-chain for each `lua_State` is not a limitation from the C API, but an accurate model of the underlying machinery happening in Lua code itself. Consider the following snippet:

```
c1 = coroutine.create(function()
    pcall(function()
        -- ...
    end)
end)
```

If `c1` is suspended in the middle of `pcall()`, it retains this private `pcall()`-chain that doesn't get mixed with `pcall()`-chains from other coroutines (i.e. the other `lua_State*` handles). Therefore the C

API accurately maps the language behaviour on retaining a private `pcall()`-chain for each `lua_State` and we can't expect any different behaviour here really. Lua documentation on the issue has been ironed out little-by-little throughout its releases. Lua 5.3 was the one to finally explicitly state the behaviour we just described:

> The panic function, as its name implies, is a mechanism of last resort. Programs should avoid it. As a general rule, when a C function is called by Lua with a Lua state, it can do whatever it wants on that Lua state, as it should be already protected. However, when C code operates on other Lua states (e.g., a Lua argument to the function, a Lua state stored in the registry, or the result of `lua_newthread`), it should use them only in API calls that cannot raise errors.
>
> — Lua 5.3 Reference

In short, that means our `spawn()` implementation that is exposed to the {`L`, current fiber, new fiber} triple would throw to the wrong `pcall()`-chain if it calls `lua_newtable(new_fiber)`. The solution is to use `lua_xmove()` when necessary and maintain **rigorous discipline** as to which C API functions are called on "foreign" `lua_State*` handles paying very special attention to their respective throw specifications. As for the discipline required, Rici Lake wrote a good summary on the lua-users wiki:

> There are quite a number of API functions which will never throw a Lua error. API functions that throw errors are identified in the reference manual as of 5.1.3. First, none of the stack adjustment functions throw errors; this includes `lua_pop`, `lua_gettop`, `lua_settop`, `lua_pushvalue`, `lua_insert`, `lua_replace` and `lua_remove`. If you provide incorrect indexes to these functions, or you haven't called `lua_checkstack`, then you're either going to get garbage or a segfault, but not a Lua error.
>
> None of the functions which push atomic data — `lua_pushnumber`, `lua_pushnil`, `lua_pushboolean` and `lua_pushlightuserdata` ever throw an error. API functions which push complex objects (strings, tables, closures, threads, full userdata) may throw a memory error. None of the type enquiry functions — `lua_is*`, `lua_type` and `lua_typename` — will ever throw an error, and neither will the functions which set/get metatables and environments. `lua_rawget`, `lua_rawgeti` and `lua_rawequal` will also never throw an error. Aside from `lua_tostring`, none of the `lua_to*` functions will throw an error, and you can avoid the possibility of `lua_tostring` throwing an out of memory error by first checking that the object is a string, using `lua_type`. `lua_rawset` and `lua_rawseti` may throw an out of memory error. The functions which may throw arbitrary errors are the ones which may call

> metamethods; these include all of the non-raw `get` and `set` functions, as well as `lua_equal` and `lua_lt`.

On a side note, Lua 5.2 added the following:

> If an error happens outside any protected environment, Lua calls a *panic function* (see `lua_atpanic`) and then calls `abort`, thus exiting the host application. Your panic function can avoid this exit by never returning (e.g., doing a long jump to your own recovery point outside Lua).
>
> The panic function runs as if it were a message handler (see §2.3); in particular, the error message is at the top of the stack. However, there is no guarantees about stack space. To push anything on the stack, the panic function should first check the available space (see §4.2).
>
> — Lua 5.2 Reference

That's actually behaviour that already existed on the version 5.1. An alternative panic function could just throw a C++ exception to implement this `__attribute__((noreturn))` behaviour. However this hypothetical panic function is not an alternative solution to our problems due to the combination of the following facts:

- As described elsewhere in this document, we require `lua_error()` to act as-if it throws a C++ exception so our destructors are properly called. That requires the underlying Lua VM (LuaJIT in our case) to throw and catch C++ exceptions.
- A C++-throw is triggered from `lua_newtable(L)`. The type thrown here is internal to the Lua VM and we cannot throw it ourselves. `LUA_ERRMEM` information is correctly preserved.
- A panic is triggered from `lua_newtable(new_fiber)`. Our panic function would in turn discard `LUA_ERRMEM` and throw a generic C++ exception.
- On `lua_newtable(new_fiber)` hitting `LUA_ERRMEM`, the `L`'s C++-catch handler wouldn't receive the original error (`LUA_ERRMEM`). That means information loss. That means our host code (the code that first calls into the Lua VM) won't call `lua_close()` (when it should) as its `lua_pcall()` /`lua_resume()` call might not report the correct error reason (`LUA_ERRMEM`). That also means the possibility to unwind the wrong number of cascaded `pcall()` blocks (a `pcall()` from Lua code is not supposed to handle `LUA_ERRMEM` — if correctly detected — so the number of blocks unwinded differs whenever `LUA_ERRMEM` is involved).
- Although LuaJIT can catch generic C++ exceptions, it lacks context and cannot possibly restore the stack state on each lateral `lua_State*` handle at play (the triple {`L`, current fiber, new fiber} in our case). If the `spawn()` `lua_CFunction` had a value pushed on the `current_fiber` stack when a `new_fiber` panic-triggered exception raises, the value on the `current_fiber` stack wouldn't be properly popped by the time `L` handles the C++ exception (and do remember that `L` is executing nested on top of `current_fiber` so you can already imagine the chaos here). In short, the Lua VM needs our cooperation to maintain some invariants.
- By wrapping these calls into our own C++ catch blocks we could work around some of these

issues, but the thought that thread control would still return to the Lua VM one last time *after* the panic handler got called is just too scary and previous mailing list threads on this topic weren't very reassuring. For one, if the exception is panic-triggered by `current_fiber`, we won't know what remains on this stack (except for the stack top), but that's exactly the `lua_State` that the host is operating on when our `lua_CFunction` got called on `L`. Even if control does return safely to our host it would still have problems to deal with there.

That covers our policy when implementing `lua_CFunction` objects. In short, we cannot resort to Lua panics here and the only real solution is the **rigorous discipline** on C API usage mentioned earlier.

Now let's talk about our policy for host code. The Lua suspending IO functions are implemented by querying which fiber is current and scheduling a `lua_resume()` on it as the callback for some Boost.Asio supported C++ `async_*()` function (plus a ton of other details properly documented elsewhere on this document such as strand handling and so on). The initiating function is called from the Lua VM, but the callback is not. The callback will act as the host.

Back to `lua_resume()`, this function itself doesn't throw:

```
int lua_resume(lua_State *L, int narg); // [-?, +?, ]
```

However the code that runs before `lua_resume()` might throw. This is the code that pushes the arguments to the coroutine. For instance, if a string is one of the coroutine parameters, you will have to use C API that might throw on `ENOMEM`:

```
void lua_pushlstring(lua_State *L, const char *s, size_t len); // [-0, +1, m]
```

It's no use trying to call `lua_pcall()` to wrap `lua_pushlstring()` here. `lua_state()` now returns `LUA_YIELD` and that means you can't use `lua_pcall()` on this `lua_State*` handle. You can't create a new handle and use the `lua_xmove()` trick either as `lua_newthread()` itself can throw on `ENOMEM`:

```
lua_State *lua_newthread(lua_State *L); // [-0, +1, m]
```

Fear not, for here is the place where we can finally use a panic function to throw a custom C++ exception. There are only two caveats. The first one is related to LuaJIT having such tight integration with native exceptions that it makes (almost) no distinction between `lua_pcall()` and C++ catch frames[2]. The net result is that you can use C++'s catch-all blocks and then no panic function will ever be involved (by now you must be feeling that we just travelled to the farthest candy shop in the kingdom just to make a full-turn just one block away from destination when we changed our minds and decided to go on the neighbour's candy shop). Despite the lack of a real panic function throwing our own exceptions, I'll still use the same previous terminology (i.e. panic-triggered exceptions).

The second caveat is a little charming race to avoid. The completion handler doing the host job is executed through the strand that protects the VM. If we let the exception escape the completion handler, another thread might try to use the VM before we have the chance to close it. In other words, the following approach has a race and thus is not used:

```
for (;;) {
    try {
        // Completion handler allows the panic
        // exception to escape here.
        ioctx.run();
        break;
    } catch (...) {
        // This is a bug. This code isn't executed
        // through the VM strand. A pending operation
        // that just finished could try to access
        // `current` from another thread while we're
        // here.
        vm_context* current = ...;
        current->close();
        continue;
    }
}
```

Therefore, it is responsibility from the completion handler to handle the panic-triggered exception (sorry about the boilerplate on your side, but that's the way it is).

```
try {
    // lua_push*() calls
} catch (...) {
    vm_ctx->close();
    return;
}
int res = lua_resume(fiber, narg);
```

That is enough to cover the policy for host code and finally finish the `LUA_ERRMEM` discussion too.

Channels and resources

The biggest challenge to cross-VM resource management are the multi-strand sync primitives (i.e. the channels). They have to execute code that jumps from one strand to another to finish their jobs. If the associated execution context already finished, then they would be stuck forever. The solution is for them to keep the execution context busy through a work guard.

However some rules are needed to make this work:

- Rx-channels (i.e. `inbox`) don't keep work guards.
- Tx-channels keep a work guard to the other end while they are alive. But they only keep a work guard to their own strands when they have an active operation.

If the tx-channels are not closed, they will prevent execution contexts that are no longer necessary from being destroyed. But that's the best we can do. We could periodically call the GC to free unused channels, but so will lua code anyway and there's nothing left for us to do on the C++ side. A

good practice for lua code would be to add the following chunk at the beginning of the fiber who's gonna process the actor messages:

```
scope_cleanup_push(function() inbox:close() end)
```

Extra rules for channels management:

- As an extra safety measure, if the main fiber finishes and `inbox` wasn't imported, the runtime closes it.
- Channels (tx and rx) also get closed when the VM is terminated.
- Channels must only upgrade their weak references to `vm_context` once they migrated to the target strand. Otherwise, they would prevent the VM from auto-closing (and hairy problems would follow).

The exception mechanism

C++ exceptions must not be used to propagate errors across lua/C++ frames. However, lua errors may simply trigger stack unwinding (the code makes heavy use of `setjmp()`) and we do depend on RAII to keep the code correct.

It is assumed that any call to `lua_error()` will behave as-if it throws a C++ exception (thus triggering our destructors). We require some support from the luaJIT VM for this. Specifically, we can't rely on the "no interoperability" category from their "exception" section on the "extensions" page because the following restriction:

> Throwing Lua errors across C++ frames will not call C++ destructors.

To make matters worse, the feature we do depend on only appears in the the "full interoperability" category:

> Throwing Lua errors across C++ frames is safe. C++ destructors will be called.

A different approach would be to implement an exception mechanism in terms of coroutines (although it'd add to code complexity):

> ```
> Exceptions < Coroutines < Continuations
> ```
>
> Exceptions can be thought of as a subclass of coroutines. You can implement an exception mechanism with coroutines.
>
> — leafo, leafo.net

But this path would be a dead-end as native lua errors would still be reported through `lua_error()`. For luaJIT, `lua_error()` plays well with our code because:

> The LuaJIT VM is fully resumable. This means you can yield from a coroutine even across contexts, where this would not possible with the standard Lua 5.1 VM: e.g. you can yield across `pcall()` and `xpcall()`, across iterators and across metamethods.
>
> — http://luajit.org/extensions.html#resumable

Wasn't for this guarantee, the project would be monstrous. To understand why this guarantee is important, let's unravel the fundamental pattern for fibers support. We always implicitly wrap every user code inside a lua coroutine:

```
local fib = coroutine.create(user_fn)
```

So async operations can suspend the calling fiber and resume them later.

But `user_fn` might very well contain a `pcall()` and execute our suspending async function inside it:

```
function user_fn()
    pcall(function()
        io_obj:emilua_async_op()
    end)
end
```

The exception mechanism should not block our ability to suspend fibers. When our own native code calls `lua_yield()` to suspend a fiber, the suspension mechanism should be able to cross the `pcall()` barrier.

To wrap all up so far, the standard lua exception mechanism is used to report errors. The only difference is that emilua will `lua_error()` a structured error object inspired by `std::error_code` for our own errors.

Things would get a little tricky on the following point that we raised previously though:

> [...] and we do depend on RAII to keep the code correct.

Imagine we have some code like the following:

```
class reference
{
public:
    reference() : L(nullptr) {}

    reference(lua_State* L)
        : L(L)
        , idx(luaL_ref(L, LUA_REGISTRYINDEX))
    {}
```

```
    ~reference()
    {
        if (!L)
            return;

        luaL_unref(L, LUA_REGISTRYINDEX, idx);
    }

    reference(reference&& o)
        : L(o.L)
        , idx(o.idx)
    {
        o.L = nullptr;
    }

    lua_State* state() const
    {
        return L;
    }

    void push() const
    {
        assert(L);
        lua_pushinteger(L, idx);
        lua_gettable(L, LUA_REGISTRYINDEX);
    }

private:
    lua_State* L;
    int idx;
};
```

If an object of this type has its destructor called on `lua_error()`-triggered stack unwinding, it means we're manipulating the `lua_State*` (`luaL_unref(L)` in this example) on stack unwinding (i.e. outside of a lua-catch block which would be just after a `pcall()` return). If the VM is not in a safe state for manipulations at this moment (this scenario just doesn't happen if you stick with plain C which is the target lua was developed for) then we're screwed. Luckily, the VM can handle such situations just fine as it is hinted on the luaJIT documentation:

> ```
> static int wrap_exceptions(lua_State *L, lua_CFunction f)
> {
>   try {
>     return f(L);  // Call wrapped function and return result.
>   } catch (const char *s) {  // Catch and convert exceptions.
>     lua_pushstring(L, s);
>   } catch (std::exception& e) {
>     lua_pushstring(L, e.what());
>   } catch (...) {
> ```

```
    lua_pushliteral(L, "caught (...)");
  }
  return lua_error(L);  // Rethrow as a Lua error.
}
```

— http://luajit.org/ext_c_api.html#mode_wrapcfunc, Recommended usage pattern for `LUAJIT_MODE_WRAPCFUNC`

This guarantee is promised again (although this version of the promise is read-only) in their "extensions" page (and again only at the *full interoperability* category):

> Lua errors can be caught on the C++ side with `catch(…)`. The corresponding Lua error message **can be retrieved from the Lua stack**.
>
> — http://luajit.org/extensions.html#exceptions (emphasis mine)

The final piece for our puzzle is related to async ops converting `std::error_code` into lua exceptions (i.e. `lua_error()`). The completion handler for async ops is not called in a lua context, so they cannot just call `lua_error()` and hope the correct context will catch the exception (there's no API similar to `resume_with()` from Boost.Context). They need to return control to the native code that suspended the fiber so it can throw a lua exception before control returns to lua code.

This guarantee used to exist on luaJIT 1.x (which included Coco):

> Now, if the current coroutine has an associated C stack, `lua_yield()` returns the number of arguments passed back from the resume.
>
> — http://coco.luajit.org/api.html#lua_yield

The lack of allocated C stacks brings more complications to the implementation that will be discussed later. `lua_yieldk()` from Lua 5.2 would be enough for us (and cheaper!), but we don't have that either.

Yet another option would be to set an one-time hook to be called immediately just before resuming the lua coroutine, but it'd present challenges in the future if we ever add debugging support, so it is avoided.

And the solution Emilua get away with is wrapping the C function inside a lua function. The C function returns a 2-tuple. If the first argument is not nil, the lua function itself will take care of use it to raise an error.

```
local error, native = ...
return function(...)
    local e, v = native(...)
    if e then
        error(e)
    else
        return v
    end
```

```
end
```

User-coroutines

Let's jump straight to a topic that gives some sense of continuity to the previous section. The `pcall()` barrier is not the only barrier that the user can insert to prevent `lua_yield()` from suspending the fiber. The user might very well just wrap calls using `coroutine.create()`:

```
function user_fn()
    coroutine.create(function()
        io_obj:emilua_async_op()
    end)
end
```

Rule

Lua's `coroutine` module must never be directly exposed to lua code.

The problem is solved by exposing a different `coroutine` module — a small shim over the original one. This version inspects `this_fiber`'s suspension reason (native code or lua code).

Conceptually, the implementation looks like this:

```
function coroutine.resume(co, ...)
    if _G.busy_coroutines[co] then
        -- CORUN
        error("cannot resume running coroutine", 2)
    end

    local args = {...}
    while true do
        local ret = {raw_coroutine.resume(co, unpack(args))}
        if ret[1] == false then
            return unpack(ret)
        end
        if _G.this_fiber.native_yield then
            _G.busy_coroutines[co] = true
            args = {raw_coroutine.yield(unpack(ret, 2))}
            _G.busy_coroutines[co] = nil
        else
            return unpack(ret)
        end
    end
end

function coroutine.yield(...)
    if _G.fibers[raw_coroutine.running()] ~= nil then
```

```
        error("bad coroutine", 2)
    end
    return raw_coroutine.yield(...)
end

function coroutine.status(co)
    if _G.busy_coroutines[co] then
        return "normal"
    end

    return raw_coroutine.status(co)
end

function coroutine.running()
    local co = raw_coroutine.running()
    if _G.fibers[co] ~= nil then
        -- Fiber's coroutines work just like the main coroutine
        return nil
    end

    return co
end

coroutine.create = ...
coroutine.wrap = ...
```

Dead fibers

When an exception escapes the fiber stack, the hook registered with `sys.set_uncaught_hook()` is called. The default hook prints the stack trace to `stderr` and additionally terminates the VM if the exception escaped from the main fiber. If the custom hook itself fails, the default hook is then called anyway.

Scope handlers are properly popped and called after the hook returns control of the thread to the runtime.

The hook is only called for detached fibers. Therefore, a different behaviour can be chosen for each `join()`ed fiber. Also, if the fiber isn't explicitly `detach()`ed, the hook action will be deferred until some GC round.

There isn't a `pcall` block around the whole program. `lua_resume` is enough and it has the nice property of not unwinding the stack so it can be examined from the error handler. A new lua thread is created to execute the uncaught-hook while it has the chance to examine the unchanged error'ed call stack.

The hook mechanism isn't implemented yet.

Functions that receive a lua callback

There are plenty of functions that have a lua closure as a parameter (e.g. `pcall()`, `scope()`, ...). If we blindly implement them in plain C, they will configure a non-leaf C stack frame which we cannot suspend.

To avoid the C stack frame in the middle of the call-stack altogether, we implement (parts of) these functions in lua, not C. The problem is then how to expose sensitive raw resources that the C functions would use. One of the goals is to not let these resources escape elsewhere.

A quick way to achieve it is by having a lua bootstrap function/chunk to create closures and later change their upvalues through C:

```
local private_resource = ...
return function()
    -- use `private_resource`
end
```

This approach is naive as luaJIT 2.x does not implement some lua functions (i.e. the sensitive raw resources that we want to keep private) as C functions and we cannot feed them as upvalues for the imported bytecode. For instance, we have this behaviour for `pcall()`:

```
lua_pushcfunction(L, luaopen_base);
lua_call(L, 0, 0);
lua_getglobal(L, "pcall");
lua_CFunction pcall_addr = lua_tocfunction(L, -1);
assert(pcall_addr == nullptr); // :-(
```

Therefore the lua bytecode won't be a closure with uninitialized upvalues per se, but a function that receives the private resources and returns the needed closure. It is an extra step on startup, but at least we save some cycles by compiling the bytecode with stripped debug info in the project build stage.

Process environment

A part of the process environment (e.g. UNIX signals) should be under complete control of the program and no external library should meddle with it. However, no protections will be provided to enforce this good practice.

VM settings inheritance

New actors should inherit generic customization points for the GC (e.g. step count and period) and the JIT. They should also inherit allocator settings, but they must **not** be prevented from creating new actors with higher allocation quotas (unless of course the global pool is already at its limit).

Lua 5.2/LuaJIT extensions

We use some C functions found only on Lua 5.2+ and/or LuaJIT:

- `luaL_traceback()`
- `luaopen_bit()`
- `luaopen_jit()`
- `luaopen_ffi()`

There are projects such as Kepler that offer a port of these functions to Lua 5.1.

2GB addressing limit

luaJIT has a serious 2GB limit that has been fixed on forks. By default, the broken 64-bit addressing mode is hidden behind `LUAJIT_ENABLE_GC64`. Emilua might consider moving to moonjit if its author don't try to part away from the lua 5.1 core and keep himself distant from 5.3+ syntactic explosion madness. I **don't** like this C++-like culture expanding to lua or other languages (kudos to Go here for avoiding it).

JIT parameters

The JIT parameters are also changed from the old defaults:

```
maxtrace=1000
maxrecord=4000
maxmcode=512  -- in KB
```

To defaults based on OpenResty findings:

```
maxtrace=8000
maxrecord=16000
maxmcode=40960  -- in KB
```

Locales

A recent POSIX standard specified anemic per-thread and per-function locale support, but, aside from this anemic support, C uses the same locale globally for the whole process.

Meanwhile, C++ has somewhat usable support for multiple locales per process (and an extra global one that also affects the global C locale).

Functions such as `perror()` and `strerror()` will query `LC_MESSAGES` from the global C locale. However the sole function to query this attribute — `setlocale()` — is not thread-safe so we shouldn't change the locale after the program starts and minimal initialization to the process state is done. Changing the global locale is highly unsafe and such API will not be exposed to Lua code.

The thread-safe C++ locales export functionality for `LC_MESSAGES` through the facet `std::messages`. This facet allows one to open system-defined message catalogs, and get translation messages for them. This facet exposes no equivalent for the query `setlocale(LC_MESSAGES, NULL)`. Even if we query it at the beginning of the program and try to attach a new custom facet to the global locale object, this will create a nameless locale. Unnamed global C++ locales will break `LC_MESSAGES` for the C ecosystem (e.g. `perror()` will no longer print localized messages). Therefore custom facets are out of question.

A direct call to `setlocale(LC_MESSAGES, NULL)` is avoided too because ISO C++ doesn't define the macro `LC_MESSAGES`. To query the current `LC_MESSAGES` we just look for `LC_MESSAGES` in the current C++ locale's name. This approach doesn't interfere with the C ecosystem, and also paves the way for multiple per-process locales.

One can find the list of POSIX environment variables that affect the process' locale at https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap08.html#tag_08_02. The format for these variables is defined as:

```
[language[_territory][.codeset][@modifier]]
```

This format is compatible with RDF's Turtle where `LANGTAG` is defined as:

```
LANGTAG ::= '@' [a-zA-Z]+ ('-' [a-zA-Z0-9]+)*
```

And it matches the semantics for BCP47 definition:

```
obs-language-tag = primary-subtag *( "-" subtag )
primary-subtag   = 1*8ALPHA
subtag           = 1*8(ALPHA / DIGIT)
```

The registry of subtags is maintained by IANA at https://www.iana.org/assignments/language-subtag-registry/language-subtag-registry.

So `LC_MESSAGES=pt_BR` becomes Turtle's `"literal"@pt-BR` (and at least the subtag is case sensitive).

A Turtle language-tagged string ceases to be of the datatype http://www.w3.org/2001/XMLSchema#string. Its datatype will be http://www.w3.org/1999/02/22-rdf-syntax-ns#langString. If this is a problem for your application, do not use Turtle language-tagged strings.

For more information about C++ locales, the following links are relevant:

- https://stdcxx.apache.org/doc/stdlibug/24-3.html
- https://gcc.gnu.org/onlinedocs/libstdc++/manual/facets.html#std.localization.facet.messages%23facet.messages.design
- https://www.gnu.org/software/libc/manual/html_node/Locale-Names.html

Open questions

- Describe the behaviour for `sys.exit()` (for main and secondary VMs). Should it call the cancellator for every active operation? Should it exit the application?

Extra caution to take when writing plug-ins

Always keep in mind:

- If you enable your IO object to be sent over channels, it'll also be able to migrate to a different `asio::io_context` and you must take care to keep a work guard to the original `asio::io_context`.
- Pending operations must hold a strong reference to `vm_context` and a work guard — directly or indirectly — to `vm_context.strand()`.
- IO objects (channels included) by themselves must not hold any strong references to their own `vm_context` (this cycle would prevent auto-closing the VM and associated channels). Operation initiation is the perfect time to upgrade *weak* references (if any) to strong ones.
- Pending operations must not trust `L` from the initiating operation to decide which fiber to wake-up later on. They must resort — at initiation time — to the `vm_context` API. Check the simple `sleep_for()` implementation for a code template.

Final note

Emilua software is complex. There should be no pursuit in indefinitely extending this base. Rather, we should search for stabilization and maturity (and also tooling around a solid base).

If you think there should be a nice lua library to handle IRC and what-not, by all means do write it, but write it as a separate lua library (or native plug-in), and compete against the free market of libraries. Do not submit a proposal to integrate it in the core. There are no batteries included. And there shall be no committee-driven development.

Likewise, we should be stuck in the current lua syntax (5.1 plus some extensions found in the beta branch of luaJIT 2.1[3]) forever. If you want more syntax, use a transpiler.

[1] http://lua-users.org/lists/lua-l/2007-10/msg00600.html

[2] Do notice that contrary to the feeling nourished in the mailing list thread, panic functions also would work in our case. I've tested/verified and I also followed the relevant source code for multiple LuaJIT versions. Really, it's okay.

[3] http://luajit.org/extensions.html#lua52 (`-DLUAJIT_ENABLE_LUA52COMPAT`).

Interruption API

Emilua also provides an interruption API that you can use to cancel fibers (you might use it to free resources from fibers stuck in IO requets that might never complete).

The main question that an interruption API needs to answer is how to keep the application in a consistent state. A consistent state is a knowledge that is part of the application and the programmer assumptions, not a knowledge encoded in emilua source code itself. So it is okay to offload some of the responsibility on the application itself.

One dumb'n'quick example that illustrates the problem of a consistent state follows:

```
local m = mutex.new()

local f = spawn(function()
    m:lock()
    sleep(2)
    m:unlock()
end)

sleep(1)
f:interrupt()
m:lock()
```

Before a fiber can be discarded at interruption, it needs to restore state invariants and free resources. The GC would be hopeless in the previous example (and many more) because the mutex is shared and still reachable even if we collect the interrupted fiber's stack. There are other reasons why we can't rely on the GC for the job.

Windows approach to thread cancellation would be a contract. This contract requires the programmer to never call a blocking function directly — always using `WaitForMultipleObjects()`. And another rule: pass a cancellation handle along the call chain for other functions that need to perform blocking calls. Conceptually, this solution is just the same as Go's:

```
select {
case job <- queue:
    // ... do job ...
case <- ctx.Done():
    // goroutine cancelled
}
```

The difference being that Go's `Context` is part of the standard library and a contract everybody adopts. The lesson here is that cancellation is part of the runtime, or else it just doesn't work. In Emilua, the runtime is extended to provide cancellation API inspired by POSIX's thread cancellation.

I've looked many environments, and the only difference I've observed between the

> terms *cancellation* and *interruption* is that interruption is used to convey the property of cancellation being implemented in terms of exceptions.
>
> When I refer to fiber cancellation within the Emilua runtime, I'll stick to the term interruption.

The rest of this document will gloss over many details, but as long as you stay on the common case, you won't need to keep most of these details in mind (sensible defaults) and for the details that you do need to remember, there is a smaller "recap" section at the end.

Do **not** copy-paste code snippets surrounded by **WARNING** blocks. They're most likely to break your program. Do read the manual to the end. These code snippets are there as intermediate steps for the general picture.

The lua exception model

It is easy to find a try-catch construct in mainstream languages like so:

```
try {
    // code that might err
} catch (Exception e) {
    // error handler
}

// other code
```

And here's lua translation of this pattern:

```
local ok = pcall(function()
    -- code that might err
end)
if not ok then
    -- error handler
end
-- other code
```

The main difference here is that lua's exception mechanism doesn't integrate tightly with the type system (and that's okay). So the `catch`-block is always a catch-all really. Also, the structure initially suggests we don't need special syntax for a `finally` block:

```
try {
    // code that might err
} catch (Exception e) {
    // error handler
} finally {
    // cleanup handler
```

```
}

// other code
```

```
local ok = pcall(function()
    -- code that might err
end)
if not ok then
    -- error handler
end
-- cleanup handler
-- other code
```

In sloppy terms, the interruption API just re-schedules the fiber to be resumed but with the fiber stack slightly modified to throw an exception when execution proceeds. This property will trigger stack unwinding to call all the error & cleanup handlers in the reverse order that they were registered.

The interruption protocol

The fiber handle returned by the `spawn()` function is the heart to communicate intent to interrupt a fiber. To better accommodate support for structured concurrency and not introduce avoidable co-dependency between them, we follow the POSIX thread cancellation model (Java's confusing state machine is ignored). Long story short, once a fiber has been interrupted, it cannot be un-interrupted.

To interrupt a fiber, just call the `interrupt()` function from a fiber handle:

```
fib:interrupt()
```

You can only interrupt joinable fibers (but the function is safe to call with any handle at any time).

Afterwards, you can safely `join()` or `detach()` the target fiber:

```
fib:join()

-- ...or
fib:detach()
```

If you don't detach a fiber, the GC will do it for you.

It's that easy. Your fiber doesn't need to know the target fiber's internal state and the target fiber doesn't need to know your fiber' internal state. On the other end, to handle an interruption request is a little trickier.

Handling interruption requests

The key concept required to understand the interruption's flow is the *interruption point*. Understand this, and you'll have learnt how to handle interruption requests.

Definition

An *interruption point* configures a point in your application where it is allowed for the Emilua runtime to stop normal execution flow and raise an exception to trigger stack unwinding if an interruption request from another fiber has been received.

When the possibility of interruption is added to the table, your mental model has to take into account that calls to certain functions *now* might throw an error for no other reason but rewind the stack before freeing the fiber.

The only places that are allowed to serve as interruption points are calls to suspending functions (plus the `pcall()` family and `coroutine.resume()` for reasons soon to be explained).

```
-- this snippet has no interruption points
-- exceptions are never raised here
local i = 0
while true do
    i = i + 1
end
```

The following function doesn't need to worry about leaving the object `self` in an inconsistent state if the fiber gets interrupted. And the reason for this is quite simple: this function doesn't have interruption points (which is usually the case for functions that are purely compute-bound). It won't ever be interrupted in the middle of its work.

```
function mt:new_sample(sample)
    self.mean_ = self.a * sample + (1 - self.a) * self.mean_
    self.f = self.a + (1 - self.a) * self.f
end
```

Functions that suspend the fiber (e.g. IO and functions from the `condition_variable` module) configure interruption points. The function `echo` defined below has interruption points.

```
function echo(sock, buf)
    local nread = sock:read(buf) ①
    sock:write(buf, nread)       ②
end
```

Now take the following code to orchestrate the interaction between two fibers.

```
local child_fib = spawn(function()
    local buf = buffer.new(1024)
    echo(global_sock, buf)
end)

child_fib:interrupt()
```

The mother-fiber doesn't have interruption points, so it executes til the end. The `child_fib` fiber calls `echo()` and `echo()` will in turn act as an interruption point (i.e. the property of being an interruption point propagates up to the caller functions).

`this_fiber.yield()` can be used to introduce interruption points for fibers that otherwise would have none.

The mother-fiber doesn't call any suspending function, so it'll run until the end and only yields execution back to other fibers when it does end. At the last line, an interruption request is sent to the child fiber. The runtime's scheduler doesn't guarantee when the interruption request will be delivered and can schedule execution of the remaining fibers with plenty of freedom given we're not using any synchronization primitives.

In this simple scenario, it's quite likely that the interruption request will be delivered pretty quickly and the call to `sock:read()` inside `echo()` will suspend `child_fib` just to awake it again but with an exception being raised instead of the result being returned. The exception will unwind the whole stack and the fiber finishes.

Any of the interruption points can serve for the fiber to act on the interruption request. Another possible point where these mechanisms would be triggered is the `sock:write()` suspending function.

The uncaught-hook isn't called when the exception is `fiber_interrupted` so you don't really have to care about trapping interruption exceptions. You're free to just let the stack fully unwind.

```
local child_fib = spawn(function()
    local buf = buffer.new(1024)
    global_sock_mutex:lock()
    local ok, ex = pcall(function()
        echo(global_sock, buf)
    end)
    global_sock_mutex:unlock()
    if not ok then
        error(ex)
    end
end)
```

To register a cleanup handler in case the fiber gets interrupted, all you need to do is handle the

raised exceptions.

A fiber is always either interrupted or not interrupted. A fiber doesn't go back to the un-interrupted state. Once the fiber has been interrupted, it'll stay in this state. The task in hand is to rewind the stack calling the cleanup handlers to keep the application state consistent after the GC collect the fiber — all done by the Emilua runtime.

So you can't call more suspending functions after the fiber gets interrupted:

```
local ok, ex = pcall(function()
    -- lots of IO ops                ①
end)
if not ok then
    watchdog_sock:write(errored_msg) ②
    error(ex)
end
```

① Lots of interruption points. All swallowed by `pcall()`.

② If fiber gets interrupted at `#1`, it won't init any IO operation here but instead throw another `fiber_interrupted` exception.

The previous snippet has an error. To properly achieve the desired behaviour, you have to temporally disable interruptions in the cleanup handler like so:

```
local ok, ex = pcall(function()
    -- lots of IO ops
end)
if not ok then
    this_fiber.disable_interruption()
    pcall(function()
        watchdog_sock:write(errored_msg)
    end)
    this_fiber.restore_interruption()
    error(ex)
end
```

`this_fiber.restore_interruption()` has to be called as many times as `this_fiber.disable_interruption()` has been called to restore interruptibility.

It looks messy, but this behaviour actually helps the common case to stay clean. Were not for these choices, a common fiber that doesn't have to handle interruption like the following would accidentally swallow an interruption request and never get collected:

```
local ok = false
while not ok do
    ok = pcall(function()
        my_udp_sock:send(notify_msg)
```

```
    end)
end
```

And the `pcall()` family in itself also configures an interruption point exactly to make sure that loops like this won't prevent the fiber from being properly interrupted. `pcall()` family and `coroutine.resume()` are the only functions which aren't suspending functions but introduce interruption points nevertheless.

It is guaranteed that `fib:interrupt()` will never be an interruption point (and neither a suspension point).

This guarantee is useful to build certain concurrency patterns.

The `scope()` facility

The control flow for the common case is good, but handling interruptions right now is tricky to say the least. To make matters less error-prone, the `scope()` family of functions exist.

- `scope()`
- `scope_cleanup_push()`
- `scope_cleanup_pop()`

The `scope()` function receives a closure and executes it, but it maintains a list of cleanup handlers to be called on the exit path (be it reached by the common exit flow or by a raised exception). When you call it, the list of cleanup handlers is empty, and you can use `scope_cleanup_push()` to register cleanup handlers. They are executed in the reverse order in which they were registered. The handlers are called with the interruptions disabled, so you don't need to disable them yourself.

It is safe to have nested `scope()`s.

One of the previous examples can now be rewritten as follows:

```
local child_fib = spawn(function()
    local buf = buffer.new(1024)
    global_sock_mutex:lock()
    scope_cleanup_push(function() global_sock_mutex:unlock() end)
    echo(global_sock, buf)
end)
```

A hairy situation happens when a cleanup handler itself throws an error. The reason why the default uncaught-hook doesn't terminate the VM when secondary fibers fail is that cleanup handlers are trusted to keep the program invariants. Once a cleanup handler fails we can no longer hold this assumption.

Once a cleanup handler itself throws an error, the VM is terminated[1] (there's no way to recover from this error without context, and conceptually by the time

uncaught hooks are executed, the context was already lost). If you need some sort of protection against one complex module that will fail now and then, run it in a separate actor.

In C++ this scenario is analogous to a destructor throwing an exception when the destructor itself was triggered by an exception-provoked stack unwinding. And the result is the same, `terminate()`.

If you want to call the last registered cleanup handler and pop it from the list, just call `scope_cleanup_pop()`. `scope_cleanup_pop()` receives an optional argument informing whether the cleanup handler must be executed after removed from the list (defaulting to `true`).

```
scope(function()
    scope_cleanup_push(function()
        watchdog_sock:write(errored_msg)
    end)

    -- lots of IO ops

    scope_cleanup_pop(false)
end)
```

Every fiber has an implicit root scope so you don't need to always create one yourself. The standard lua's `pcall()` is also modified to act as a scope which is a lot of convenience for you.

Given `pcall()` is also an interruption point, examples written enclosed in **WARNING** blocks from the previous section had bugs related to maintaining invariants and the `scope()` family is the safest way to register cleanup handlers.

IO objects

It's not unrealistic to share a single IO object among multiple fibers. The following snippets are based (the original code was not lua's) on real-world code:

Fiber ping-sender

```
while true do
    sleep(20)
    write_mutex:lock()
    scope_cleanup_push(function() write_mutex:unlock() end)
    local ok = pcall(function() ws:ping() end)
    if not ok then
        return
    end
    scope_cleanup_pop()
end
```

Fiber consume-subscriptions

```
while true do
    local ok = pcall(function()
        -- `app` may call `write_mutex:lock()`
        app:consume_subscriptions()
    end)
    if not ok then
        return
    end
    -- uses `condition_variable`
    app:wait_on_subscriptions()
end
```

Fiber main

```
local buffer = buffer.new(1024)
while true do
    local ok = pcall(function()
        local nread = ws:read(buffer)
        -- `app` may call `write_mutex:lock()`
        app:on_ws_read(buffer, nread)
    end)
    if not ok then
        break
    end
end

f1:interrupt()
f2:interrupt()
this_fiber.disable_interruption()
f1:join()
f2:join()
```

A fiber will never be interrupted in the *middle* (tricky concept to define) of some IO operation. If a fiber suspended on some IO operation and it was successfully interrupted, it means the operation is not delivered at all and can be tried again later as if it never happened in the first place. The following artificial example illustrates this guarantee (restricting the IO object to a single fiber to keep the code sample small and easy to follow):

```
scope_cleanup_push(function()
    my_sctp_sock:write(checksum.shutdown_msg)
end)
while true do
    sleep(20)
    my_sctp_sock:write(broadcast_msg)
    checksum:update(broadcast_msg)
end
```

If the interruption request arrives when the fiber is suspended at `my_sctp_sock:write()`, the runtime will schedule cancellation of the underlying IO operation and only resume the fiber when the reply for the cancellation request arrives. At this point, if the original IO operation already succeeded, `fiber_interrupted` exception won't be raised so you have a chance to examine the result and the interruption handling will be postponed to the next interruption point.

The `pcall()` family actually provides the same fundamental guarantee. Once it starts executing the argument passed, it won't throw any `fiber_interrupted` exception so you have a chance to examine the result of the executed code. The `pcall()` family only checks for interruption requests before executing the argument.

Some IO objects might use relaxed semantics here to avoid expensive implementations. For instance, HTTP sockets might close the underlying TCP socket if you cancel an IO operation to avoid bookkeeping state.

Refer to their documentation to check when the behaviour uses relaxed semantics. All in all, they should never block indefinitely. That's a guarantee you can rely on. Preferably, they won't use a timeout to react on cancellations either (that would be just bad).

User-level coroutines

Interruptibility is not a property from the coroutine. The coroutine can be created in one fiber, started in a second fiber and resumed in a third one. Interruptibility is a property from the fiber.

```
fibonacci = coroutine.create(function()
    local a, b = 0, 1
    while true do
        a, b = b, a + b
        coroutine.yield(a)
    end
end)
```

`coroutine.resume()` swallows exceptions raised within the coroutine, just like `pcall()`. Therefore, the runtime guarantees `coroutine.resume()` enjoys the same properties found in `pcall()`:

- `coroutine.resume()` is an interruption point.
- `coroutine.resume()` only checks for interruption requests before resuming the coroutine (i.e. the interruption notification is not fully asynchronous).
- Like `pcall()`, `coroutine.create()` will also create a new `scope()` for the closure. However, this scope (and any nested one) is independent from the parent fiber and tied not to the enclosing parent fiber's lexical scopes but to the coroutine lifetime.

We can't guarantee deterministic resumption of zombie coroutines to (re-)deliver interruption

requests (nor should). Therefore, if the GC collects any of your unreachable coroutines with remaining `scope_cleanup_pop()` to be done, it does nothing besides collecting the coroutine stack. You have to prepare your code to cope with this non-guarantee otherwise you most likely will have buggy code.

```
local co = coroutine.create(function()
    m:lock()
    -- this handler will never be called
    scope_cleanup_push(function() m:unlock() end)
    coroutine.yield()
end)

coroutine.resume(co)
```

The safe bet is to just structure the code in a way that there is no need to call `scope_cleanup_push()` within user-created coroutines.

Recap

The fiber handle returned by `spawn()` has an `interrupt()` member-function that can be used to interrupt joinable fibers. The fiber only gets interrupted at interruption points. To preserve invariants your app relies on, register cleanup handlers with `scope_cleanup_push()`.

The relationship between user-created coroutines and interruptions is tricky. Therefore, you should avoid creating (either manually or through some abstraction) cleanup handlers within them.

```
this_fiber.disable_interruption()
local numbers = {8, 42, 38, 111, 2, 39, 1}

local sleeper = spawn(function()
    local children = {}
    scope_cleanup_push(function()
        for _, f in pairs(children) do
            f:interrupt()
        end
    end)
    for _, n in pairs(numbers) do
        children[#children + 1] = spawn(function()
            sleep(n)
            print(n)
        end)
    end
    for _, f in pairs(children) do
        f:join()
    end
end)

local sigwaiter = spawn(function()
```

```
    local sigusr1 = signals.new(signals.SIGUSR1)
    sigusr1:wait()
    sleeper:interrupt()
end)

sleeper:join()
sigwaiter:interrupt()
```

[1] I initially drafted a design to recover on limited scenarios (check git history if you're curious), but then realized it was not only brittle but also unable to handle leaked fiber handles. Worse, it was very sensitive to leak fiber handles. Therefore I dismissed the idea altogether.

Lua 5.1

Emilua is based on LuaJIT which means Lua 5.1 + some Lua 5.2 extensions. However some builtin Lua modules conflict with Emilua modules and thus are not available. This page lists the divergences.

Enabled modules

- Basic library, which includes the coroutine sub-library.
- String.
- Table.
- Math.
- BitOp.
- JIT.
- FFI.

In other words, the following modules are **not** enabled:

- IO.
- OS.
- Package (a replacement which may or may not be a drop-in replacement will be available in the future).
- Debug (it'll be available in a future release).

Modules

Emilua has its own module system. It may look familiar, and indeed it is the intention. Given the fact that other libraries on the wild will have incompatible execution models, compatibility with existing lua libraries is not a concern (although it is most likely to just work for libraries w/o advanced needs).

The module system is highly inspired by the Rust packaging system. The two languages, however, are too different and these differences impact the module system as well. To import a module in dynamic languages such as lua, Python and JavaScript, it is to evaluate/execute source code. Rust doesn't have this constraint and Rust gets just fine with a lot of static analysis. The two languages live in separate worlds. Finally, the module system is also inspired by what Python and NodeJS do.

A module system is meant to isolate pieces of code, symbols and names. One module should not interfere with each other. And a module can have dependencies on other modules to reuse code. So, there is the need for private members and exported members. Lua has all features we need — closures, nested scopes, environments, global scope as a table — to implement a module system easily.

Quick-start

The things you need to know to get started:

- `require()` is a free function receiving a string with the module id and returning the module. Two imports to the same module will only evaluate it once. The result is cached per running VM instance.
- Every file you write is a module.
- Global names will be exported for modules that import your module.
- Modules can also be directories. In this case, a file named `init.lua` will be searched and imported in that directory. `init.lua` can import any other module inside its directory.
- Cyclic references are unsupported and will raise an error on import.
- You can use the syntax `require('../foobar')` to import a sibling module named `foobar`.
- If the module id doesn't start with `'./'` or `'../'` then it is assumed to refer to an external package and different rules apply (see section at the end).

Small example

File `src/init.lua`:

```
local server = require('./server')

local hostname = '127.0.0.1'
local port = 3000

local s = server.new(function(sock, req, res)
```

```
  res.headers = {
    ['content-type'] = 'text/plain'
  }
  res.body = 'Hello World\n'
  sock:write_response(res)
end)

s:listen(hostname, port)
```

File `src/server.lua`:

```
local ip = require('ip')
local http = require('http')

local mt = {}
mt.__index = mt

function new(handler)
  return setmetatable({ handler = handler }, mt)
end

function mt:listen(hostname, port)
  local acceptor = ip.tcp.acceptor.new()
  acceptor:open(ip.address.new(hostname))
  acceptor:bind(hostname, port)
  acceptor:listen()
  spawn(function()
    while true do
      local s = http.socket.new(acceptor:accept())
      spawn(function()
        local req = http.request.new()
        local res = http.response.new()

        while true do
          s:read_request(req)
          res.status = 200
          res.reason = 'OK'
          res.headers = nil
          res.body = nil
          res.trailers = nil
          self.handler(s, req, res)
        end
      end)
    end
  end):detach()
end
```

Big modules

A typical project structure may look as follows:

```
src
├──── init.lua
├──── my_module
│     ├──── error.lua
│     ├──── init.lua
│     ├──── util.lua
│     └──── worker.lua
└──── util.lua
```

In this example, there is the project scope whose root begins at `src/init.lua` — the root module.

In the root module, it is forbidden to use `require('../')` statements as there is no parent module. Any name the `src/init.lua` file `require()`s will be searched on the `src` directory. For instance, if `src/init.lua` contains `require('./util')`, emilua will use the `src/util.lua` file to define the importing module.

But modules may grow and can be further split into files within a directory by itself. That was the case for `my_module`. The `init.lua` file in that directory will be searched for, and, once found, evaluated. If `src/my_module/init.lua` contains more `require()` calls whose arguments start with `'./'`, files within that directory (`src/my_module`) will be searched for.

For instance, if `src/my_module/init.lua` contains `require('./worker')`, the file `src/my_module/worker.lua` will be searched for. Any file (except for `init.lua`) within `src/my_module` can import other files from the same directory (i.e. their siblings) using the `require('../')` form (`src/my_module/init.lua` siblings live in the directory above, `src`). For instance, `src/my_module/worker.lua` and `src/my_module/util.lua` may both want to use the same error type (possibly private) to that module — `src/my_module/error.lua`. In this case, all they need to contain is the call `require('../error')`. And finally due to how they are defined by files (not directories by themselves), they don't have children modules and can't use the usual `require('./')` call (i.e. the call argument must start with `../`).

Any number of super levels is allowed (e.g. `require('../../../../foobar')`).

External packages

If the module name to import doesn't begin with `'./'` nor `'../'` then it'll be searched for outside of the project directory. The places Emilua will look for are:

- Core modules (e.g. `'inbox'`).
- External packages.

Emilua looks for external packages by examining the following locations (in that order):

1. The `EMILUA_PATH` environment variable. That's a colon-separated list[1] of directories.

2. The installation-dependent default (usually `$PREFIX/lib/emilua-$VERSION`).

Misc

You might be interested in restricting the filenames of your modules to the set discovered by Boost developers over the years:

- https://www.boost.org/development/requirements.html#Directory_structure

[1] It's semicolon-separated on Windows.

Errors

Emilua is a concurrency runtime for Lua programs. The intra-VM concurrency support is exploited to offer async I/O. IO errors reported from the operating system are preserved and reported back to the user. That's specially important for logging and tracing.

POSIX systems report errors through `errno`. Meanwhile Windows report errors through `GetLastError()`. In both cases, we have an integer holding an error code. So that's the first piece of information captured and reported.

The enumeration for `errno` cannot be extended by libraries or user code, so each new module that uses the same error reporting style (integer error codes) must defined their own enumeration (which can safely conflict with error code values from `errno`). The origin of the integer code defines the error domain. For instance, POSIX's `getaddrinfo()` uses its own set of error codes (`EAI_`...). The error domain is the second piece of information captured and reported by Emilua: that's the error category.

An error reported by Emilua is a Lua table with two members:

`code: integer`

: The error code (e.g. value from `errno`).

`category: userdata`

: An object that encodes the error domain (e.g. whether value was read out of `errno`).

Extra information about the error's origin might be available depending on the function that throws the error (e.g. many functions attach the integer `"arg"` for `EINVAL` errors).

The error category

Error categories define the metamethods `__tostring()` and `__eq()`. The category for errors read from `errno` (or `GetLastError()` on Windows) will return `"system"` for `__tostring()`. That's the category's name.

Another important category on Emilua is the `"generic"` category. This category is meant to represent POSIX errors (even on Windows). The purpose of this category is to compare errors portably so you can write cross-platform programs, but you'll see more on that later.

`message(self, code: integer) → string`

Returns the explanatory message string for the error specified by `code`.

For the `"system"` category on POSIX platforms, that's the same as strerror(3p).

The error table

The metatable for raised error tables also define the metamethods `__tostring()` and `__eq()`. Its `__tostring()` is just a shorthand to use the category's `message()`. Only `code` and `category` are

compared for `__eq()` and extra members are ignored.

`togeneric(self) → error_code`

That's a function present in `__index`. It'll return the default error condition for `self`.

For instance, `filesystem.create_hard_link()` will report the original error from the OS so you don't lose information on errors. On Windows, this function might throw `ERROR_ALREADY_EXISTS`, but this error maps perfectly to POSIX's `EEXIST`. If you're *reacting* on error codes to determine an action to take (i.e. you're actually handling the error instead of throwing it up higher in the stack or logging/tracing it), then adding the specific error code for each platform serves you no purpose. That's the purpose for the function `togeneric()`. If there's a mapping between the error code and POSIX, it'll return a new error table from the `"generic"` category. If no such mapping exists, the original error is returned.

```
local ok, ec = pcall(...)
if ec:togeneric() == generic_error.EEXIST then
    -- ...
end
```

RDF error categories

Errors are also user-extensible by defining your own error categories. Emilua has the concept of modules defined by RDF's Turtle files[1]. In the future, this will also be used to define application/packaging resources in Android and Windows binaries, for instance. However, right now, they're only used to define error categories.

```
# Easter egg codes from:
# <https://www.gnu.org/software/libc/manual/html_node/Error-Codes.html>

@prefix cat: <https://schema.emilua.org/error_category/0/#>.

<about:emilua-module>
    a <https://schema.emilua.org/error_category/0/>;
    cat:error [
        cat:code 1;
        cat:alias "ED";
        # The experienced user will know what is wrong.
        cat:message "?"
    ], [
        cat:code 2;
        cat:alias "EGREGIOUS";
        # You did what?
        cat:message "You really blew it this time",
            "Você realmente se superou dessa vez"@pt-BR
    ], [
        cat:code 3;
        cat:alias "EIEIO";
```

```
        # Go home and have a glass of warm, dairy-fresh milk.
        cat:message "Computer bought the farm"
    ], [
        cat:code 4;
        cat:alias "EGRATUITOUS";
        # This error code has no purpose.
        cat:message "Gratuitous error"
    ].
```

> [Turtle is] RDF syntax for those with taste
>
> — David Robillard, LV2 co-author

Just throw a `.ttl` file in the place where you'd put your `.lua` file and the module system will find it.

```
local my_error_category = require "/my_error_category"

-- it creates a new error every time,
-- so you don't need to worry about reusing
-- old values
local my_error = my_error_category.EGREGIOUS
my_error.context = "Lorem ipsum"
error(my_error)
```

You can also refer to errors in a category module by number, but that should be avoided:

```
error(my_error_category[2])
```

You can also define a mapping for generic errors:

```
@prefix cat: <https://schema.emilua.org/error_category/0/#>.

<about:emilua-module>
    a <https://schema.emilua.org/error_category/0/>;
    cat:error [
        cat:code 1;
        cat:alias "operation_would_block",
            "resource_unavailable_try_again";
        cat:message "Resource temporarily unavailable";
        cat:generic_error "EAGAIN"
    ].
```

It might be useful to define generic errors for categories other than `"generic"` too[2]. However Emilua doesn't offer this ability yet as someone needs to put some

> thought on the design.

This is an unusual design in the Lua ecosystem, so you might want some rationale: https://blog.emilua.org/2021/03/14/lua-errors-from-multiple-vms/.

[1] https://github.com/JoshData/rdfabout

[2] http://breese.github.io/2017/05/12/customizing-error-codes.html

Sandboxes

Emilua provides support for creating actors in isolated processes using Capsicum, FreeBSD jails, Linux namespaces or Landlock. The idea is to prevent potentially exploitable code from accessing resources beyond what has been explicitly handed to them. That's the basis for capability-based security systems, and it maps pretty well to APIs implementing the actor model such as Emilua.

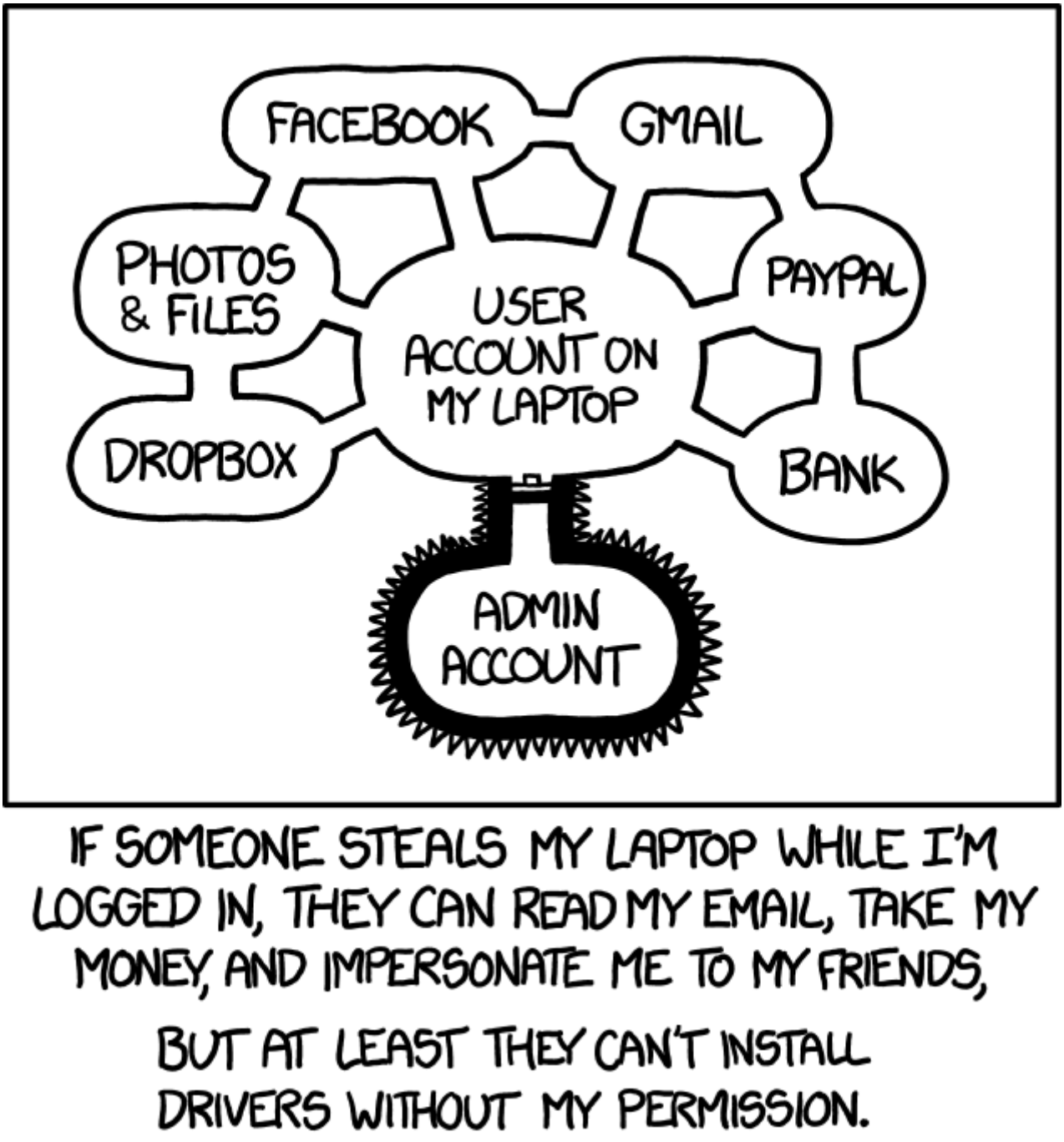


Figure 1. XKCD 1200: Authorization

Even modern operating systems are still somehow rooted in an age where we didn't know how to properly partition computer resources adequately to user needs keeping a design focused on practical and conscious security. Several solutions are stacked together to somehow fill this gap and they usually work for most of the applications, but that's not all of them.

Consider the web browser. There is an active movement that try to push for a future where only

the web browser exists and users will handle all of their communications, store & share their photos, book hotels & tickets, check their medical history, manage their banking accounts, and much more… all without ever leaving the browser. In such scenario, any protection offered by the OS to protect programs from each other is rendered useless! Only a single program exists. If a hacker exploits the right vulnerability, all of the user's data will be stolen. There is no real compartmentalisation.

The browser is part of a special class of programs. The browser is a shell. A shell is any interface that acts as a layer between the user and the world. The web browser is the shell for the www world. Www browser or not, any shell will face similar problems and has to be consciously designed to safely isolate contexts that distrust each other. The Emilua team is not aware of **anything** better than FreeBSD's Capsicum to do just this. In the absence of Capsicum, we have Linux Landlock which can be used to build something close. Browsers actually use Linux namespaces which are older.

The API

> Compartmentalised application development is, of necessity, distributed application development, with software components running in different processes and communicating via message passing.
>
> — Capsicum: practical capabilities for UNIX, Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway

The Emilua's API to spawn an actor lies within the reach of a simple function call:

```
local my_channel = spawn_vm(module)
```

Check the manual elsewhere to understand the details. As for sandboxes, the idea is to spawn an actor where no system resources are available (e.g. the filesystem is mostly empty, no network interfaces are available, no PIDs from other processes can be seen, …).

Consider the hypothetical `sandbox` class:

```
local mysandbox1 = sandbox.new()
local my_channel = spawn_vm(mysandbox1:context(module))
mysandbox1:handshake()
```

That would be the ideal we're pursuing. Nothing other than 2 extra lines of code at most under your application. All complexity for creating sandboxes taken care of by specialized teams of security experts. The Capsicum paper[1] released in 2010 analysed and compared different sandboxing technologies and showed some interesting figures. Consider the following figure that we reproduce here:

Table 4. Sandboxing mechanisms employed by Chromium

Operating system	Model	Line count	Description
Windows	ACLs	22350	Windows ACLs and SIDs
Linux	`chroot`	605	`setuid` root helper sandboxes renderer
Mac OS X	Seatbelt	560	Path-based MAC sandbox
Linux	SELinux	200	Restricted sandbox type enforcement domain
Linux	`seccomp`	11301	`seccomp` and userspace syscall wrapper
FreeBSD	Capsicum	100	Capsicum sandboxing using `cap_enter`

Do notice that line count is not the only metric of interest. The original paper accompanies a very interesting discussion detailing applicability, risks, and levels of security offered by each approach. Just a few years after the paper was released, user namespaces was merged to Linux and yet a new option for sandboxing is now available. Fast-forward a few more years and we also have Linux Landlock which is even better than Linux namespaces. Within this discussion, we can discard most of the approaches — DAC-based, MAC-based, or too intrusive to be even possible to abstract away as a reusable component — as inadequate to our endeavour.

Out of them, Capsicum wins hands down. It's just as capable to isolate parts of an application, but with much less chance to error (for the Chromium patchset, it was just 100 lines of extra C code after all). Unfortunately, Capsicum is not available in every modern OS.

Do keep in mind that this is code written by experts in their own fields, and their salary is nothing less than what Google can afford. 11301 lines of code written by a team of Google engineers for a lifetime project such as Google Chromium is not an investment that any project can afford. That's what the democratization of sandboxing technology needs to do so even small projects can afford them. That's why it's important to use sound models that are easy to analyse such as capability-based security systems. That's why it's important to offer an API that only adds two extra lines of code to your application. That's the only way to democratize access to such technology.

Rust programmers' vision of security is to rewrite the world in Rust, a rather unfeasible undertaking, and a huge waste of resources. In a similar fashion, Deno was released to exploit v8 as the basis for its sandboxing features (now they expect the world to be rewritten in TypeScript). The heart of Emilua's sandboxing relies on technologies that can isolate any code (e.g. C libraries to parse media streams).

Back to our API, the hypothetical `sandbox` class that we showed earlier will have to be some library that abstracts the differences between each sandbox technology in the different platforms. The API that Emilua actually exposes as of this release abstracts all of the semantics related to actor messaging, work/lifetime accounting, process reaping, DoS protection, serialization, lots of Linux namespaces details (e.g. PID1), and much more, but it still expects you to actually initialize the sandbox.

The `init.script`

Every process carries associated credentials that enable operation on system-wide addressable objects such as filesystem objects and sockets. We setup a sandbox by disabling the ambient authority so the address space itself becomes inaccessible. Sandboxed code thus should be run only after such setup already completed successfully. The proper hook to perform this setup is `init.script`. `init.script` runs right after the process is created.

After the sandboxed actor is up it can receive access to new resources through its inbox. If any security exploit is performed on the sandboxed code, then only the objects it has access to are rendered vulnerable (the damage is thus contained in its compartment).

Landlock (Linux)

```
local init_script = [[
    local rules = C.landlock_create_ruleset{ handled_access_fs = {
        "execute", "write_file" "read_file", "read_dir", "remove_dir",
        "remove_file", "make_char", "make_dir", "make_reg", "make_sock",
        "make_fifo", "make_block", "make_sym", "refer", "truncate" } }
    set_no_new_privs()
    C.landlock_restrict_self(rules)
]]

spawn_vm{
    subprocess = {
        init = { script = init_script }
    }
}
```

Landlock as of now can only control access to filesystem objects, but future versions will be more complete.

Capsicum

```
spawn_vm{
    subprocess = {
        init = { script = "C.cap_enter()" }
    }
}
```

Implementation details

The purpose of this section is to help you attack the system. If you're trying to find security holes, this section should be a good overview on how the whole system works.

If you find any bug in the code, please responsibly send a bug report so the Emilua team can fix it.

Message serialization

Emilua follows the advice from WireGuard developers to avoid parsing bugs by avoiding object serialization altogether. Sequenced-packet sockets with builtin framing are used so we always receive/send whole messages in one API call.

There is a hard-limit (configurable at build time) on the maximum number of members you can send per message. This limit would need to exist anyway to avoid DoS from bad clients.

Another limitation is that no nesting is allowed. You can either send a single non-nil value or a non-empty dictionary where every member in it is a leaf from the root tree. The messaging API is part of the attack surface that bad clients can exploit. We cannot afford a single bug here, so the code must be simple. By forbidding subtrees we can ignore recursion complexities and simplify the code a lot.

The struct used to receive messages follows:

```
enum kind
{
    boolean_true    = 1,
    boolean_false   = 2,
    string          = 3,
    file_descriptor = 4,
    actor_address   = 5,
    nil             = 6
};

struct ipc_actor_message
{
    union
    {
        double as_double;
        uint64_t as_int;
    } members[EMILUA_CONFIG_IPC_ACTOR_MESSAGE_MAX_MEMBERS_NUMBER];
    unsigned char strbuf[
        EMILUA_CONFIG_IPC_ACTOR_MESSAGE_MAX_MEMBERS_NUMBER * 512];
};
```

A variant class is needed to send the messages. Given a variant is needed anyway, we just adopt NaN-tagging for its implementation as that will make the struct members packed together and no memory from the host process hidden among paddings will leak to the containers.

The code assumes that no signaling NaNs are ever produced by the Lua VM to simplify the NaN-tagging scheme[2][3]. The type is stored in the mantissa bits of a signaling NaN.

If the first member is nil, then we have a non-dictionary value stored in `members[1]`. Otherwise, a `nil` will act as a sentinel to the end of the dictionary. No sentinel will exist when the dictionary is fully filled.

`read()` calls will write to objects of this type directly (i.e. no intermediate `char[N]` buffer is used) so we avoid any complexity with code related to alignment adjustments.

`memset(buf, 0, s)` is used to clear any unused member of the struct before a call to `write()` so we avoid leaking memory from the process to any container.

Strings are preceded by a single byte that contains the size of the string that follows. Therefore, strings are limited to 255 characters. Following from this scheme, a buffer sufficiently large to hold the largest message is declared to avoid any buffer overflow. However, we still perform bounds checking to make sure no uninitialized data from the code stack is propagated back to Lua code to avoid leaking any memory. The bounds checking function in the code has a simple implementation that doesn't make the code much more complex and it's easy to follow.

To send file descriptors over, `SCM_RIGHTS` is used. There are a lot of quirks involved with `SCM_RIGHTS` (e.g. extra file descriptors could be stuffed into the buffer even if you didn't expect them). The encoding scheme for the network buffer is far simpler to use than `SCM_RIGHTS`' ancillary data. Complexity-wise, there's far greater chance to introduce a bug in code related to `SCM_RIGHTS` than a bug in the code that parses the network buffer.

Code could be simpler if we only supported messaging strings over, but that would just defer the problem of secure serialization on the user's back. Code should be simple, but not simpler. By throwing all complexity on the user's back, the implementation would offer no security. At least we centralized the sensitive object serialization so only one block of code need to be reviewed and audited.

Spawning a new process

UNIX systems allow the userspace to spawn new processes by a `fork()` followed by an `exec()`. `exec()` really means an executable will be available in the container, but this assumption doesn't play nice with our idea of spawning new actors in an empty container.

What we really want is to to perform a fork followed by **no** `exec()` call. This approach in itself also has its own problems. `exec()` is the only call that will flush the address space of the running process. If we don't `exec()` then the new process that was supposed to run untrusted code with no access to system resources will be able to read all previous memory — memory that will most likely contain sensitive information that we didn't want leaked. Other problems such as threads (supported by the Emilua runtime) would also hinder our ability to use `fork()` without `exec()`ing.

One simple approach to solve all these problems is to `fork()` at the beginning of the program so we `fork()` before any sensitive information is loaded in the process' memory. Forking at a well known point also brings other benefits. We know that no thread has been created yet, so resources such as locks and the global memory allocator stay in a well defined state. By creating this extra process before much more extra virtual memory or file descriptor slots in our process table have been requested, we also make sure that further processes creation will be cheaper.

```
└── emilua program
    └── emilua runtime (supervisor fork()ed near main())
```

Every time the main process wants to create an actor in a new process, it'll defer its job onto the supervisor that was `fork()`ed near `main()`. An `AF_UNIX`+`SOCK_SEQPACKET` socket is used to orchestrate this process. Given the supervisor is only used to create new processes, it can use blocking APIs that will simplify the code a lot. The blocking `read()` on the socket also means that it won't be draining any CPU resources when it's not needed. Also important is the threat model here. The main process is not trying to attack the supervisor process. The supervisor is also trusted and it doesn't need to run inside a container. `SCM_RIGHTS` handling between the main process and the supervisor is simplified a lot due to these constraints.

However some care is still needed to setup the supervisor. Each actor will initially be an exact copy of the supervisor process memory and we want to make sure that no sensitive data is leaked there. The first thing we do right after creating the supervisor is collecting any sensitive information that might still exist in the main process (e.g. `argv` and `envp`) and instructing the supervisor process to `explicit_bzero()` them. This compromise is not as good as `exec()` would offer, but it's the best we can do while we limit ourselves to reasonably portable C code with few assumptions about dynamic/static linkage against system libraries, and other settings from the host environment.

This problem doesn't end here. Now that we assume the process memory from the supervisor contains **no** sensitive data, we want to keep it that way. It may be true that every container is assumed as a container that some hacker already took over (that's why we're isolating them, after all), but one container shouldn't leak information to another one. In other words, we don't even want to load sensitive information regarding the setup of any container from the supervisor process as that could leak into future containers. The solution here is to serialize such information (e.g. the `init.script`) such that it is only sent directly to the final process. Another `AF_UNIX` +`SOCK_SEQPACKET` socket is used.

Now to the assumptions on the container process. We do assume that it'll run code that is potentially dangerous and some hacker might own the container at some point. However the initial setup does **not** run arbitrary dangerous code and it still is part of the trusted computing base. The problem is that we don't know whether the `init.script` will need to load sensitive information at any point to perform its job. That's why we setup the Lua VM that runs `init.script` to use a custom allocator that will `explicit_bzero()` all allocated memory at the end. Allocations done by external libraries such as libcap lie outside of our control, but they rarely matter anyway.

That's mostly the bulk of our problems and how we handle them. Other problems are summarized in the short list below.

- `SIGCHLD` would be sent to the main process, but we cannot install arbitrary signal handlers in the main process as that's a property from the application (i.e. signal handling disposition is not a resource owned by the Emilua runtime). The problem was already solved by making the actor a child of the supervisor process.
- We can't install arbitrary signal handlers in the container process either as that would break every module by bringing different semantics depending on the context where it runs (host/container). To handle PID1 automatically we just fork a new process and forward its signals to the new child.
- `"/proc/self/exe"` is a resource inherited from the main process (i.e. a resource that exists outside the container, so the container is not existing in a completely empty world), and could be exploited in the container. `ETXTBSY` will hinder the ability from the container to meddle with

`"/proc/self/exe"`, and `ETXTBSY` is guaranteed by the existence of the supervisor process (even if the main process exits, the supervisor will stay alive).

The output from tools such as `top` start to become rather cool when you play with nested containers:

```
└── emilua program
   └── emilua runtime (supervisor fork()ed near main())
      ├── emilua runtime (PID1 within the new namespace)
      │   └── emilua program
      │      └── emilua runtime (supervisor fork()ed near main())
      └── emilua runtime (PID1 within the new namespace)
         └── emilua program
            └── emilua runtime (supervisor fork()ed near main())
```

Work lifetime management

For Linux namespaces, PID1 eases our life a lot. As soon as any container starts to act suspiciously we can safely kill the whole subtree of processes by sending `SIGKILL` to the PID1 that started it.

For FreeBSD's Capsicum, `PD_DAEMON` is not permitted in subprocesses that were placed into capability mode. If all references to a procdesc file descriptor are closed, the associated process will be automatically terminated by the kernel.

`AF_UNIX`+`SOCK_SEQPACKET` sockets are connection-oriented and simplify our work even further. We `shutdown()` the ends of each pair such that they'll act unidirectionally just like pipes. When all copies of one end die, the operation on the other end will abort. The actor API translates to MPSC channels, so we never ever send the reading end to any container (we only make copies of the sending end). The kernel will take care of any tricky reference counting necessary (and `SIGKILL`ing PID1 will make sure no unwanted end survives).

The only work left for us to do is pretty much to just orchestrate the internal concurrency architecture of the runtime (e.g. watch out for blocking reads). Given that we want to abort reads when all the copies of the sending end are destroyed, we don't keep any copy to the sending end in our own process. Everytime we need to send our address over, we create a new pair of sockets to send the newly created sending end over. `inbox` will unify the receipt of messages coming from any of these sockets. You can think of each newly created socket as a new capability. If one capability is revoked, others remain unaffected.

One good actor could send our address further to a bad actor, and there is no way to revoke access to the bad actor without also revoking access to the good actor, but that is in line with capability-based security systems. Access rights are transitive. In fact, a bad actor could write 0-sized messages over the `AF_UNIX`+`SOCK_SEQPACKET` socket to trick us into thinking the channel was already closed. We'll happily close the channel and there is no problem here. The system can happily recover later on (and only this capability is revoked anyway).

Flow control

The runtime doesn't schedule any read on the socket unless the user calls `inbox:receive()`. Upon reading a new message the runtime will either wake the receiving fiber directly, or enqueue the result in a buffer if no receiving fiber exists at the time (this can happen if the user interrupted the fiber, or another result arrived and woke the fiber up already). `inbox:receive()` won't schedule any read on the socket if there's some result already enqueued in the buffer.

`setns(fd, CLONE_NEWPID)`

We don't offer any helper to spawn a program (i.e. `system.spawn()`) within an existing PID namespace. That's intentional (although one could still do it through `init.script`). `setns(fd, CLONE_NEWPID)` is dangerous. Only `exec()` will flush the address space for the process. The window of time that exists until `exec()` is called means that any memory from the previous process could be read by a compromised container (cf. ptrace(2)).

Tests

A mix of approaches is used to test the implementation.

There's an unit test for every class of good inputs. There are unit tests for accidental bad inputs that one might try to perform through the Lua API. The unit tests always try to create one scenario for buffered messages and another for immediate delivery of the result.

When support for plugins is enabled, fuzz tests are built as well. The fuzzers are generation-based. One fuzzer will generate good input and test if the program will accept all of them. Another fuzzer will mutate a good input into a bad one (e.g. truncate the message size to attempt a buffer overflow), and check if the program rejects all of them.

There are some other tests as well (e.g. ensure no padding exists between the members of the C struct we send over the wire).

[1] https://www.cl.cam.ac.uk/research/security/capsicum/papers/2010usenix-security-capsicum-website.pdf

[2] http://www.lua.org/source/5.2/lapi.c.html#lua_pushnumber

[3] https://github.com/LuaJIT/LuaJIT/blob/v2.0.5/src/lj_api.c#L569

Linux namespaces

Here we show a few recipes on how to deal with Linux namespaces from Emilua.

LWN.net has a good overview on Linux namespaces.

The user namespace

Unless you execute the process as root, Linux will deny the creation of all namespaces except for the user namespace. The user namespace is the only namespace that an unprivileged process can create. However it's fine to pair the user namespace with any combination of the other ones.

When a user namespace is created, it starts out without a mapping of user IDs and group IDs to the parent user namespace. One can fill the mapping directly as shown in the example that follows:

```
local init_script = [[
    local uidmap = C.open('/proc/self/uid_map', C.O_WRONLY)
    send_with_fd(arg, '.', uidmap)
    C.write(C.open('/proc/self/setgroups', C.O_WRONLY), 'deny')
    local gidmap = C.open('/proc/self/gid_map', C.O_WRONLY)
    send_with_fd(arg, '.', gidmap)

    -- sync point
    C.read(arg, 1)
]]

local shost, sguest = unix.seqpacket_socket.pair()
sguest = sguest:release()

spawn_vm{
    subprocess = {
        newns_user = true,
        init = { script = init_script, arg = sguest }
    }
}
sguest:close()
local ignored_buf = byte_span.new(1)

local uidmap = ({system.getresuid()})[2]
uidmap = byte_span.append('0 ', tostring(uidmap), ' 1\n')
local uidmapfd = ({shost:receive_with_fds(ignored_buf, 1)})[2][1]
file.stream.new(uidmapfd):write_some(uidmap)

local gidmap = ({system.getresgid()})[2]
gidmap = byte_span.append('0 ', tostring(gidmap), ' 1\n')
local gidmapfd = ({shost:receive_with_fds(ignored_buf, 1)})[2][1]
file.stream.new(gidmapfd):write_some(gidmap)
```

```
-- sync point #1
shost:send(ignored_buf)

shost:close()
```

An `AF_UNIX+SOCK_SEQPACKET` socket is used to coordinate the parent and the child processes. This type of socket allows duplex communication between two parties with builtin framing for messages, disconnection detection (process reference counting if you will), and it also allows sending file descriptors back-and-forth.

We also close `sguest` from the host side as soon as we're done with it. This will ensure any operation on `shost` will fail if the child process aborts for any reason (i.e. no deadlocks happen here).

Even if it's a sandbox, and root inside the sandbox doesn't mean root outside it, maybe you still want to drop all root privileges at the end of the `subprocess.init.script`:

```
C.cap_set_proc('=')
```

It won't be particularly useful for most people, but that technique is still useful to — for instance — create alternative LXC/FlatPak front-ends to run a few programs (if the program can't update its own binary files, new possibilities for sandboxing practice open up).

Alternatively, one can fill the mapping indirectly. Below we show how to do it using the suid-helper `newuidmap`:

```
local init_script = [[
    local pidfd = C.open('/proc/self', C.O_RDONLY)
    send_with_fd(arg, '.', pidfd)

    -- sync point
    C.read(arg, 1)
]]

local shost, sguest = unix.seqpacket_socket.pair()
sguest = sguest:release()

spawn_vm{
    subprocess = {
        newns_user = true,
        init = { script = init_script, arg = sguest }
    }
}
sguest:close()
local ignored_buf = byte_span.new(1)
local pidfd = ({shost:receive_with_fds(ignored_buf, 1)})[2][1]
```

```
system.spawn{
    program = 'newuidmap',
    stdout = 'share',
    stderr = 'share',
    arguments = {
        'newuidmap',
        'fd:3', '0', '100000', '1001'
    },
    extra_fds = {
        [3] = pidfd
    }
}:wait()

system.spawn{
    program = 'newgidmap',
    stdout = 'share',
    stderr = 'share',
    arguments = {
        'newgidmap',
        'fd:3', '0', '100000', '1001'
    },
    extra_fds = {
        [3] = pidfd
    }
}:wait()

-- sync point #1
shost:send(ignored_buf)

shost:close()
```

You need to configure `/etc/subuid` to have `newuidmap` working.

The network namespace

Let's start by isolating the network resources as that's the easiest one:

```
spawn_vm{ subprocess = {
    newns_user = true,
    newns_net = true
} }
```

The process will be created within a new network namespace where no interfaces besides the loopback device exist. And even the loopback device will be down! If you want to configure the loopback device so the process can at least bind sockets to it you can use the program `ip`. However the program `ip` needs to run within the new namespace. To spawn the program `ip` within the namespace of the new actor you need to acquire the file descriptors to its namespaces. There are two ways to do that. You can either use race-prone PID primitives (easy), or you can use a

handshake protocol to ensure that there are no races related to PID dances. Below we show the race-free method.

```
local init_script = [[
    local userns = C.open('/proc/self/ns/user', C.O_RDONLY)
    send_with_fd(arg, '.', userns)
    local netns = C.open('/proc/self/ns/net', C.O_RDONLY)
    send_with_fd(arg, '.', netns)

    -- sync point
    C.read(arg, 1)
]]

local shost, sguest = unix.seqpacket_socket.pair()
sguest = sguest:release()

spawn_vm{
    subprocess = {
        newns_user = true,
        newns_net = true,
        init = { script = init_script, arg = sguest }
    }
}
sguest:close()
local ignored_buf = byte_span.new(1)
local userns = ({shost:receive_with_fds(ignored_buf, 1)})[2][1]
local netns = ({shost:receive_with_fds(ignored_buf, 1)})[2][1]
system.spawn{
    program = 'ip',
    arguments = {'ip', 'link', 'set', 'dev', 'lo', 'up'},
    nsenter_user = userns,
    nsenter_net = netns
}:wait()
shost:close()
```

The PID namespace

When a new PID namespace is created, the process inside the new namespace ceases to see processes from the parent namespace. Your process still can see new processes created in the child's namespace, so invisibility only happens in one direction. PID namespaces are hierarchically nested in parent-child relationships.

The first process in a PID namespace is PID1 within that namespace. PID1 has a few special responsibilities. After `subprocess.init.script` exits, the Emilua runtime will fork if it's running as PID1. This new child will assume the role of starting your module (the Lua VM).

The controlling terminal

If you want to set up a pty in `init.script`, the PID1 will be the session leader. That

> way, the actor running in PID2 wouldn't accidentally acquire a new ctty if it happens to `open()` a tty that isn't currently controlling any session.

If the PID1 dies, all processes from that namespace (including further descendant PID namespaces) will be killed. This behavior allows you to fully dispose of a container when no longer needed by sending `SIGKILL` to PID1. No process will escape.

Communication topology may be arbitrarily defined as per the actor model, but the processes always assume a topology of a tree (supervision trees), and no PID namespace ever "re-parents".

The Emilua runtime automatically sends `SIGKILL` to every process spawned using the Linux namespaces API when the actor that spawned them exits. If you want fine control over these processes, you can use a few extra methods that are available to the channel object that represents them.

The mount namespace

Let's build up on our previous knowledge and build a sandbox with an empty `"/"` (that's right!).

```
local init_script = [[
    ...

    -- unshare propagation events
    C.mount(nil, '/', nil, C.MS_PRIVATE)

    C.umask(0)
    C.mount(nil, '/mnt', 'tmpfs', 0)
    C.mkdir('/mnt/proc', mode(7, 5, 5))
    C.mount(nil, '/mnt/proc', 'proc', 0)
    C.mkdir('/mnt/tmp', mode(7, 7, 7))

    -- pivot root
    C.mkdir('/mnt/mnt', mode(7, 5, 5))
    C.chdir('/mnt')
    C.pivot_root('.', '/mnt/mnt')
    C.chroot('.')
    C.umount2('/mnt', C.MNT_DETACH)

    -- sync point
    C.read(arg, 1)
]]

spawn_vm{
    subprocess = {
        ...,
        newns_mount = true,

        -- let's go ahead and create a new
        -- PID namespace as well
```

```
        newns_pid = true
    }
}
```

We could certainly create a better initial `"/"`. We could certainly do away with a few of the lines by cleverly reordering them. However the example is still nice to just illustrate a few of the syscalls exposed to the Lua script. There's nothing particularly hard about mount namespaces. We just call a few syscalls, and no fd-dance between host and guest is really necessary.

One technique that we should mention is how `module` in `spawn_vm(module)` is interpreted when you use Linux namespaces. This argument no longer means an actual module when namespaces are involved. It'll just be passed along to the new process. The following snippet shows you how to actually get the new actor in the container by finding a proper module to start.

```
local guest_code = [[
    local inbox = require 'inbox'
    local ip = require 'ip'

    local ch = inbox:receive().dest
    ch:send(ip.host_name())
]]

local init_script = [[
    ...

    local modulefd = C.open(
        '/app.lua',
        bit.bor(C.O_WRONLY, C.O_CREAT),
        mode(6, 0, 0))
    send_with_fd(arg, '.', modulefd)
]]

local my_channel = spawn_vm{ module = '/app.lua', ... }

...

local module = ({shost:receive_with_fds(ignored_buf, 1)})[2][1]
module = file.stream.new(module)
stream.write_all(module, guest_code)
shost:close()

my_channel:send{ dest = inbox }
print(inbox:receive())
```

Full example

```
local stream = require 'stream'
```

```
local system = require 'system'
local inbox = require 'inbox'
local file = require 'file'
local unix = require 'unix'

local guest_code = [[
    local inbox = require 'inbox'
    local ip = require 'ip'

    local ch = inbox:receive().dest
    ch:send(ip.host_name())
]]

local init_script = [[
    local uidmap = C.open('/proc/self/uid_map', C.O_WRONLY)
    send_with_fd(arg, '.', uidmap)
    C.write(C.open('/proc/self/setgroups', C.O_WRONLY), 'deny')
    local gidmap = C.open('/proc/self/gid_map', C.O_WRONLY)
    send_with_fd(arg, '.', gidmap)

    -- sync point #1 as tmpfs will fail on mkdir()
    -- with EOVERFLOW if no UID/GID mapping exists
    -- https://bugzilla.kernel.org/show_bug.cgi?id=183461
    C.read(arg, 1)

    local userns = C.open('/proc/self/ns/user', C.O_RDONLY)
    send_with_fd(arg, '.', userns)
    local netns = C.open('/proc/self/ns/net', C.O_RDONLY)
    send_with_fd(arg, '.', netns)

    -- unshare propagation events
    C.mount(nil, '/', nil, C.MS_PRIVATE)

    C.umask(0)
    C.mount(nil, '/mnt', 'tmpfs', 0)
    C.mkdir('/mnt/proc', mode(7, 5, 5))
    C.mount(nil, '/mnt/proc', 'proc', 0)
    C.mkdir('/mnt/tmp', mode(7, 7, 7))

    -- pivot root
    C.mkdir('/mnt/mnt', mode(7, 5, 5))
    C.chdir('/mnt')
    C.pivot_root('.', '/mnt/mnt')
    C.chroot('.')
    C.umount2('/mnt', C.MNT_DETACH)

    local modulefd = C.open(
        '/app.lua',
        bit.bor(C.O_WRONLY, C.O_CREAT),
        mode(6, 0, 0))
    send_with_fd(arg, '.', modulefd)
```

```
    -- sync point #2 as we must await for
    --
    -- * loopback net device
    -- * `/app.lua`
    --
    -- before we run the guest
    C.read(arg, 1)

    C.sethostname('mycoolhostname')
    C.setdomainname('mycooldomainname')

    -- drop all root privileges
    C.cap_set_proc('=')
]]

local shost, sguest = unix.seqpacket_socket.pair()
sguest = sguest:release()

local my_channel = spawn_vm{
    module = '/app.lua',
    subprocess = {
        newns_user = true,
        newns_net = true,
        newns_mount = true,
        newns_pid = true,
        newns_uts = true,
        newns_ipc = true,
        init = { script = init_script, arg = sguest }
    }
}
sguest:close()

local ignored_buf = byte_span.new(1)

local uidmap = ({system.getresuid()})[2]
uidmap = byte_span.append('0 ', tostring(uidmap), ' 1\n')
local uidmapfd = ({shost:receive_with_fds(ignored_buf, 1)})[2][1]
file.stream.new(uidmapfd):write_some(uidmap)

local gidmap = ({system.getresgid()})[2]
gidmap = byte_span.append('0 ', tostring(gidmap), ' 1\n')
local gidmapfd = ({shost:receive_with_fds(ignored_buf, 1)})[2][1]
file.stream.new(gidmapfd):write_some(gidmap)

-- sync point #1
shost:send(ignored_buf)

local userns = ({shost:receive_with_fds(ignored_buf, 1)})[2][1]
local netns = ({shost:receive_with_fds(ignored_buf, 1)})[2][1]
system.spawn{
```

```
    program = 'ip',
    arguments = {'ip', 'link', 'set', 'dev', 'lo', 'up'},
    nsenter_user = userns,
    nsenter_net = netns
}:wait()

local module = ({shost:receive_with_fds(ignored_buf, 1)})[2][1]
module = file.stream.new(module)
stream.write_all(module, guest_code)

-- sync point #2
shost:close()

my_channel:send{ dest = inbox }
print(inbox:receive())
```

C++ embedder API

If you want to embed Emilua in your own Boost.Asio-based programs, this is the list of steps you need to do:

1. Compile and link against Emilua (use Meson or pkg-config to have the appropriate compiler flags filled automatically).
2. `#include <emilua/state.hpp>`
3. Instantiate `emilua::app_context`. This object needs to stay alive for as long as at least one Lua VM is alive. If you want to be sure, just make sure it outlives `boost::asio::io_context` and you're good to go.
4. Create an `emilua::properties_service` object with the same concurrency hint passed to `boost::asio::io_context` and add it to the `boost::asio::io_context` object using `boost::asio::make_service`. This step will be needed for as long as Boost.Asio refuses to add a getter for the concurrency hint: https://github.com/chriskohlhoff/asio/pull/1254.
5. Call `emilua::make_vm()` (see `src/main.ypp` for an example).
6. Call `emilua::vm_context::fiber_resume()` inside the strand returned by `emilua::vm_context::strand()` to start the Lua VM created in the previous step (see `src/main.ypp` for an example).
7. Optionally synchronize against other threads before you exit the application. If you're going to spawn actors in foreign `boost::asio::io_context` objects in your Lua programs then it's a good idea to include this step. See below.

Emilua is not designed to work properly with `boost::asio::io_context::stop()`. Many cleanup steps will be missed if you call this function. If you need to use it, then spawn Emilua programs in their own `boost::asio::io_context` instances.

`emilua::app_context`

This type stores process-wide info that is shared among all Lua VMs (e.g. process arguments, environment, module paths, module caches, default logger, which VM is the master VM, ...).

If you want to embed the Lua programs in your binary as well you should pre-populate the module cache here with the contents of all Lua files you intend to ship in the binary. `modules_cache_registry` is the member you're looking for. Do this before you start the first Lua VM.

Master VM

If you want to allow your Lua programs to change process state that is shared among all program threads (e.g. current working directory, signal handlers, ...) then you need to elect one Lua VM to be the master VM.

The 1-one snippet that follows is enough to finish this setup. This step must be done before you call `fiber_resume()`.

```
appctx.master_vm = vm_ctx;
```

Cleanup at exit

First make sure `emilua::app_context` outlives `boost::asio::io_context`.

After `boost::asio::io_context::run()` returns you can use the following snippet to synchronize against extra threads and `boost::asio::io_context` objects your Lua scripts created[1].

```
{
    std::unique_lock<std::mutex> lk{appctx.extra_threads_count_mtx};
    while (appctx.extra_threads_count > 0)
        appctx.extra_threads_count_empty_cond.wait(lk);
}
```

Actors spawned in different processes

Emilua has the ability to spawn Lua VMs in their own processes for isolation or sandboxing purposes. To enable this feature, a supervisor process must be created while the program is still single-threaded.

For communication with the supervisor process, Emilua uses an UNIX socket. The file descriptor for this process is stored in `app_context::ipc_actor_service_sockfd`. See `src/main.ypp` for an example on how to initialize this variable.

You also need to initialize `emilua::app_context::environp`. This is a pointer to `environ`. This step must be done before you fork to create the supervisor process. Emilua uses an indirection instead of using `environ` directly because FreeBSD is not POSIX-compliant and the usual declaration for `environ` doesn't work from shared libraries. Emilua never modifies the environment for the main process, so you don't need to worry about what Emilua uses this variable for. Its usage is internal to Emilua and won't affect your C++ program.

```
#if BOOST_OS_UNIX
emilua::app_context::environp = &environ;
#endif // BOOST_OS_UNIX
```

On Linux, you also need to initialize `emilua::clone_stack_address`.

If you don't intend to have Lua VMs tied to their own processes triggered by Lua programs then you can skip this step.

[1] Emilua only instantiates new threads and `boost::asio::io_context` objects if your Lua programs explicitly ask for that when it calls `spawn_vm()`. You can also disable this feature altogether at build time.

Reference

generic_error

```
local generic_error = require 'generic_error'

local my_error = generic_error.EINVAL
my_error.arg = 1
error(my_error)
```

An userdata for which the `__index()` metamethod returns a new error code from the generic category on access.

Error list

- EAFNOSUPPORT
- EADDRINUSE
- EADDRNOTAVAIL
- EISCONN
- E2BIG
- EDOM
- EFAULT
- EBADF
- EBADMSG
- EPIPE
- ECONNABORTED
- EALREADY
- ECONNREFUSED
- ECONNRESET
- EXDEV
- EDESTADDRREQ
- EBUSY
- ENOTEMPTY
- ENOEXEC
- EEXIST
- EFBIG
- ENAMETOOLONG
- ENOSYS
- EHOSTUNREACH

- EIDRM
- EILSEQ
- ENOTTY
- EINTR
- EINVAL
- ESPIPE
- EIO
- EISDIR
- EMSGSIZE
- ENETDOWN
- ENETRESET
- ENETUNREACH
- ENOBUFS
- ECHILD
- ENOLINK
- ENOLCK
- ENOMSG
- ENOPROTOOPT
- ENOSPC
- ENXIO
- ENODEV
- ENOENT
- ESRCH
- ENOTDIR
- ENOTSOCK
- ENOTCONN
- ENOMEM
- ENOTSUP
- ECANCELED
- EINPROGRESS
- EPERM
- EOPNOTSUPP
- EWOULDBLOCK
- EOWNERDEAD
- EACCES

- EPROTO
- EPROTONOSUPPORT
- EROFS
- EDEADLK
- EAGAIN
- ERANGE
- ENOTRECOVERABLE
- ETXTBSY
- ETIMEDOUT
- ENFILE
- EMFILE
- EMLINK
- ELOOP
- EOVERFLOW
- EPROTOTYPE

asio_error

```lua
local asio_error = require 'asio_error'

error(asio_error.misc.eof)
```

An userdata for which the `__index()` metamethod returns a new error code from the asio category on access.

Error list

Basic errors

These errors may be just an alias to common errors from the system category depending on the host operating system.

- `basic.access_denied`
- `basic.address_family_not_supported`
- `basic.address_in_use`
- `basic.already_connected`
- `basic.already_started`
- `basic.broken_pipe`
- `basic.connection_aborted`
- `basic.connection_refused`
- `basic.connection_reset`
- `basic.bad_descriptor`
- `basic.fault`
- `basic.host_unreachable`
- `basic.in_progress`
- `basic.interrupted`
- `basic.invalid_argument`
- `basic.message_size`
- `basic.name_too_long`
- `basic.network_down`
- `basic.network_reset`
- `basic.network_unreachable`
- `basic.no_descriptors`
- `basic.no_buffer_space`

- basic.no_memory
- basic.no_permission
- basic.no_protocol_option
- basic.no_such_device
- basic.not_connected
- basic.not_socket
- basic.operation_aborted
- basic.operation_not_supported
- basic.shut_down
- basic.timed_out
- basic.try_again
- basic.would_block

netdb.h errors

- netdb.host_not_found
- netdb.host_not_found_try_again
- netdb.no_data
- netdb.no_recovery

addrinfo errors

- addrinfo.service_not_found
- addrinfo.socket_type_not_supported

Misc errors

- misc.already_open
- misc.eof
- misc.not_found
- misc.fd_set_failure

format

Synopsis

```
format(fmt: string[, ...]) -> string
```

Description

Formats args according to specifications in `fmt` and returns the result as a string.

Supported arguments:

- `boolean`
- `number`
- `string`

You may also specify pairs. First element must be a string and it works as a named argument.

The full specification for the format string can be found in libfmt homepage.

`format()` is a global so it doesn't need to be `require()`d.

Example

```
format("{0}, {1}, {2}", 'a', 'b', 'c')
-- Result: "a, b, c"

format("{}, {}, {}", 'a', 'b', 'c')
-- Result: "a, b, c"

format("{2}, {1}, {0}", 'a', 'b', 'c')
-- Result: "c, b, a"

format("{0}{1}{0}", "abra", "cad") -- arguments' indices can be repeated
-- Result: "abracadabra"

format("{:.{}f}", 3.14, 1)
-- Result: "3.1"

format("Elapsed time: {s:.2f} seconds", {"s", 1.23})
-- Result: "Elapsed time: 1.23 seconds"
```

byte_span

```
local byte_span = require 'byte_span'
```

A span of bytes. In Emilua, they're used as network buffers.

Plugin authors

This class is intended for network buffers in a proactor-based network API (i.e. true asynchronous IO). A NIC could be writing to this memory region while the program is running. This has the same effect of another thread writing to the same memory region.

If you're writing state machines, do not construct the state machine on top of the memory region pointed by a `byte_span`. It's not safe to store state here as buggy Lua applications could mutate this area in a racy way. Only use the memory region as the result of operations.

A future Emilua release could introduce read-write locks, but as of now I'm unconvinced of their advantages here.

It's modeled after Golang's slices. However 1-indexed access is used.

Functions

`new(length: integer[, capacity: integer]) → byte_span`

Constructor.

When the `capacity` argument is omitted, it defaults to the specified `length`.

`slice(self[, start: integer, end: integer]) → byte_span`

Returns a new `byte_span` that points to a slice of the same memory region.

The `start` and `end` indices are optional; they default to `1` and the `byte_span`'s length respectively.

We can grow a `byte_span` to its capacity by slicing it again.

Invalid ranges (e.g. `start` below `1`, a `byte_span` running beyond its capacity, negative indexes, ...) will raise `EINVAL`.

`copy(self, src: byte_span|string) → integer`

Copy `src` into `self`.

Returns the number of elements copied.

Copying between slices of different lengths is supported (it'll copy only up to the smaller number of

elements). In addition it can handle source and destination spans that share the same underlying memory, handling overlapping spans correctly.

`append() → byte_span`

```
function append(self, ...: byte_span|string|nil) -> byte_span ①
function append(...: byte_span|string|nil) -> byte_span       ②
```

Returns a new `byte_span` by appending trailing arguments into `self`. If `self`'s capacity is enough to hold all data, the underlying memory is modified in place. Otherwise the returned `byte_span` will point to newly allocated memory[1].

For the second overload (non-member function), a new byte span is created from scratch.

Functions (string algorithms)

These functions operate in terms of octets/bytes (kinda like an 8-bit ASCII) and have no concept of UTF-8 encoding.

`starts_with(self, prefix: string|byte_span) → boolean`

Returns whether `self` begins with `prefix`.

`ends_with(self, suffix: string|byte_span) → boolean`

Returns whether `self` ends with `suffix`.

`find(self, tgt: string|byte_span[, start: integer]) → integer|nil`

Finds the first substring equals to `tgt` and returns its index, or `nil` if not found.

`rfind(self, tgt: string|byte_span[, end_: integer]) → integer|nil`

Finds the last substring equals to `tgt` and returns its index, or `nil` if not found.

`find_first_of(self, strlist: string|byte_span[, start: integer]) → integer|nil`

Finds the first octet equals to any of the octets within `strlist` and returns its index, or `nil` if not found.

`find_last_of(self, strlist: string|byte_span[, end_: integer]) → integer|nil`

Finds the last octet equals to any of the octets within `strlist` and returns its index, or `nil` if not found.

`find_first_not_of(self, strlist: string|byte_span[, start: integer]) → integer|nil`

Finds the first octet not equals to any of the octets within `strlist` and returns its index, or `nil` if not

found.

`find_last_not_of(self, strlist: string|byte_span[, end: integer]) → integer|nil`

Finds the last octet not equals to any of the octets within `strlist` and returns its index, or `nil` if not found.

`trimmed(self[, lws: string|byte_span = " \f\n\r\t\v"]) → byte_span`

Returns a slice from `self` that doesn't start nor ends with any octet from `lws`.

Properties

`capacity: integer`

The capacity.

Metamethods

- `__tostring()`
- `__len()`
- `__index()`
- `__newindex()`
- `__eq()`

You can index the spans by numerical valued keys and the numerical (ASCII) value for the underlying byte will be returned (or assigned on `__newindex()`).

[1] Allocation strategy (the new `byte_span`'s capacity) is left unspecified and may change among Emilua releases.

condition_variable

```
local condition_variable = require('condition_variable')

local function queue_consumer()
    scope(function()
        scope_cleanup_push(function() queue_mtx:unlock() end)
        queue_mtx:lock()
        while #queue == 0 do
            queue_cond:wait(queue_mtx)
        end
        for _, e in ipairs(queue) do
            consume_item(e)
        end
        queue = {}
    end)
end
```

A condition variable.

Functions

new() → condition_variable

Constructor.

wait(self, m: mutex)

Read `pthread_cond_wait()`

`wait()` is an interruption point. Prior to the delivery of the interruption request, the underlying mutex is re-acquired under the hood.

notify_all(self)

Read `pthread_cond_broadcast()`.

notify_one(self)

Read `pthread_cond_signal()`.

Notifying without a lock

If the condition variable, the notifier fiber and the waiting fiber all run in the same thread (and cooperative multitasking is used instead preemptive multitasking), then there is enough level of determinism to lift one restriction that exists in traditional condition variables.

> Even if the shared variable is atomic, it must be modified under the mutex

> in order to correctly publish the modification to the waiting thread.
>
> — https://en.cppreference.com/w/cpp/thread/condition_variable

The reason why this restriction on the notifier fiber/thread exists is to avoid a race. Consider the following waiter fiber and the notifier fiber:

```
local function consumer()
    scope(function()
        scope_cleanup_push(function() m:unlock() end)
        m:lock()
        while not ready do
            c:wait(m)
        end

        -- ...
    end)
end

local function producer()
    ready = true
    c:notify_one()
end
```

Pay attention to the points when the waiter fiber checks if the event has been signalled by testing `ready` and the instant it blocks on `c.wait()`. If the notifier fiber mutates the shared variable and calls `c.notify_one()` between these two points, then the signalization is lost. `c.notify_one()` would be called by the time there would be no fiber blocked on `c.wait()`. That's why the notifier fiber need to mutate the shared variable through a mutex.

In Emilua, this restriction doesn't apply (as long as there are no suspension points between the time the waiting fiber tests the condition and calls `c.wait()`) and the notifier fiber can mutate the shared variable without holding a lock on the mutex. In this case, the condition variable essentially becomes a non-suspending way (post semantics) to unpark a parked fiber (yes, I've exploited this property in the past to avoid a few round-trips).

filesystem.path

Objects of this class abstract path-manipulation algorithms for the host operating system.

Methods from this class are purely computational and never trigger any syscall. They only operate on the in-memory representation of a path. They do not perform any operation on the filesytem. They do not initiate any I/O request.

Paths are immutable. Any operation on a path will return a new path with the result.

Functions

`new() → path`

```
new()    ①
new(str) ②
```

① Default constructor.

② Create a path from an UTF-8 encoded string (in the host system format).

`from_generic(source: string) → path`

Creates a path from the generic non-native format.

`to_generic(self) → string`

Returns the path in the generic format encoded in UTF-8.

`iterator(self) → function`

Returns an iterator to the path components (as strings). The iteration order follows:

1. The root name, if any.
2. The root directory, if any.
3. The sequence of file names, omitting directory separators.
4. If there is a directory separator after the last file name in the path, the last element is an empty element.

`make_preferred(self) → path`

Returns a new path where all directory separators are converted to the preferred directory separator.

On Windows, where `"\"` is the preferred separator, the path `"foo/bar"` will be converted to `"foo\bar"`.

`remove_filename(self) → path`

Returns a new path where the filename component is removed.

`replace_filename(self, replacement: string|path) → path`

Returns a new path where the filename component is replaced.

`replace_extension(self[, replacement: string|path]) → path`

Returns a new path where the extension is replaced (or removed on `nil`).

`lexically_normal(self) → path`

Returns a new path converted to normal form.

`lexically_relative(self, base: string|path) → path`

Returns a new path where `self` is made relative to `base`.

`lexically_proximate(self, base: string|path) → path`

Same as above if the return is non empty. Same as `self`, otherwise.

Properties

`root_name: string`

Returns the root name, or an empty path.

`root_directory: string`

Returns the root directory, or an empty path.

`root_path: path`

Returns `path.new(root_name) / root_directory`.

`relative_path: path`

Returns path relative to `root_path`.

`parent_path: path`

Returns the path to the parent directory.

`filename: string`

Returns filename component.

`stem: string`

Returns filename component stripped of its extension.

`extension: string`

Returns the extension of the filename component.

`empty: boolean`

Whether the path is empty.

`has_root_path: boolean`

Whether the root path is non-empty.

`has_root_name: boolean`

Whether the root name is non-empty.

`has_root_directory: boolean`

Whether the root directory is non-empty.

`has_relative_path: boolean`

Whether relative path is non-empty.

`has_parent_path: boolean`

Whether the parent path is non-empty.

`has_filename: boolean`

Whether the filename is non-empty.

`has_stem: boolean`

Whether the stem is non-empty.

`has_extension: boolean`

Whether the extension is non-empty.

`is_absolute: boolean`

Whether the path is absolute.

`is_relative: boolean`

Whether the path is relative.

Metamethods

- `__tostring()`: Encodes the native representation as UTF-8 and returns it.
- `__eq()`: Compares two paths lexicographically.
- `__lt()`: Compares two paths lexicographically.
- `__le()`: Compares two paths lexicographically.
- `__div()`: Concatenates two paths with a directory separator.
- `__concat()`: Concatenates the underlying native representation of the paths (i.e. no additional directory separators are introduced). This operation may not be portable between operating systems.

Module attributes

`preferred_separator: string`

The preferred directory separator on the host operating system encoded in UTF-8.

filesystem.mode

Synopsis

```
local fs = require "filesystem"

fs.mode(user: integer, group: integer, other: integer) -> integer
    return bit.bor(bit.lshift(user, 6), bit.lshift(group, 3), other)
end
```

Description

A helper function to create POSIX mode permission bits.

filesystem.directory_entry

The object returned by directory iterators. Objects of this class cannot be created directly.

Functions

`refresh(self)`

Updates the cached file attributes.

Properties

`path: filesystem.path`

The path the entry refers to.

`file_size: integer`

The size in bytes of the file to which the directory entry refers to.

`hard_link_count: integer`

The number of hard links referring to the file to which the directory entry refers to.

`last_write_time: filesystem.clock.time_point`

The time of the last data modification of the file to which the directory entry refers to.

`status`

Returns the same value as `filesystem.status()`.

`symlink_status`

Returns the same value as `filesystem.symlink_status()`.

filesystem.directory_iterator

Synopsis

```
local fs = require "filesystem"
fs.directory_iterator(p: fs.path[, opts: table]) -> function
```

Description

Returns an iterator function that, each time it is called, returns a `filesystem.directory_entry` object for an element of the directory `p`.

opts

`skip_permission_denied: boolean = false`

On `true`, an `EPERM` will not be reported. Instead, an iterator to an empty collection will be returned.

filesystem.recursive_directory_iterator

Synopsis

```
local fs = require "filesystem"
fs.recursive_directory_iterator(p: fs.path[, opts: table]) -> function, handle
```

Description

Returns an iterator function, and a handle to control iteration.

Each time the iterator is called, returns a `filesystem.directory_entry` object for an element of the directory `p` (and, recursively, over the entries of all of its subdirectories), and the current recursion depth (an `integer`).

opts

`skip_permission_denied: boolean = false`

: Whether to skip directories that would otherwise result in `EPERM`.

`follow_directory_symlink: boolean = false`

: Whether to follow directory symlinks.

Wrapping the iterator to skip over CVS files.

Some programs such as `rsync` have command line options such as `--cvs-exclude` that skip over unwanted files for the directory traversal. Such patterns can be easily abstracted on top of `recursive_directory_iterator`. Here's the implementation for a function that does just that:

```
function cvs_exclude(iter, ctrl)
    local function next()
        local entry, depth = iter()
        if entry == nil then
            return
        end

        local p = entry.path.filename
        if p == ".git" or p == ".svn" or p == ".hg" then
            ctrl:disable_recursion_pending()
        end
        return entry, depth
    end
    return next, ctrl
end
```

The same trick can be used to create functions to perform shell globbing.

`handle` functions

`pop(self)`

Moves the iterator one level up in the directory hierarchy.

`disable_recursion_pending(self)`

Disables recursion until the next increment.

`handle` properties

`recursion_pending: boolean`

Whether the recursion is disabled for the current directory.

Example

```lua
local fs = require "filesystem"

for entry, depth in fs.recursive_directory_iterator(fs.path.new(".")) do
    print(string.rep("\t", depth) .. entry.path.filename)
end
```

filesystem.absolute

Synopsis

```
local fs = require "filesystem"
fs.absolute(p: fs.path) -> fs.path
```

Description

Returns a path referencing the same file system location as `p`, for which `filesystem.path.is_absolute` is `true`.

filesystem.canonical

Synopsis

```
local fs = require "filesystem"
fs.canonical(p: fs.path) -> fs.path
```

Description

Returns a canonical absolute path referencing the same file system location as `p`.

filesystem.weakly_canonical

Synopsis

```
local fs = require "filesystem"
fs.weakly_canonical(p: fs.path) -> fs.path
```

Description

Returns a path in normal form referencing the same file system location as `p`.

filesystem.relative

Synopsis

```
local fs = require "filesystem"
fs.relative(p: fs.path, base: fs.path = fs.current_working_directory()) -> fs.path
```

Description

See https://en.cppreference.com/w/cpp/filesystem/relative.

filesystem.proximate

Synopsis

```
local fs = require "filesystem"
fs.proximate(p: fs.path, base: fs.path = fs.current_working_directory()) -> fs.path
```

Description

See https://en.cppreference.com/w/cpp/filesystem/relative.

filesystem.current_working_directory

Synopsis

```
local fs = require "filesystem"
fs.current_working_directory() -> fs.path ①
fs.current_working_directory(p: fs.path) ②
```

① Get the current working directory.

② Set the current working directory.

Description

Get or set the current working directory for the calling process.

Only the master VM is allowed to change the current working directory.

filesystem.copy

Synopsis

```
local fs = require "filesystem"
fs.copy(from: fs.path, to: fs.path[, opts: table])
```

Description

See https://en.cppreference.com/w/cpp/filesystem/copy.

opts

`existing: "skip"|"overwrite"|"update"|nil`

Behavior when the file already exists.

`nil`

Report an error.

`"skip"`

Keep the existing file, without reporting an error.

`"overwrite"`

Replace the existing file.

`"update"`

Replace the existing file only if it is older than the file being copied.

`recursive: boolean = false`

`false`

Skip subdirectories.

`true`

Recursively copy subdirectories and their content.

`symlinks: "copy"|"skip"|nil`

`nil`

Follow symlinks.

`"copy"`

Copy symlinks as symlinks, not as the files they point to.

`"skip"`

Ignore symlinks.

`copy: "directories_only"|"create_symlinks"|"create_hard_links"|nil`

`nil`

Copy file content.

`"directories_only"`

Copy the directory structure, but do not copy any non-directory files.

`"create_symlinks"`

Instead of creating copies of files, create symlinks pointing to the originals.

`"create_hard_links"`

Instead of creating copies of files, create hardlinks that resolve to the same files as the originals.

filesystem.copy_file

Synopsis

```
local fs = require "filesystem"
fs.copy_file(from: fs.path, to: fs.path[, opts: table]) -> boolean
```

Description

See https://en.cppreference.com/w/cpp/filesystem/copy_file.

opts

`existing: "skip"|"overwrite"|"update"|nil`

Behavior when the file already exists.

`nil`

Report an error.

`"skip"`

Keep the existing file, without reporting an error.

`"overwrite"`

Replace the existing file.

`"update"`

Replace the existing file only if it is older than the file being copied.

filesystem.copy_symlink

Synopsis

```
local fs = require "filesystem"
fs.copy_symlink(from: fs.path, to: fs.path)
```

Description

See https://en.cppreference.com/w/cpp/filesystem/copy_symlink.

filesystem.create_directory

Synopsis

```
local fs = require "filesystem"
fs.create_directory(p: fs.path[, existing_p: fs.path]) -> boolean
fs.create_directories(p: fs.path) -> boolean
```

Description

Creates the directory `p` as if by POSIX `mkdir()` with a second argument of `0777`. If the function fails because `p` resolves to an existing directory, no error is reported.

If `existing_p` is given, then the attributes of the new directory are copied from `existing_p`.

`filesystem.create_directories()` calls `filesystem.create_directory()` for every element of `p` that does not already exist.

Returns whether a directory was created for the directory `p` resolves to.

filesystem.create_hard_link

Synopsis

```
local fs = require "filesystem"
fs.create_hard_link(target: fs.path, link: fs.path)
```

Description

See https://en.cppreference.com/w/cpp/filesystem/create_hard_link.

filesystem.create_symlink

Synopsis

```
local fs = require "filesystem"
fs.create_symlink(target: fs.path, link: fs.path)
fs.create_directory_symlink(target: fs.path, link: fs.path)
```

Description

See https://en.cppreference.com/w/cpp/filesystem/create_symlink.

filesystem.mkfifo

Synopsis

```
local fs = require "filesystem"
fs.mkfifo(p: fs.path, mode: integer)
```

Description

See mkfifo(3).

filesystem.mknod

Synopsis

```
local fs = require "filesystem"
fs.mknod(p: fs.path, mode: integer, dev: integer)
```

Description

See mknod(3).

filesystem.makedev

Synopsis

```
local fs = require "filesystem"
fs.makedev(major: integer, minor: integer) -> integer
```

Description

See makedev(3).

filesystem.equivalent

Synopsis

```
local fs = require "filesystem"
fs.equivalent(p1: fs.path, p2: fs.path) -> boolean
```

Description

See https://en.cppreference.com/w/cpp/filesystem/equivalent.

filesystem.file_size

Synopsis

```
local fs = require "filesystem"
fs.file_size(p: fs.path) -> integer
```

Description

For a regular file `p`, returns its size in bytes.

filesystem.hard_link_count

Synopsis

```
local fs = require "filesystem"
fs.hard_link_count(p: fs.path) -> integer
```

Description

Returns the number of hard links for the filesystem object identified by path `p`.

filesystem.clock

```lua
local clock = require('filesystem').clock
```

A clock to represent file time. Its epoch is unspecified.

Functions

`from_system(tp: time.system_clock.time_point) → clock.time_point`

Converts `tp` to a clock.time_point representing the same point in time.

`time_point` functions

`add(self, secs: number)`

Modifies the time point by the given duration.

When the duration is converted to the native tick representation of the clock, it'll be rounded to the nearest time point rounding to even in halfway cases.

`sub(self, secs: number)`

Modifies the time point by the given duration.

When the duration is converted to the native tick representation of the clock, it'll be rounded to the nearest time point rounding to even in halfway cases.

`to_system(self) → time.system_clock.time_point`

Converts `self` to a time.system_clock.time_point representing the same point in time.

`time_point` properties

`seconds_since_epoch: number`

The number of elapsed seconds since the clock's epoch.

`time_point` metamethods

- `__add()`
- `__sub()`
- `__eq()`
- `__lt()`

- `__le()`

filesystem.last_write_time

Synopsis

```
local fs = require "filesystem"
fs.last_write_time(p: fs.path) -> fs.clock.time_point ①
fs.last_write_time(p: fs.path, tp: fs.clock.time_point) ②
```

① Get last write time.

② Set last write time.

Description

Get or set the time of the last modification of `p`.

Symlinks are followed.

It is not guaranteed that immediately after setting the write time, the value returned by (1) is the same as what was passed as the argument to (2) because the file system's time may be more granular than `filesystem.clock.time_point`.

filesystem.chown

Synopsis

```
local fs = require "filesystem"
fs.chown(p: fs.path, owner: integer, group: integer)
fs.lchown(p: fs.path, owner: integer, group: integer)
```

Description

Changes POSIX owner and group of the file to which p resolves.

If the owner or group is specified as -1, then that ID is not changed.

filesystem.chmod

Synopsis

```
local fs = require "filesystem"
fs.chmod(p: fs.path, mode: integer)
fs.lchmod(p: fs.path, mode: integer)
```

Description

Changes POSIX access permissions of the file to which p resolves.

filesystem.read_symlink

Synopsis

```
local fs = require "filesystem"
fs.read_symlink(p: fs.path) -> fs.path
```

Description

Returns a new path which refers to the target of the symbolic link.

filesystem.remove

Synopsis

```
local fs = require "filesystem"
fs.remove(p: fs.path) -> boolean
fs.remove_all(p: fs.path) -> integer
```

Description

See https://en.cppreference.com/w/cpp/filesystem/remove.

filesystem.rename

Synopsis

```
local fs = require "filesystem"
fs.rename(old_p: fs.path, new_p: fs.path)
```

Description

See https://en.cppreference.com/w/cpp/filesystem/rename.

filesystem.resize_file

Synopsis

```
local fs = require "filesystem"
fs.resize_file(p: fs.path, new_size: integer)
```

Description

See https://en.cppreference.com/w/cpp/filesystem/resize_file.

filesystem.is_empty

Synopsis

```
local fs = require "filesystem"
fs.is_empty(p: fs.path) -> boolean
```

Description

Checks whether the given path refers to an empty file or directory.

filesystem.exists

Synopsis

```
local fs = require "filesystem"
fs.exists(p: fs.path) -> boolean
```

Description

Checks whether the given path refers to an existing file or directory.

filesystem.is_block_file

Synopsis

```
local fs = require "filesystem"
fs.is_block_file(p: fs.path) -> boolean
```

Description

Checks whether the given path refers to a block special file.

filesystem.is_character_file

Synopsis

```
local fs = require "filesystem"
fs.is_character_file(p: fs.path) -> boolean
```

Description

Checks whether the given path refers to a character special file.

filesystem.is_directory

Synopsis

```
local fs = require "filesystem"
fs.is_directory(p: fs.path) -> boolean
```

Description

Checks whether the given path refers to a directory.

filesystem.is_fifo

Synopsis

```
local fs = require "filesystem"
fs.is_fifo(p: fs.path) -> boolean
```

Description

Checks whether the given path refers to a FIFO or pipe file.

filesystem.is_other

Synopsis

```
local fs = require "filesystem"
fs.is_other(p: fs.path) -> boolean
```

Description

Checks whether the given path refers to a file of type other type. That is, the file exists, but is neither regular file, nor directory nor a symlink.

filesystem.is_regular_file

Synopsis

```
local fs = require "filesystem"
fs.is_regular_file(p: fs.path) -> boolean
```

Description

Checks whether the given path refers to a regular file.

filesystem.is_socket

Synopsis

```
local fs = require "filesystem"
fs.is_socket(p: fs.path) -> boolean
```

Description

Checks whether the given path refers to a named IPC socket.

filesystem.is_symlink

Synopsis

```
local fs = require "filesystem"
fs.is_symlink(p: fs.path) -> boolean
```

Description

Checks whether the given path refers to a symbolic link.

filesystem.space

Synopsis

```
local fs = require "filesystem"
fs.space(p: fs.path) -> { capacity: integer, free: integer, available: integer }
```

Description

Determines the information about the filesystem on which the pathname `p` is located.

Bytes are used for the units.

filesystem.status

Synopsis

```
local fs = require "filesystem"
fs.status(p: fs.path) -> { type: string, mode: integer|"unknown" }
fs.symlink_status(p: fs.path) -> { type: string, mode: integer|"unknown" }
```

Description

See https://en.cppreference.com/w/cpp/filesystem/status.

The acceptable strings for the member named `type` in the returned object are:

- `"not_found"`
- `"regular"`
- `"directory"`
- `"symlink"`
- `"block"`
- `"character"`
- `"fifo"`
- `"socket"`
- `"junction"` (Windows-only)
- `"unknown"`

The member named `mode` in the returned object refers to the POSIX file access mode (permissions).

filesystem.temp_directory_path

Synopsis

```
local fs = require "filesystem"
fs.temp_directory_path() -> fs.path
```

Description

Returns the directory location suitable for temporary files.

filesystem.umask

Synopsis

```
local fs = require "filesystem"
fs.umask(mask: integer) -> integer
```

Description

Sets the file mode creation mask (umask) of the calling process to `mask & 0777`.

Returns the old mask.

Only the master VM is allowed to use this function.

filesystem.cap_get_file

Synopsis

```
local fs = require "filesystem"
fs.cap_get_file(path: fs.path) -> system.linux_capabilities
```

Description

See cap_get_file(3).

filesystem.cap_set_file

Synopsis

```
local fs = require "filesystem"
fs.cap_set_file(path: fs.path, caps: system.linux_capabilities)
```

Description

See cap_set_file(3).

file.open_flag

This module contains flag constants useful to send/receive operations.

append

Open the file in append mode.

create

Create the file if it does not exist.

exclusive

Ensure a new file is created. Must be combined with create.

read_only

Open the file for reading.

read_write

Open the file for reading and writing.

sync_all_on_write

Open the file so that write operations automatically synchronise the file data and metadata to disk (`FILE_FLAG_WRITE_THROUGH`/`O_SYNC`).

truncate

Open the file with any existing contents truncated.

write_only

Open the file for writing.

file.random_access

Functions

`new() → file.random_access`

```
new()                    ①
new(fd: file_descriptor) ②
```

① Default constructor.

② Converts a file descriptor into a `file.random_access` object.

`open(self, path: filesystem.path, flags: integer)`

Open the file using the specified path.

`flags` is an or-combination of values from file.open_flag(3em).

`close(self)`

Close the file.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous read or write operations will be cancelled immediately, and will complete with the `boost::asio::error::operation_aborted` error.

`cancel(self)`

Cancel all asynchronous operations associated with the file.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous read and write operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, fd: file_descriptor)`

Assign an existing native file to `self`.

`release(self) → file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous read and write operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native file is then transferred to the caller.

`resize(self, n: integer)`

Alter the size of the file.

This function resizes the file to the specified size, in bytes. If the current file size exceeds n then any extra data is discarded. If the current size is less than n then the file is extended and filled with zeroes

`read_some_at(self, offset: integer, buffer: byte_span) → integer`

Read data from the file at the specified offset and blocks current fiber until it completes or errs.

Returns the number of bytes read.

Lua conventions on index starting at `1` are ignored. Indexes here are OS-mandated and start at `0`.

`write_some_at(self, offset: integer, buffer: byte_span) → integer`

Write data to the file at the specified and blocks current fiber until it completes or errs.

Returns the number of bytes written.

Lua conventions on index starting at `1` are ignored. Indexes here are OS-mandated and start at `0`.

Properties

`is_open: boolean`

Whether the file is open.

`size: integer`

The size of the file.

file.stream

Functions

`new() → file.stream`

```
new()                    ①
new(fd: file_descriptor) ②
```

① Default constructor.

② Converts a file descriptor into a `file.stream` object.

`open(self, path: filesystem.path, flags: integer)`

Open the file using the specified path.

`flags` is an or-combination of values from file.open_flag(3em).

`close(self)`

Close the file.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous read or write operations will be cancelled immediately, and will complete with the `boost::asio::error::operation_aborted` error.

`cancel(self)`

Cancel all asynchronous operations associated with the file.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous read and write operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, fd: file_descriptor)`

Assign an existing native file to `self`.

`release(self) → file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous read and write operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native file is then transferred to the caller.

`resize(self, n: integer)`

Alter the size of the file.

This function resizes the file to the specified size, in bytes. If the current file size exceeds n then any extra data is discarded. If the current size is less than n then the file is extended and filled with zeroes

`seek(self, offset: integer, whence: string) → integer`

Sets and gets the file position, measured from the beginning of the file, to the position given by `offset` plus a base specified by the string `whence`, as follows:

`"set"`
: Seek to an absolute position.

`"cur"`
: Seek to an offset relative to the current file position.

`"end"`
: Seek to an offset relative to the end of the file.

Returns the final file position, measured in bytes from the beginning of the file.

Lua conventions on index starting at `1` are ignored. Indexes here are OS-mandated and start at `0`.

`read_some(self, buffer: byte_span) → integer`

Read data from the stream file and blocks current fiber until it completes or errs.

Returns the number of bytes read.

`write_some(self, buffer: byte_span) → integer`

Write data to the stream file and blocks current fiber until it completes or errs.

Returns the number of bytes written.

Properties

`is_open: boolean`

Whether the file is open.

`size: integer`

The size of the file.

file.read_all_at

Synopsis

```
local file = require "file"
file.read_all_at(io_object, offset: integer, buffer: byte_span) -> integer
```

Description

Attempt to read a certain amount of data at the specified offset before returning.

This operation is implemented in terms of zero or more calls to the device's `read_some_at` function.

file.read_at_least_at

Synopsis

```
local file = require "file"
file.read_at_least_at(io_object, offset: integer, buffer: byte_span, minimum: integer)
-> integer
```

Description

Attempt to read a certain amount of data at the specified offset before returning.

This operation is implemented in terms of zero or more calls to the device's `read_some_at` function.

file.write_all_at

Synopsis

```
local file = require "file"
file.write_all_at(io_object, offset: integer, buffer: byte_span|string) -> integer
```

Description

Write all of the supplied data at the specified offset before returning.

This operation is implemented in terms of zero or more calls to the device's `write_some_at` function.

file.write_at_least_at

Synopsis

```
local file = require "file"
file.write_at_least_at(io_object, offset: integer, buffer: byte_span, minimum:
integer) -> integer
```

Description

Write data until a `minimum` number of bytes has been transferred at the specified offset before returning.

This operation is implemented in terms of zero or more calls to the device's `write_some_at` function.

ip.address

A variant type to represent IPv4 and IPv6 addresses. Some features are only available for one version of the protocol and will raise an error when you try to use it against an IP address of a different version.

Functions

new() → ip.address

```
new()    ①
new(str) ②
```

① Default constructor.

② Create an IPv4 address in dotted decimal form, or from an IPv6 address in hexadecimal notation.

any_v4() → ip.address

Create an address object that represents any (v4) address.

any_v6() → ip.address

Create an address object that represents any (v6) address.

loopback_v4() → ip.address

Create an address object that represents the loopback (v4) address.

loopback_v6() → ip.address

Create an address object that represents the loopback (v6) address.

broadcast_v4() → ip.address

Create an address object that represents the broadcast (v4) address.

Functions (v4)

to_v6(self) → ip.address

Create an IPv4-mapped IPv6 address.

Functions (v6)

to_v4(self) → ip.address

Create an IPv4 address from a IPv4-mapped IPv6 address.

Properties

`is_loopback: boolean`

Whether the address is a loopback address.

`is_multicast: boolean`

Whether the address is a multicast address.

`is_unspecified: boolean`

Whether the address is unspecified.

`is_v4: boolean`

Whether the address is an IP version 4 address.

`is_v6: boolean`

Whether the address is an IP version 6 address.

Properties (v6)

An error will be raised if you try to use against a v4 object.

`is_link_local: boolean`

Whether the address is link local.

`is_multicast_global: boolean`

Whether the address is a global multicast address.

`is_multicast_link_local: boolean`

Whether the address is a link-local multicast address.

`is_multicast_node_local: boolean`

Whether the address is a node-local multicast address.

`is_multicast_org_local: boolean`

Whether the address is a org-local multicast address.

`is_multicast_site_local: boolean`

Whether the address is a site-local multicast address.

`is_site_local: boolean`

Whether the address is site local.

`is_v4_mapped: boolean`

Whether the address is a mapped IPv4 address.

`scope_id: integer`

The scope ID of the address. Read-write property.

Metamethods

- `__tostring()`
- `__eq()`
- `__lt()`
- `__le()`

ip.address_info_flag

This module contains flag constants useful to name resolvers.

The following example demonstrates how you can use these flags in other modules:

```
ip.tcp.get_address_info(
    'www.example.com',
    'http',
    ip.address_info_flag.address_configured
)
ip.tcp.get_address_info(
    'www.example.com',
    80,
    bit.bor(
        ip.address_info_flag.all_matching,
        ip.address_info_flag.v4_mapped
    )
)
```

address_configured

The flag with same name in Boost.Asio:

> Only return IPv4 addresses if a non-loopback IPv4 address is configured for the system. Only return IPv6 addresses if a non-loopback IPv6 address is configured for the system.

all_matching

The flag with same name in Boost.Asio:

> If used with v4_mapped, return all matching IPv6 and IPv4 addresses.

canonical_name

The flag with same name in Boost.Asio:

> Determine the canonical name of the host specified in the query.

passive

The flag with same name in Boost.Asio:

> Indicate that returned endpoint is intended for use as a locally bound socket

> endpoint.

v4_mapped

The flag with same name in Boost.Asio:

> If the query protocol family is specified as IPv6, return IPv4-mapped IPv6 addresses on finding no IPv6 addresses.

ip.get_address_info

Synopsis

```
local ip = require "ip"

ip.tcp.get_address_info()
ip.tcp.get_address_v4_info()
ip.tcp.get_address_v6_info()
ip.udp.get_address_info()
ip.udp.get_address_v4_info()
ip.udp.get_address_v6_info()

function(host: string|ip.address, service: string|integer[, flags: integer])
    -> { address: ip.address, port: integer, canonical_name: string|nil }[]
```

Description

Forward-resolves host and service into a list of endpoint entries. Current fiber is suspended until operation finishes.

`flags` is `0` or an or-combination of values from ip.address_info_flag(3em).

If no `flags` are passed to this function (i.e. `flags` is `nil`) then this function will follow the glibc defaults even on non-glibc systems: `bit.bor(address_configured,v4_mapped)`.

Returns a list of entries. Each entry will be a table with the following members:

- `address: ip.address`.
- `port: integer`.

If `ip.address_info_flag.canonical_name` is passed in `flags` then each entry will also include:

- `canonical_name: string`.

More info on Boost.Asio documentation.

If `host` is an `ip.address` then no host name resolution should be attempted.

If `service` is a number then no service name resolution should be attempted.

ip.get_name_info

Synopsis

```
local ip = require "ip"

ip.tcp.get_name_info()
ip.udp.get_name_info()

function(a: ip.address, port: integer)
    -> { host_name: string, service_name: string }[]
```

Description

Reverse-resolves the endpoint into a list of entries. Current fiber is suspended until operation finishes.

Returns a list of entries. Each entry will be a table with the following members:

- `host_name: string`.
- `service_name: string`.

More info on Boost.Asio documentation.

ip.connect

Synopsis

```
local ip = require "ip"
ip.connect(sock, resolve_results: table[, condition: function]) -> ip.address, integer
```

Description

Attempts to connect a socket to one of a sequence of endpoints. It does this by repeated calls to the `socket`'s connect member function, once for each endpoint in the sequence, until a connection is successfully established.

Parameters

`sock`

The socket to be connected. If the socket is already open, it will be closed.

`resolve_results`

The return from the function `get_address_info()`. If the sequence is empty, the error `not_found` will be raised.

`condition`

A function that is called prior to each connection attempt. The signature of the function object must be:

```
function condition(last_error, next_address, next_port) -> boolean
```

The `last_error` parameter contains the result from the most recent connect operation. Before the first connection attempt, `last_error` is `nil`. The next parameters together specify the next endpoint to be tried. The closure should return `true` if the next endpoint should be tried, and `false` if it should be skipped.

Example

```
local addr, port = ip.connect(
    sock, ip.tcp.get_address_info("www.example.com", "http"),
    function(last_error, next_addr, next_port)
        if last_error then
            print("Error: " .. tostring(last_error))
        end
```

```
        print("Trying: " .. ip.tostring(next_addr, next_port))
        return true
    end
)
print("Connected to: " .. ip.tostring(addr, port))
```

ip.host_name

Synopsis

```
local ip = require "ip"
ip.host_name() -> string
```

Description

Get the current host name.

ip.tostring

Synopsis

```
local ip = require "ip"
ip.tostring(addr: ip.address[, port: integer]) -> string
```

Description

Convert a traditional network endpoint (IP address + unsigned 16-bit integer) to its string representation. If `port` is `nil`, then perform the equivalent of `tostring(addr)`.

ip.toendpoint

Synopsis

```
local ip = require "ip"
ip.toendpoint(ep: string) -> ip.address, integer
```

Description

Convert a traditional network endpoint (IP address + unsigned 16-bit integer) from its string representation to its decoupled members.

ip.message_flag

This module contains flag constants useful to send/receive operations.

do_not_route

The flag with same name in Boost.Asio:

> Specify that the data should not be subject to routing.

end_of_record

The flag with same name in Boost.Asio:

> Specifies that the data marks the end of a record.

out_of_band

The flag with same name in Boost.Asio:

> Process out-of-band data.

peek

The flag with same name in Boost.Asio:

> Peek at incoming data without removing it from the input queue.

ip.tcp.acceptor

```
local a = ip.tcp.acceptor.new()
a:open('v4')
a:set_option('reuse_address', true)
a:bind('127.0.0.1', 8080)
a:listen()

while true do
    local s = a:accept()
    spawn(function()
        my_client_handler(s)
    end)
end
```

Functions

new() → ip.tcp.acceptor

Constructor.

open(self, address_family: "v4"|"v6"|ip.address)

Open the acceptor.

`address_family` can be either `"v4"` or `"v6"`. If you provide an `ip.address` object, the appropriate value will be inferred.

set_option(self, opt: string, val)

Set an option on the acceptor.

Currently available options are:

`"reuse_address"`
: Check Boost.Asio documentation.

`"enable_connection_aborted"`
: Check Boost.Asio documentation.

`"debug"`
: Check Boost.Asio documentation.

`"v6_only"`
: Check Boost.Asio documentation.

`get_option(self, opt: string) → value`

Get an option from the acceptor.

Currently available options are:

`"reuse_address"`
: Check Boost.Asio documentation.

`"enable_connection_aborted"`
: Check Boost.Asio documentation.

`"debug"`
: Check Boost.Asio documentation.

`"v6_only"`
: Check Boost.Asio documentation.

`bind(self, addr: ip.address|string, port: integer)`

Bind the acceptor to the given local endpoint.

`listen(self [, backlog: integer])`

Place the acceptor into the state where it will listen for new connections.

`backlog` is the maximum length of the queue of pending connections. If not provided, an implementation defined maximum length will be used.

`accept(self) → ip.tcp.socket`

Initiate an accept operation and blocks current fiber until it completes or errs.

`close(self)`

Close the acceptor.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous accept operations will be cancelled immediately.
>
> A subsequent call to open() is required before the acceptor can again be used to again perform socket accept operations.

`cancel(self)`

Cancel all asynchronous operations associated with the acceptor.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and

> receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, address_family: "v4"|"v6"|ip.address, fd: file_descriptor)`

Assign an existing native acceptor to `self`.

`address_family` can be either `"v4"` or `"v6"`. If you provide an `ip.address` object, the appropriate value will be inferred.

`release(self) → file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous accept operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native acceptor is then transferred to the caller.

Properties

`is_open: boolean`

Whether the acceptor is open.

`local_address: ip.address`

The local address endpoint of the acceptor.

`local_port: integer`

The local port endpoint of the acceptor.

ip.tcp.socket

```lua
-- `socket_pair()` implementation is
-- left as an exercise for the reader
local a, b = socket_pair()

spawn(function()
    local buf = byte_span.new(1024)
    local nread = b:read_some(buf)
    print(buf:slice(1, nread))
end):detach()

local nwritten = stream.write_all(a, 'Hello World')
print(nwritten)
```

Functions

`new() → ip.tcp.socket`

Constructor.

`open(self, address_family: "v4"|"v6"|ip.address)`

Open the socket.

`address_family` can be either `"v4"` or `"v6"`. If you provide an `ip.address` object, the appropriate value will be inferred.

`bind(self, addr: ip.address|string, port: integer)`

Bind the socket to the given local endpoint.

`close(self)`

Close the socket.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `boost::asio::error::operation_aborted` error.

`cancel(self)`

Cancel all asynchronous operations associated with the acceptor.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, address_family: "v4"|"v6"|ip.address, fd: file_descriptor)`

Assign an existing native socket to `self`.

`address_family` can be either `"v4"` or `"v6"`. If you provide an `ip.address` object, the appropriate value will be inferred.

`release(self) → file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native socket is then transferred to the caller.

`io_control(self, command: string[, ⋯])`

Perform an IO control command on the socket.

Currently available commands are:

`"bytes_readable"`
: Expects no arguments. Get the amount of data that can be read without blocking. Implements the `FIONREAD` IO control command.

`shutdown(self, what: "receive"|"send"|"both")`

Disable sends or receives on the socket.

`what` can be one of the following:

`"receive"`
: Shutdown the receive side of the socket.

`"send"`
: Shutdown the send side of the socket.

`"both"`
: Shutdown both send and receive on the socket.

`connect(self, addr: ip.address, port: integer)`

Initiate a connect operation and blocks current fiber until it completes or errs.

`disconnect(self)`

Dissolve the socket's association by resetting the socket's peer address (i.e. connect(3) will be called with an `AF_UNSPEC` address).

`read_some(self, buffer: byte_span) → integer`

Read data from the stream socket and blocks current fiber until it completes or errs.

Returns the number of bytes read.

`write_some(self, buffer: byte_span) → integer`

Write data to the stream socket and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`receive(self, buffer: byte_span, flags: integer) → integer`

Read data from the stream socket and blocks current fiber until it completes or errs.

Returns the number of bytes read.

`flags` is `0` or an or-combination of values from ip.message_flag(3em).

`send(self, buffer: byte_span, flags: integer) → integer`

Write data to the stream socket and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`flags` is `0` or an or-combination of values from ip.message_flag(3em).

`send_file(self, file: file.random_access, offset: integer, size_in_bytes: integer[, head: byte_span[, tail: byte_span[, n_number_of_bytes_per_send: integer]]]) → integer`

A wrapper for the `TransmitFile()` function.

Only available on Windows.

Lua conventions on index starting at `1` are ignored. Indexes here are OS-mandated and start at `0`.

`wait(self, wait_type: "read"|"write"|"error")`

Wait for the socket to become ready to read, ready to write, or to have pending error conditions.

In short, the reactor model is exposed on top of the proactor model.

You shouldn't be using reactor-style operations on Emilua. However there's this one obsolete and buggy TCP feature that presumes reactor-style operations: `SO_OOBINLINE` (`out_of_band_inline`) + `sockatmark()` (`at_mark`). If you're implementing an ancient obscure protocol that for some reason can avoid the TCP OOB bugs then you'll need to use this function.

`wait_type` can be one of the following:

`"read"`
: Wait for a socket to become ready to read.

`"write"`
: Wait for a socket to become ready to write.

`"error"`
: Wait for a socket to have error conditions pending.

`set_option(self, opt: string, val)`

Set an option on the socket.

Currently available options are:

`"tcp_no_delay"`
: Check Boost.Asio documentation.

`"send_low_watermark"`
: Check Boost.Asio documentation.

`"send_buffer_size"`
: Check Boost.Asio documentation.

`"receive_low_watermark"`
: Check Boost.Asio documentation.

`"receive_buffer_size"`
: Check Boost.Asio documentation.

`"out_of_band_inline"`
: Socket option for putting received out-of-band data inline.

Do bear in mind that the BSD socket API for `SO_OOBINLINE` is incompatible with proactor-style operations.

`"linger"`
: Check Boost.Asio documentation.

`"keep_alive"`
: Check Boost.Asio documentation.

`"do_not_route"`
: Check Boost.Asio documentation.

`"debug"`
: Check Boost.Asio documentation.

`"v6_only"`
: Check Boost.Asio documentation.

`get_option(self, opt: string) → value`

Get an option from the socket.

Currently available options are:

`"tcp_no_delay"`
: Check Boost.Asio documentation.

`"send_low_watermark"`
: Check Boost.Asio documentation.

`"send_buffer_size"`
: Check Boost.Asio documentation.

`"receive_low_watermark"`
: Check Boost.Asio documentation.

`"receive_buffer_size"`
: Check Boost.Asio documentation.

`"out_of_band_inline"`
: Check Boost.Asio documentation.

`"linger"`
: Check Boost.Asio documentation.

`"keep_alive"`
: Check Boost.Asio documentation.

`"do_not_route"`
: Check Boost.Asio documentation.

`"debug"`
: Check Boost.Asio documentation.

`"v6_only"`

Check Boost.Asio documentation.

Properties

`is_open: boolean`

Whether the socket is open.

`local_address: ip.address`

The local address endpoint of the socket.

`local_port: integer`

The local port endpoint of the socket.

`remote_address: ip.address`

The remote address endpoint of the socket.

`remote_port: integer`

The remote port endpoint of the socket.

`at_mark: boolean`

Whether the socket is at the out-of-band data mark.

You must set the `out_of_band_inline` socket option and use reactor-style operations (`wait()`) to use this feature.

ip.udp.socket

```lua
local sock = ip.udp.socket.new()
sock.open('v4')
sock:bind(ip.address.any_v4(), 1234)

local buf = byte_span.new(1024)
local nread, remote_addr, remote_port = sock:receive_from(buf)
sock:send_to(buf:slice(1, nread), remote_addr, remote_port)
```

Functions

new() → ip.udp.socket

Constructor.

open(self, address_family: "v4"|"v6"|ip.address)

Open the socket.

`address_family` can be either `"v4"` or `"v6"`. If you provide an `ip.address` object, the appropriate value will be inferred.

bind(self, addr: ip.address|string, port: integer)

Bind the socket to the given local endpoint.

shutdown(self, what: "receive"|"send"|"both")

Disable sends or receives on the socket.

`what` can be one of the following:

`"receive"`
: Shutdown the receive side of the socket.

`"send"`
: Shutdown the send side of the socket.

`"both"`
: Shutdown both send and receive on the socket.

Doing this only mutates the socket object, but nothing will be sent over the wire. It could be useful if you're planning to send the FD around to other processes.

connect(self, addr: ip.address, port: integer)

Set the default destination address so datagrams can be sent using `send()` without specifying a

destination address.

`disconnect(self)`

Dissolve the socket's association by resetting the socket's peer address (i.e. connect(3) will be called with an `AF_UNSPEC` address).

`close(self)`

Close the socket.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `boost::asio::error::operation_aborted` error.

`cancel(self)`

Cancel all asynchronous operations associated with the acceptor.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, address_family: "v4"|"v6"|ip.address, fd: file_descriptor)`

Assign an existing native socket to `self`.

`address_family` can be either `"v4"` or `"v6"`. If you provide an `ip.address` object, the appropriate value will be inferred.

`release(self) → file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native socket is then transferred to the caller.

`receive(self, buffer: byte_span[, flags: integer]) → integer`

Receive a datagram and blocks current fiber until it completes or errs.

Returns the number of bytes read.

`flags` is `0` or an or-combination of values from ip.message_flag(3em).

`receive_from(self, buffer: byte_span[, flags: integer]) → integer, ip.address, integer`

Receive a datagram and blocks current fiber until it completes or errs.

Returns the number of bytes read plus the endpoint (address + port) of the remote sender of the datagram.

`flags` is `0` or an or-combination of values from ip.message_flag(3em).

`send(self, buffer: byte_span[, flags: integer]) → integer`

Send data on the datagram socket and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`flags` is `0` or an or-combination of values from ip.message_flag(3em).

The `send` operation can only be used with a connected socket. Use the `send_to` function to send data on an unconnected datagram socket.

`send_to(self, buffer: byte_span, remote_addr: ip.address, remote_port: integer[, flags: integer]) → integer`

Send a datagram to the specified remote endpoint and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`flags` is `0` or an or-combination of values from ip.message_flag(3em).

`set_option(self, opt: string, val)`

Set an option on the socket.

Currently available options are:

`"debug"`
: Check Boost.Asio documentation.

`"broadcast"`
: Check Boost.Asio documentation.

`"do_not_route"`
: Check Boost.Asio documentation.

`"send_buffer_size"`
: Check Boost.Asio documentation.

`"receive_buffer_size"`
: Check Boost.Asio documentation.

`"reuse_address"`
: Check Boost.Asio documentation.

`"multicast_loop"`
: Check Boost.Asio documentation.

`"multicast_hops"`
: Check Boost.Asio documentation.

`"join_multicast_group"`
: Check Boost.Asio documentation.

`"leave_multicast_group"`
: Check Boost.Asio documentation.

`"multicast_interface"`
: Check Boost.Asio documentation.

`"unicast_hops"`
: Check Boost.Asio documentation.

`"v6_only"`
: Check Boost.Asio documentation.

`get_option(self, opt: string) → value`

Get an option from the socket.

Currently available options are:

`"debug"`
: Check Boost.Asio documentation.

`"broadcast"`
: Check Boost.Asio documentation.

`"do_not_route"`
: Check Boost.Asio documentation.

`"send_buffer_size"`
: Check Boost.Asio documentation.

`"receive_buffer_size"`

Check Boost.Asio documentation.

`"reuse_address"`

Check Boost.Asio documentation.

`"multicast_loop"`

Check Boost.Asio documentation.

`"multicast_hops"`

Check Boost.Asio documentation.

`"unicast_hops"`

Check Boost.Asio documentation.

`"v6_only"`

Check Boost.Asio documentation.

`io_control(self, command: string[, ···])`

Perform an IO control command on the socket.

Currently available commands are:

`"bytes_readable"`

Expects no arguments. Get the amount of data that can be read without blocking. Implements the `FIONREAD` IO control command.

Properties

`is_open: boolean`

Whether the socket is open.

`local_address: ip.address`

The local address endpoint of the socket.

`local_port: integer`

The local port endpoint of the socket.

`remote_address: ip.address`

The remote address endpoint of the socket.

`remote_port: integer`

The remote port endpoint of the socket.

json

```
-- Encoding

json.encode({'foo', {bar = {'baz', json.null, 1, 2}}})
--< '["foo", {"bar": ["baz", null, 1, 2]}]'
print(json.encode('\\'))
--< "\\"
print(json.encode({c = 0, b = 0, a = 0}))
--< {"a": 0, "b": 0, "c": 0}

-- Decoding

local obj = json.decode('["foo", {"bar":["baz", null, 1.0, 2]}]')
print(json.decode('"\\"foo\\bar"'))
```

Types

- json.writer(3em).

Constants

null: unspecified

The single object that represents the JSON null value.

It's safe to compare against this object to test for JSON's `null`.

If you call `tostring()` on this object, the string `"null"` will be returned.

lexer_ecat

It comes straight from the imported library and we don't really control the error codes.

dom_ecat

Errors from this category don't mean the textual JSON representation is invalid. Rather, conversion to/from lua value failed (e.g. number overflow would occur, nesting level too deep, cyclic references, ...).

This error category represents the very membrane between textual and Lua data representation.

Functions

decode(raw_json: string) → value

Deserialize `raw_json` to a lua value.

```
local json_str = '{"items":[],"properties":{}}'
print(json_str)
print(json.encode(json.decode(json_str)))
```

will output (do note that order is unspecified and might change from emilua version to version):

```
{"items":[],"properties":{}}
{"properties":{},"items":[]}
```

encode(value[, opts: table]) → string

Serialize `value` to a JSON formatted string.

```
print(json.encode(json.null))
print(json.encode({hello = 'world', what = json.null,
                   animals = {'cow', 'coelho'}}))
print(json.encode(json.into_array()))
print(json.encode('hey "pretty"'))
```

will output:

```
null
{"what":null,"hello":"world","animals":["cow","coelho"]}
[]
"hey \"pretty\""
```

If `value` (or any nested element) has a `__tojson()` metamethod, it'll be used to serialize that nesting level. Check `__tojson()` below to see parameters documentation.

`opts` is an options table that might contain the following fields:

- `indent`: the indentation string (or `nil` if a compact ugly JSON is desired).
- `state`: the `state` object passed in the `__tojson()` call. Useful to serialize further subobjects from the metamethod site. This option overrides other options in the `opts` table.

If called with `state`, `encode()` will **NOT** return the generated string as it expects to write a partial value using `state.writer` only.

is_array(json: value) → boolean

Test if `json` is a lua table and it has been tagged using the `json.into_array()` function to indicate that it represents a JSON array.

Example

```
local raw_json = [[ ["test", 4, false] ]]

function poor_print(value)
   if json.is_array(value) then
      print(unpack(value))
   elseif type(value) == 'table' then
      print('{')
      for k, v in pairs(value) do
         print('', '"' .. k .. '"', v)
      end
      print('}')
   elseif type(value) == 'string' then
      print('"' .. value .. '"')
   else
      print(value)
   end
end

poor_print(json.decode(raw_json))
```

`into_array([json: table]) → table`

Change `json`'s metatable to a certain tag that indicates either:

- The associated table was created from the result of parsing a JSON array.
- If this table is used to generate JSON textual representation, it should be encoded as a JSON array.

`json` is returned from this function to favour certain useful syntactic idioms.

If called with no arguments, a new array is created and returned.

Use `json.is_array()` to check if some value has been marked using this function.

Customization point metamethods

`__tojson(self, state)`

Called to write current node in the JSON tree.

`state` is a table with the following fields:

- `writer`: the generator.
- `visited`: a table to detect reference cycles. Before serializing a suboject, check whether `visited` already contains the to-be-serialized table. If a cycle is detected, raise `cycle_exists` error. If all is good, set `visited[t] = true` *before* calling `getmetatable(t).__tojson(t, state)` on the subobject `t`.

- `indent`: the indentation string (or `nil` if a compact ugly JSON is desired). Current level of nested containers can be queried through `writer`, so you should write this string as many times as this reported level.

A trick to avoid the error-prone interactions involving `state` (e.g. updating `visited`, etc) to serialize subobjects is to call `json.encode(t, { state = state })` on the subobject `t`. This way, you move the responsibility away to the `json` module itself. Example:

```
-- NOTE: this example ignores `indent`
mt = {
    __tojson = function(o, state)
        local writer = state.writer

        writer:begin_object()
        writer:value('foo')
        writer:value(o.foo) --< a number
        writer:value('bar')

        -- a subobject
        -- might contain its own `__tojson()`
        json.encode(o.bar, { state = state })

        writer:end_object()
    end
}
```

Conversion table

Lua type	JSON type	Notes
`json.null`	`null`	
boolean	boolean	
number	number	
string	string	

Lua type	JSON type	Notes
table	array	On `decode(raw_json)`: • The lua table is marked with the `json.into_array()` function. On `encode(lua_obj)`: • `lua_obj` is encoded as a JSON array if it has been marked as so using `json.into_array()` or `#lua_obj` evaluates to a value larger than `0`. • Non-integer keys are ignored.
table	object	On `encode(lua_obj)`: • Non-string keys are ignored.

Rationale

These choices are also used by other lua libraries in the wild.

David Heiko Kolf's work on collecting and comparing JSON libraries for Lua, and generally documenting common pitfalls as well, was specially helpful. Thanks to his work it was much easier for me to design my own solution.

`null`

Encoding the JSON `null` value is a problem. Lua treats `nil` as indistinguishable from an absent value so we can't really map `null` to `nil`. This problem only gets worse when interactions with sparse tables begin. However, JavaScript uses a different value for absent, `undefined`. And the same solution is chosen here with the introduction of a `json.null` value.

JSON arrays

JSON arrays and JSON objects will map to the same type — lua tables. How do we differentiate them? This problem isn't exclusive to Lua. JavaScript itself suffers from this problem:

```
> typeof({})
'object'
> typeof([])
'object'
```

The solution chosen by JavaScript is an `Array.isArray()` function:

```
> Array.isArray({})
false
> Array.isArray([])
true
```

Therefore the same solution is chosen here:

```
local value = json.decode(raw_json)
if json.is_array(value) then
  -- ...
end
```

And `json.into_aray()` is introduced to make certain patterns easier to work with (especially for the `encode()` function).

I acknowledge that dkjson's `__jsontype` metafield is more general, but JSON doesn't really need this kind of generality. JSON is a closed world.

encode()

The following libraries and pages inspired this function:

- Section "value type mappings" from lua-rapidjson homepage.
- Section "handling of empty arrays" from lua-users wiki's JSON Modules page.
- Section "examples" from dkjson homepage.

The `decode()` function avoids a recursive implementation. However, the `encode()` function does **not** share the same property. The reason why no effort was made to offer a recursion-free `encode()` implementation is the `__tojson()` metamethod. This metamethod would force an unbounded call-stack anyway, so there is no point. However, the recursion was implemented in lua bytecode, so at least your process shouldn't crash on stack overflow. If you wish for a recursion-free implementation, you can use the generator interface directly and avoid `__tojson()` yourself.

json.writer

The JSON incremental generator. It keeps track of the context and inserts the appropriate separators between values where needed.

Functions

`new() → json.writer`

The constructor.

`value(self, v)`

Write formatted leaf value `v` into the JSON output.

`v` can be one of the following types:

- `boolean`
- `number`
- `string`
- `json.null`

`begin_object(self)`

Write the begin-object token to initiate an object into the JSON output.

`end_object(self)`

Write the end-object token to terminate an object into the JSON output.

`begin_array(self)`

Write the begin-array token to initiate an array into the JSON output.

`end_array(self)`

Write the end-array token to terminate an array into the JSON output.

`literal(self, raw: string)`

Write a literal value directly into the JSON output without formatting it.

`generate(self) → string`

Returns the generated JSON and consumes `self`. After this call, `self` can no longer be used.

Attributes

`level: integer`

The current level of nested containers.

mutex

```
local mutex = require('mutex')

local function ping_sender()
    sleep(30)
    scope(function()
        scope_cleanup_push(function() ws_write_mtx:unlock() end)
        ws_write_mtx:lock()
        ws:ping()
    end)
end

local function queue_consumer()
    scope(function()
        scope_cleanup_push(function() queue_mtx:unlock() end)
        queue_mtx:lock()
        while #queue == 0 do
            queue_cond:wait(queue_mtx)
        end
        for _, e in ipairs(queue) do
            consume_item(e)
        end
        queue = {}
    end)
end
```

A mutex.

Functions

new() → mutex

Constructor.

lock(self)

Locks the mutex.

This suspending function does **not** act as an interruption point.

This mutex applies dispatch semantics. That means no context switch to other ready fibers will take place if it's possible to acquire the mutex immediately.

try_lock(self) → boolean

Tries to lock the mutex. Returns whether lock acquisition was successful.

It's an error to call `try_lock()` if current fiber already owns the mutex (cf. recursive_mutex(3em) for an alternative).

The current fiber is never suspended.

`unlock(self)`

Unlocks the mutex.

recursive_mutex

A recursive mutex.

A fiber that already has exclusive ownership of a given `recursive_mutex` instance can call `lock()` or `try_lock()` to acquire an additional level of ownership of the mutex. `unlock()` must be called once for each level of ownership acquired by a single fiber before ownership can be acquired by another fiber.

Functions

`new() → recursive_mutex`

Constructor.

`lock(self)`

Locks the mutex.

This suspending function does **not** act as an interruption point.

This mutex applies dispatch semantics. That means no context switch to other ready fibers will take place if it's possible to acquire the mutex immediately.

`try_lock(self) → boolean`

Tries to lock the mutex. Returns whether lock acquisition was successful.

The current fiber is never suspended.

`unlock(self)`

Unlocks the mutex.

future

Futures and promises.

This implementation follows the model of shared futures. Thus multiple waiters on the same future are allowed.

Functions

`new() → promise, future`

Constructor.

Creates a promise and its associated future and returns them.

`future` functions

`get(self) → value`

If result is available, returns result. Otherwise, blocks current fiber until result is ready and returns it.

`promise` functions

`set_value(self, v)`

Atomically stores the value into the shared state and makes the state ready.

`set_error(self, e)`

Atomically stores the exception `e` into the shared state and makes the state ready.

pipe.read_stream

Functions

`new() → pipe.read_stream`

```
new()                    ①
new(fd: file_descriptor) ②
```

① Default constructor.

② Converts a file descriptor into a pipe end.

`close(self)`

Close the pipe.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous read operations will be cancelled immediately, and will complete with the `boost::asio::error::operation_aborted` error.

`cancel(self)`

Cancel all asynchronous operations associated with the pipe.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous read operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, fd: file_descriptor)`

Assign an existing native pipe to `self`.

`release(self) → file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous read operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native pipe is then transferred to the caller.

`read_some(self, buffer: byte_span) → integer`

Read data from the pipe and blocks current fiber until it completes or errs.

Returns the number of bytes read.

Properties

`is_open: boolean`

Whether the pipe is open.

pipe.write_stream

Functions

`new() → pipe.write_stream`

```
new()                    ①
new(fd: file_descriptor) ②
```

① Default constructor.

② Converts a file descriptor into a pipe end.

`close(self)`

Close the pipe.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous write operations will be cancelled immediately, and will complete with the `boost::asio::error::operation_aborted` error.

`cancel(self)`

Cancel all asynchronous operations associated with the pipe.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous write operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, fd: file_descriptor)`

Assign an existing native pipe to `self`.

`release(self) → file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous write operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native pipe is then transferred to the caller.

`write_some(self, buffer: byte_span) → integer`

Write data to the pipe and blocks current fiber until it completes or errs.

Returns the number of bytes written.

Properties

`is_open: boolean`

Whether the pipe is open.

pipe.pair

Synopsis

```
local pipe = require "pipe"
pipe.pair() -> pipe.read_stream, pipe.write_stream
```

Description

Creates a pipe.

regex

Types

`regex`

Functions

`new(options: table) → regex`

Constructor.

`options`

`pattern: string`

The pattern to match against.

`grammar`

The grammar.

Currently it has support for:

- `"basic"`.
- `"extended"`.
- `"ecma"`.

`ignore_case: boolean = false`

Whether to ignore casing.

`nosubs: boolean = false`

When performing matches, all marked sub-expressions are treated as non-marking sub-expressions.

`optimize: boolean = false`

Whether to optimize the regex.

Functions

`match(re: regex, str: string|byte_span) → matches…`

Try to match the pattern against the whole string `str`. If successful, then returns the captures from the pattern; otherwise it returns `nil`. If `re` specifies no captures, then the whole match is returned.

`search(re: regex, str: string|byte_span) → table`

Scan through `str` looking for the first location where the regular expression pattern produces a match, and return a corresponding match object. The returned table contains the following string keys:

`"empty": boolean`

Whether match was unsuccessful.

The table also contains numeric keys from `0` to the number of specified capture groups. `0` will represent the whole match and subsequent indexes are present if a corresponding match for that capturing group was found. Each element will be a table with the following members:

`"start": integer`

The index for the first character that matched.

`"end_": integer`

The index for the last character that matched.

`split(re: regex, str: string|byte_span) → string[]|byte_span[]`

Split `str` by the occurrences of `re`.

`patsplit(re: regex, str: string|byte_span) → string[]|byte_span[]`

Returns occurrences of `re` in `str`.

serial_port

```
local port = serial_port.new()
port:open(name)
```

Functions

new() → serial_port

```
new()                    ①
new(fd: file_descriptor) ②
```

① Default constructor.

② Converts a file descriptor into a `serial_port` object.

ptypair() → serial_port, file_descriptor

Open a pair of connected pseudoterminal devices. Returns the master and the slave ends, respectively.

The flag `O_NOCTTY` will be used to open the slave end so it doesn't accidentally become the controlling terminal for the session of the calling process.

Use the returned `file_descriptor` object in `system.spawn()`'s `set_ctty`.

open(self, device: string)

Open the serial port using the specified device name.

`device` is something like `"COM1"` on Windows, and `"/dev/ttyS0"` on POSIX platforms.

close(self)

Close the port.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous read or write operations will be cancelled immediately, and will complete with the `boost::asio::error::operation_aborted` error.

cancel(self)

Cancel all asynchronous operations associated with the acceptor.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous read or write operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, fd: file_descriptor)`

Assign an existing native port to `self`.

`release(self) → file_descriptor`

Release ownership of the native descriptor implementation.

`send_break(self)`

Send a break sequence to the serial port.

This function causes a break sequence of platform-specific duration to be sent out the serial port.

`read_some(self, buffer: byte_span) → integer`

Read data from the port and blocks current fiber until it completes or errs.

Returns the number of bytes read.

`write_some(self, buffer: byte_span) → integer`

Write data to the port and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`isatty(self) → boolean`

See isatty(3).

`tcgetpgrp(self) → integer`

See tcgetpgrp(3).

`tcsetpgrp(self, pgid_id: integer)`

See tcsetpgrp(3).

Properties

`is_open: boolean`

Whether the port is open.

`baud_rate: integer`

Read or write current baud rate setting.

`flow_control: "software"|"hardware"|nil`

Read or write current flow control setting.

`parity: "odd"|"even"|nil`

Read or write current parity setting.

`stop_bits: string`

Read or write current stop bit width setting.

It can be one of:

- `"one"`.
- `"one_point_five"`.
- `"two"`.

`character_size: integer`

Read or write current character size setting.

time.sleep

Synopsis

```
local time = require "time"
time.sleep(secs: number)
```

Description

Blocks the fiber until `secs` seconds have passed.

Floating point numbers give room for subsecond precision.

time.steady_clock

```lua
local clock = require('time').steady_clock
local timepoint = clock.now()
```

A monotonic clock (i.e. its time points cannot decrease as physical time moves forward).

Functions

now() → steady_clock.time_point

Returns a new time point representing the current value of the clock.

epoch() → steady_clock.time_point

Returns a new time point representing the epoch of the clock.

time_point functions

add(self, secs: number)

Modifies the time point by the given duration.

When the duration is converted to the native tick representation of the clock, it'll be rounded to the nearest time point rounding to even in halfway cases.

sub(self, secs: number)

Modifies the time point by the given duration.

When the duration is converted to the native tick representation of the clock, it'll be rounded to the nearest time point rounding to even in halfway cases.

time_point properties

seconds_since_epoch: number

The number of elapsed seconds since the clock's epoch.

time_point metamethods

- __add()
- __sub()
- __eq()

- `__lt()`
- `__le()`

time.steady_timer

```
local timer = require('time').steady_timer
local t = timer.new()

spawn(function() print('Hello') end)

t:expires_after(2) --< 2 seconds
t:wait()
print('World')
```

A monotonic timer (i.e. the time points of the underlying clock cannot decrease as physical time moves forward). As in Boost.Asio:

> A waitable timer is always in one of two states: "expired" or "not expired". If the `wait()` or `async_wait()` function is called on an expired timer, the wait operation will complete immediately.
>
> *Changing an active waitable timer's expiry time*
>
> Changing the expiry time of a timer while there are pending asynchronous waits causes those wait operations to be cancelled.

Functions

`new() → steady_timer`

```
local t = steady_timer.new()
```

Constructor. Returns a new `steady_timer` object.

`expires_at(self, tp: time.steady_clock.time_point) → integer`

Forward the call to the function with same name in Boost.Asio:

> Set the timer's expiry time as an absolute time. Any pending asynchronous wait operations will be cancelled. The handler for each cancelled operation will be invoked with the `boost::asio::error::operation_aborted` error code.
>
> *Return Value*
>
> The number of asynchronous operations that were cancelled.

`expires_after(self, secs: number) → integer`

Forward the call to the function with same name in Boost.Asio:

> Set the timer's expiry time relative to now. Any pending asynchronous wait operations will be cancelled. The handler for each cancelled operation will be invoked with the `boost::asio::error::operation_aborted` error code.
>
> *Return Value*
>
> The number of asynchronous operations that were cancelled.

Expiry time is given in seconds.

`wait(self)`

Initiate a wait operation on the timer and blocks current fiber until one of the events occur:

- The timer has expired.
- The timer was cancelled, in which case it raises `boost::asio::error::operation_aborted`.

`cancel(self) → integer`

Cancel any operations that are waiting on the timer. Returns the number of asynchronous operations that were cancelled.

Properties

`expiry: time.steady_clock.time_point`

The timer's expiry time as an absolute time.

Whether the timer has expired or not does not affect this value.

time.system_clock

```
local clock = require('time').system_clock
local timepoint = clock.now()
```

The system-wide real time wall clock. It uses the UNIX epoch.

On most systems, the system time can be adjusted at any moment.

Functions

now() → system_clock.time_point

Returns a new time point representing the current value of the clock.

epoch() → system_clock.time_point

Returns a new time point representing the epoch of the clock.

time_point functions

add(self, secs: number)

Modifies the time point by the given duration.

When the duration is converted to the native tick representation of the clock, it'll be rounded to the nearest time point rounding to even in halfway cases.

sub(self, secs: number)

Modifies the time point by the given duration.

When the duration is converted to the native tick representation of the clock, it'll be rounded to the nearest time point rounding to even in halfway cases.

time_point properties

seconds_since_epoch: number

The number of elapsed seconds since 1 January 1970, not counting leap seconds.

time_point metamethods

- __add()
- __sub()

- `__eq()`
- `__lt()`
- `__le()`

time.system_timer

```
local timer = require('time').system_timer
local t = timer.new()
```

A timer for the system_clock. As in Boost.Asio:

> A waitable timer is always in one of two states: "expired" or "not expired". If the `wait()` or `async_wait()` function is called on an expired timer, the wait operation will complete immediately.
>
> *Changing an active waitable timer's expiry time*
>
> Changing the expiry time of a timer while there are pending asynchronous waits causes those wait operations to be cancelled.

Functions

`new() → system_timer`

```
local t = system_timer.new()
```

Constructor. Returns a new `system_timer` object.

`expires_at(self, tp: time.system_clock.time_point) → integer`

Forward the call to the function with same name in Boost.Asio:

> Set the timer's expiry time as an absolute time. Any pending asynchronous wait operations will be cancelled. The handler for each cancelled operation will be invoked with the `boost::asio::error::operation_aborted` error code.
>
> *Return Value*
>
> The number of asynchronous operations that were cancelled.

`wait(self)`

Initiate a wait operation on the timer and blocks current fiber until one of the events occur:

- The timer has expired.
- The timer was cancelled, in which case it raises `boost::asio::error::operation_aborted`.

`cancel(self) → integer`

Cancel any operations that are waiting on the timer. Returns the number of asynchronous operations that were cancelled.

Properties

`expiry: time.system_clock.time_point`

The timer's expiry time as an absolute time.

Whether the timer has expired or not does not affect this value.

time.high_resolution_clock

```
local clock = require('time').high_resolution_clock
local timepoint = clock.now()
```

The clock with the smallest tick period provided by the system.

This clock is useful for microbenchmarking purposes.

Functions

now() → high_resolution_clock.time_point

Returns a new time point representing the current value of the clock.

epoch() → high_resolution_clock.time_point

Returns a new time point representing the epoch of the clock.

Attributes

is_steady: boolean

Whether the time between ticks is always constant (i.e. calls to `now()` return values that increase monotonically even in case of some external clock adjustment).

time_point properties

seconds_since_epoch: number

The number of elapsed seconds since the clock's epoch.

time_point metamethods

- `__sub()`
- `__eq()`
- `__lt()`
- `__le()`

spawn

Synopsis

```
spawn(f: function) -> fiber
```

Description

Spawns a new fiber to run `f`. Post semantics are used, so the current fiber (the one calling `spawn()`) continues to run until it reaches a suspension point.

Fibers are the primitive of choice to represent concurrency. Every time you need to increase the concurrency level, just spawn a fiber. Fibers are cooperative and only transfer control to other fibers in well-defined points (sync primitives, IO functions and any suspending function such as `this_fiber.yield()`). These points are also used by the interruption API.

No two fibers from the same Lua VM run in parallel (even when the underlying VM's thread pool has threads available).

`spawn()` is a global so it doesn't need to be `require()`d.

`fiber` functions

`join(self)`

Read `pthread_join()`.

Returns the values returned by the fiber's start function. If that fiber exits with an error, that error is re-raised here (and fiber is considered joined).

`detach(self)`

Read `pthread_detach()`.

If the GC collects the fiber handle, it'll be detached.

`interrupt(self)`

Read `pthread_cancel()`.

`fiber` properties

`interruption_caught: boolean`

Read `PTHREAD_CANCELED`.

`joinable: boolean`

Whether joinable.

this_fiber

Object referring to current fiber.

`this_fiber` is a global so it doesn't need to be `require()`d.

Functions

`yield()`

Reschedule current fiber to be executed in the next round so other ready fibers have a chance to run now. You usually don't need to call this function as any suspending function already do that.

`{forbid,allow}_suspend()`

```
forbid_suspend()
allow_suspend()
```

A call to `forbid_suspend()` will put the fiber in the state of *suspension-disallowed* and any attempt to suspend the fiber while it is in this state will raise an error.

`forbid_suspend()` may be called multiple times. A matching number of calls to `allow_suspend()` will put the fiber out of the *suspension-disallowed* state. You must not call `allow_suspend()` if there was no prior call to `forbid_suspend()`.

These functions aren't generally useful and they would have no purpose in preemptive multitasking. However a cooperative multitasking environment offers opportunities to avoid some round-trips to sync primitives. These opportunities shouldn't really be used and the programmer should just rely on the classical sync primitives. However I can't tame every wild programmer out there so there is this mechanism to at least document the code in mechanisms similar to `assert()` statements from native languages.

They're only useful if there are comprehensive test cases. Still, the use of these functions may make the code more readable. And some tools may be developed to understand these blocks and do some simple static analysis.

`this_fiber.{disable,restore}_interruption()`

```
disable_interruption()
restore_interruption()
```

Check the interruption tutorial to see what it does.

Properties

`is_main: boolean`

Whether this is the main fiber of the program.

`local_: table`

Fiber-local storage.

`id: string`

An id string for debugging purposes.

Use it **only** for debugging purposes. Do not exploit this value to create messy workarounds. There is no need to use it beyond anything other than debugging purposes.

inbox

Synopsis

```
local inbox = require "inbox"
```

Description

Returns the inbox associated with the caller VM.

Methods

`receive(self) → value`

Receives a message.

`close(self)`

Closes the channel. No further messages can be received after inbox is closed.

If `inbox` is not imported by the time the main fiber finishes execution, it's automatically closed.

spawn_vm

Synopsis

```
spawn_vm(module: string) -> channel
spawn_vm(opts: table) -> channel
```

Description

Creates a new actor and returns a tx-channel.

The new actor will execute with `_CONTEXT='worker'` (this `_CONTEXT` is not propagated to imported submodules within the actor).

Threading with work-stealing

Spawn more VMs than threads and spawn them all in the same thread-pool. The system will transparently steal VMs from the shared pool to keep the work-queue somewhat fair between the threads.

Threading with load-balancing

Spawn each VM in a new thread pool and make sure each-one has only one thread. Now use messaging to apply some load-balancing strategy of your choice.

Parameters

`module: string`

The module that will serve as the entry point for the new actor.

For IPC-based actors, this argument no longer means an actual module when Linux namespaces are involved. It'll just be passed along to the new process.

`'.'` is also a valid module to use when you spawn actors.

`inherit_context: boolean = true`

Whether to inherit the thread pool of the parent VM (i.e. the one calling `spawn_vm()`). On `false`, a new thread pool (starting with `1` thread) is created to run the new actor.

Emilua can handle multiple VMs running on the same thread just fine. Cooperative multitasking is used to alternate execution among the ready VMs.

A thread pool is one type of an execution context. The API prefers the term "context" as it's more general than "thread pool".

`concurrency_hint: integer|"safe" = "safe"`

`integer`

A suggestion to the new thread pool (`inherit_context` should be `false`) as to the number of active threads that should be used for scheduling actors[1].

You still need to call `spawn_context_threads()` to create the extra threads.

`"safe"`

The default. No assumption is made upfront on the number of active threads that will be created through `spawn_context_threads()`.

`new_master: boolean = false`

The first VM (actor) to run in a process has different responsibilities as that's the VM that will spawn all other actors in the system. The Emilua runtime will restrict modification of global process resources that don't play nice with threads such as the current working directory and signal handling disposition to this VM.

Upon spawning a new actor, it's possible to transfer ownership over these resources to the new VM. After `spawn_vm()` returns, the calling actor ceases to be the master VM in the process and can no longer recover its previous role as the master VM.

`subprocess: table|nil`

`table`

Spawn the actor in a new subprocess.

Not available on Windows.

`nil`

Default. Don't spawn the actor in a new subprocess.

`subprocess.newns_uts: boolean = false`

Whether to create the process within a new Linux UTS namespace.

`subprocess.newns_ipc: boolean = false`

Whether to create the process within a new Linux IPC namespace.

`subprocess.newns_pid: boolean = false`

Whether to create the process within a new Linux PID namespace.

The first process in a PID namespace is PID1 within that namespace. PID1 has a few special responsibilities. After `subprocess.init.script` exits, the Emilua runtime will fork if it's running as PID1. This new child will assume the role of starting your module (the Lua VM). The PID1 process will perform the following jobs:

- Forward `SIGTERM`, `SIGUSR1`, `SIGUSR2`, `SIGHUP`, and `SIGINT` to the child. There is no point in re-routing every signal, but more may be added to this set if you present a compelling case.
- Reap zombie processes.

- Exit when the child dies with the same exit code as the child's.

`subprocess.newns_user: boolean = false`

Whether to create the process within a new Linux user namespace.

Even if it's a sandbox, and root inside the sandbox doesn't mean root outside it, maybe you still want to drop all root privileges at the end of the `init.script`:

```
C.cap_set_proc('=')
```

It won't be particularly useful for most people, but that technique is still useful to — for instance — create alternative LXC/FlatPak front-ends to run a few programs (if the program can't update its own binary files, new possibilities for sandboxing practice open up).

`subprocess.newns_net: boolean = false`

Whether to create the process within a new Linux net namespace.

`subprocess.newns_mount: boolean = false`

Whether to create the process within a new Linux mount namespace.

`subprocess.environment: { [string] = string }|nil`

A table of strings that will be used as the created process' `envp`. On `nil`, an empty `envp` will be used.

`subprocess.stdin,stdout,stderr: "share"|file_descriptor|nil`

`"share"`

The spawned process will share the specified standard handle (`stdin`, `stdout`, or `stderr`) with the caller process.

`file_descriptor`

Use the file descriptor as the specified standard handle (`stdin`, `stdout`, or `stderr`) for the spawned process.

`nil`

Create and use a closed pipe end as the specified standard handle (`stdin`, `stdout`, or `stderr`) for the spawned process.

`subprocess.init.script: string`

The source code for a script that is used to initialize the sandbox in the child process.

`errexit`

We don't want to accidentally ignore errors from the C API exposed to the `init.script`. That's why we borrow an idea from BASH. One common folklore among BASH programmers is the unofficial strict mode. Among other things, this mode dictates the use of BASH's `set -o errexit`.

And `errexit` exists for the `init.script` as well. For `init.script`, `errexit` is just a global boolean. Every time the C API fails, the Emilua wrapper for the function will check its value. On `errexit=true` (the default when the script starts), the process will abort whenever some C API fails. That's specially important when you're using the API to drop process credentials/rights.

The controlling terminal

The Emilua runtime won't call `setsid()` nor `setpgid()` by itself, so the process will stay in the same session as its parent, and it'll have access to the same controlling terminal.

If you want to block the new actor from accessing the controlling terminal, you may perform the usual calls in `init.script`:

```
C.setsid()
C.setpgid(0, 0)
```

`subprocess.init.arg: file_descriptor`

A file descriptor that will be sent to the `init.script`. The script can access this fd through the variable `arg` that is available within the script.

`channel` functions

`send(self, msg)`

Sends a message.

You can send the address of other actors (or self) by sending the channel as a message. A clone of the tx-channel will be made and sent over.

This simple foundation is enough to:

> [...] gives Actors the ability to create and participate in arbitrarily variable topological relationships with one another [...]
>
> — https://en.wikipedia.org/wiki/Actor_model

`close(self)`

Closes the channel. No further messages can be sent after a channel is closed.

`detach(self)`

Detaches the calling VM/actor from the role of supervisor for the process/actor represented by `self`. After this operation is done, the process/actor represented by `self` is allowed to outlive the calling

process.

The channel remains open.

This method is only available for channels associated with IPC-based actors that are direct children of the caller.

`kill(self, signo: integer = system.signal.SIGKILL)`

Sends `signo` to the subprocess. On `SIGKILL`, it'll also close the channel.

This method is only available for channels associated with IPC-based actors that are direct children of the caller.

A PID file descriptor is used to send `signo` so no races involving PID numbers ever happen.

`channel` properties

`child_pid: integer`

The process id used by the OS to represent this child process (e.g. the number that shows up in `/proc` on some UNIX systems).

Do keep in mind that process reaping happens automatically and the PID won't remain reserved once the child dies, so it's racy to use the PID. Even if process reaping was **not** automatic, it'd still be possible to have races if the parent died while some other process was using this PID. Use `child_pid` only as a last resort.

You can only access this field for channels associated with IPC-based actors that are direct children of the caller.

The C API exposed to `init.script`

Helpers

`mode(user: integer, group: integer, other: integer) → integer`

```
function mode(user, group, other)
    return bit.bor(bit.lshift(user, 6), bit.lshift(group, 3), other)
end
```

`receive_with_fd(fd: integer, buf_size: integer) → string, integer, integer`

Returns three values:

1. String with the received message (or `nil` on error).
2. File descriptor received (or `-1` on none).
3. The errno value (or `0` on success).

`send_with_fd(fd: integer, str: buffer, fd2: integer) → integer, integer`

Returns two values:

1. `sendmsg()` return.
2. The errno value (or `0` on success).

`set_no_new_privs()`

Set the calling thread's no_new_privs attribute to `true`.

Functions

These functions live inside the global table `C`. `errno` (or `0` on success) is returned as the second result.

- `read()`. Opposed to the C function, it receives two arguments. The second argument is the size of the buffer. The buffer is allocated automatically, and returned as a string in the first result (unless an error happens, then `nil` is returned).
- `write()`. Opposed to the C function, it receives two arguments. The second one is a string which will be used as the buffer.
- `sethostname()`. Opposed to the C function, it only receives the string argument.
- `setdomainname()`. Opposed to the C function, it only receives the string argument.
- `setgroups()`. Opposed to the C function, it receives a list of numbers as its single argument.
- `cap_set_proc()`. Opposed to the C function, it receives a string as its single argument. The string is converted to the `cap_t` type using the function `cap_from_text()`.
- `cap_drop_bound()`. Opposed to the C function, it receives a string as its single argument. The string is converted to the `cap_value_t` type using the function `cap_from_name()`.
- `cap_set_ambient()`. Opposed to the C function, it receives a string as its first argument. The string is converted to the `cap_value_t` type using the function `cap_from_name()`. The second parameter is a boolean.
- `execve()`. Opposed to the C function, `argv` and `envp` are specified as a Lua table.
- `fexecve()`. Opposed to the C function, `argv` and `envp` are specified as a Lua table.

Other exported functions work as usual (except that `errno` or `0` is returned as the second result):

- `open()`.
- `mkdir()`.
- `chdir()`.
- `link()`.
- `symlink()`.

- `chown()`.
- `chmod()`.
- `umask()`.
- `mount()`.
- `umount()`.
- `umount2()`.
- `pivot_root()`.
- `chroot()`.
- `setsid()`.
- `setpgid()`.
- `setresuid()`.
- `setresgid()`.
- `cap_reset_ambient()`.
- `cap_set_secbits()`.
- `unshare()`.
- `setns()`.
- `cap_enter()`.
- `jail_attach()`.

Constants

These constants live inside the global table `C`.

- `O_CLOEXEC`.
- `EAFNOSUPPORT`.
- `EADDRINUSE`.
- `EADDRNOTAVAIL`.
- `EISCONN`.
- `E2BIG`.
- `EDOM`.
- `EFAULT`.
- `EBADF`.
- `EBADMSG`.
- `EPIPE`.
- `ECONNABORTED`.
- `EALREADY`.
- `ECONNREFUSED`.

- ECONNRESET.
- EXDEV.
- EDESTADDRREQ.
- EBUSY.
- ENOTEMPTY.
- ENOEXEC.
- EEXIST.
- EFBIG.
- ENAMETOOLONG.
- ENOSYS.
- EHOSTUNREACH.
- EIDRM.
- EILSEQ.
- ENOTTY.
- EINTR.
- EINVAL.
- ESPIPE.
- EIO.
- EISDIR.
- EMSGSIZE.
- ENETDOWN.
- ENETRESET.
- ENETUNREACH.
- ENOBUFS.
- ECHILD.
- ENOLINK.
- ENOLCK.
- ENODATA.
- ENOMSG.
- ENOPROTOOPT.
- ENOSPC.
- ENOSR.
- ENXIO.
- ENODEV.
- ENOENT.

- ESRCH.
- ENOTDIR.
- ENOTSOCK.
- ENOSTR.
- ENOTCONN.
- ENOMEM.
- ENOTSUP.
- ECANCELED.
- EINPROGRESS.
- EPERM.
- EOPNOTSUPP.
- EWOULDBLOCK.
- EOWNERDEAD.
- EACCES.
- EPROTO.
- EPROTONOSUPPORT.
- EROFS.
- EDEADLK.
- EAGAIN.
- ERANGE.
- ENOTRECOVERABLE.
- ETIME.
- ETXTBSY.
- ETIMEDOUT.
- ENFILE.
- EMFILE.
- EMLINK.
- ELOOP.
- EOVERFLOW.
- EPROTOTYPE.
- O_CREAT.
- O_RDONLY.
- O_WRONLY.
- O_RDWR.
- O_DIRECTORY.

- O_EXCL.
- O_NOCTTY.
- O_NOFOLLOW.
- O_TMPFILE.
- O_TRUNC.
- O_APPEND.
- O_ASYNC.
- O_DIRECT.
- O_DSYNC.
- O_LARGEFILE.
- O_NOATIME.
- O_NONBLOCK.
- O_PATH.
- O_SYNC.
- S_IRWXU.
- S_IRUSR.
- S_IWUSR.
- S_IXUSR.
- S_IRWXG.
- S_IRGRP.
- S_IWGRP.
- S_IXGRP.
- S_IRWXO.
- S_IROTH.
- S_IWOTH.
- S_IXOTH.
- S_ISUID.
- S_ISGID.
- S_ISVTX.
- MS_REMOUNT.
- MS_BIND.
- MS_SHARED.
- MS_PRIVATE.
- MS_SLAVE.
- MS_UNBINDABLE.

- MS_MOVE.
- MS_DIRSYNC.
- MS_LAZYTIME.
- MS_MANDLOCK.
- MS_NOATIME.
- MS_NODEV.
- MS_NODIRATIME.
- MS_NOEXEC.
- MS_NOSUID.
- MS_RDONLY.
- MS_REC.
- MS_RELATIME.
- MS_SILENT.
- MS_STRICTATIME.
- MS_SYNCHRONOUS.
- MS_NOSYMFOLLOW.
- MNT_FORCE.
- MNT_DETACH.
- MNT_EXPIRE.
- UMOUNT_NOFOLLOW.
- CLONE_NEWCGROUP.
- CLONE_NEWIPC.
- CLONE_NEWNET.
- CLONE_NEWNS.
- CLONE_NEWPID.
- CLONE_NEWTIME.
- CLONE_NEWUSER.
- CLONE_NEWUTS.
- SECBIT_NOROOT.
- SECBIT_NOROOT_LOCKED.
- SECBIT_NO_SETUID_FIXUP.
- SECBIT_NO_SETUID_FIXUP_LOCKED.
- SECBIT_KEEP_CAPS.
- SECBIT_KEEP_CAPS_LOCKED.
- SECBIT_NO_CAP_AMBIENT_RAISE.

- `SECBIT_NO_CAP_AMBIENT_RAISE_LOCKED`.

`C.landlock_create_ruleset(attr: table|nil, flags: table|nil) → integer, integer`

Parameters:

- `attr.handled_access_fs: string[]`
  - `"execute"`
  - `"write_file"`
  - `"read_file"`
  - `"read_dir"`
  - `"remove_dir"`
  - `"remove_file"`
  - `"make_char"`
  - `"make_dir"`
  - `"make_reg"`
  - `"make_sock"`
  - `"make_fifo"`
  - `"make_block"`
  - `"make_sym"`
  - `"refer"`
  - `"truncate"`
- `flags: string[]`
  - `"version"`

Returns two values:

1. `landlock_create_ruleset()` return.
2. The errno value (or `0` on success).

`C.landlock_add_rule(ruleset_fd: integer, rule_type: "path_beneath", attr: table) → integer, integer`

Parameters:

- `attr.allowed_access: string[]`
  - `"execute"`
  - `"write_file"`
  - `"read_file"`
  - `"read_dir"`

    - `"remove_dir"`
    - `"remove_file"`
    - `"make_char"`
    - `"make_dir"`
    - `"make_reg"`
    - `"make_sock"`
    - `"make_fifo"`
    - `"make_block"`
    - `"make_sym"`
    - `"refer"`
    - `"truncate"`
- `attr.parent_fd: integer`

Returns two values:

1. `landlock_add_rule()` return.
2. The errno value (or `0` on success).

`C.landlock_restrict_self(ruleset_fd: integer) → integer, integer`

Returns two values:

1. `landlock_restrict_self()` return.
2. The errno value (or `0` on success).

`C.jail_set(params: { [string]: string|boolean }, flags: string[]|nil) → integer, integer`

Create or modify a jail. Optionally locks the current process in it.

Jail parameters are given as strings and they'll be transparently converted to the native format accepted by the kernel.

`flags` may contain the following values:

- `"create"`
- `"update"`
- `"attach"`
- `"dying"`

See jail(8) for more information on the core jail parameters.

[1] https://www.boost.org/doc/libs/1_69_0/doc/html/boost_asio/overview/core/concurrency_hint.html

spawn_context_threads

Synopsis

```
spawn_context_threads(count: integer)
```

Description

Spawns extra `count` threads to the thread pool of the caller VM.

Emilua can handle multiple VMs running on the same thread just fine. Cooperative multitasking is used to alternate execution among the ready VMs.

It doesn't make sense to have more context threads than actors as some threads will always be idle in this scenario.

No safety-belts will prevent you from running such inefficient layout.

stream.write_all

Synopsis

```
local stream = require "stream"
stream.write_all(io_object, buffer: byte_span|string) -> integer
```

Description

Write all of the supplied data to the stream and blocks current fiber until it completes or errs.

Returns the `buffer`'s size (number of bytes written).

As in Boost.Asio:

> This operation is implemented in terms of zero or more calls to the stream's async_write_some function, and is known as a *composed operation*. The program must ensure that the stream performs no other write operations (such as async_write, the stream's async_write_some function, or any other composed operations that perform writes) until this operation completes.

stream.write_at_least

Synopsis

```
local stream = require "stream"
stream.write_at_least(io_object, buffer: byte_span, minimum: integer) -> integer
```

Description

Write data until a `minimum` number of bytes has been transferred and blocks current fiber until it completes or errs.

Returns the number of bytes written.

As in Boost.Asio:

> This operation is implemented in terms of zero or more calls to the stream's async_write_some function, and is known as a *composed operation*. The program must ensure that the stream performs no other write operations (such as async_write, the stream's async_write_some function, or any other composed operations that perform writes) until this operation completes.

stream.read_all

Synopsis

```
local stream = require "stream"
stream.read_all(io_object, buffer: byte_span) -> integer
```

Description

Read data until the supplied buffer is full and blocks current fiber until it completes or errs.

Returns the `buffer`'s size (number of bytes read).

As in Boost.Asio:

> This operation is implemented in terms of zero or more calls to the stream's async_read_some function, and is known as a *composed operation*. The program must ensure that the stream performs no other read operations (such as async_read, the stream's async_read_some function, or any other composed operations that perform reads) until this operation completes.

stream.read_at_least

Synopsis

```
local stream = require "stream"
stream.read_at_least(io_object, buffer: byte_span, minimum: integer) -> integer
```

Description

Read data until a `minimum` number of bytes has been transferred and blocks current fiber until it completes or errs.

Returns the number of bytes read.

As in Boost.Asio:

> This operation is implemented in terms of zero or more calls to the stream's async_read_some function, and is known as a *composed operation*. The program must ensure that the stream performs no other read operations (such as async_read, the stream's async_read_some function, or any other composed operations that perform reads) until this operation completes.

stream.scanner

```
local stream = require "stream"
local scanner = stream.scanner.new{ stream = system.in_ }
scanner:get_line()
```

This class abstracts formatted buffered textual input as an AWK-inspired scanner. The input stream is broken into records, and each record may be further broken down into fields.

`get_line()` is used to get the next record. Surplus data read from the stream is kept in the buffer to be used in the next call to `get_line()`.

When EOF is found on the stream, the buffered data is returned as the last record. To differentiate records finished on EOF from records finished on `record_separator`, check `self.record_terminator`.

You may change the parsing rules (e.g. record and field separators) once `get_line()` returns.

Line-based protocols

> Many commonly-used internet protocols are line-based, which means that they have protocol elements that are delimited by the character sequence `"\r\n"`. Examples include HTTP, SMTP and FTP.
>
> — https://www.boost.org/doc/libs/1_81_0/doc/html/boost_asio/overview/core/line_based.html

To easily parse these protocols, you may set a `scanner` object with `record_separator="\r\n"` (the default). Then, `get_line()` will return a new line each time it is called. If the field separator/pattern is also specified, the line will be broken into a table made of the fields.

New buffers will be allocated as more space is needed until a specified maximum (or an unspecified maximum default).

Combining strategies

You may also use different parsers & algorithms to consume some parts of the stream. For instance, HTTP starts as a line-delimited textual protocol. Once the header section is consumed, the body payload is determined by rules extracted out of the headers. For `"content-length"` defined message bodies, you read a fixed amount of bytes to consume it.

In such scenario, you may use `scanner` to parse the header section, and, once it's time to read the body, use the method `buffer()` to retrieve already buffered data. Just be sure to call `remove_line()` before calling `buffer()` so the last line of the header section doesn't get mixed up with the body. Then it'll be a matter of calling stream.read_all(3em) (or several calls to `read_some()`) to consume the body.

Once it's time to parse the header section for the next message in the stream, you can call `set_buffer()` to pass the buffered data back to the `scanner`.

Functions

`new(opts: table|nil) → scanner`

Set attributes required by `scanner.mt`, set `opts`'s metatable to `scanner.mt` and returns `opts`. If `opts` is `nil`, then a new table is returned.

You **MUST** set the `stream` attribute (before or after the call to `new()`) before using `scanner`'s methods.

Optional attributes to `opts`:

`record_separator: string|regex = "\r\n"`

The pattern used to split records.

Regexes must be used with care on streaming content. For instance, if you set `record_separator` to the regex `/abc(XYZ)?/`, it is possible that "XYZ" will not match just because it wasn't buffered yet even if it'll appear in the next calls to read() on the stream.

Other tools such as GAWK suffer from the same constraint. Some regexes engines offer special support when working on streaming content, but they don't solve the whole problem as it'd be impossible to differentiate "max record size reached" from "`record_separator` not found" if an attempt were made to use this support.

`field_separator: string|regex|function|nil`

If non-`nil`, defines how to split fields. Otherwise, the whole line/record is returned as is.

Check `regex.split()` to understand how fields are separated. In short, `field_separator` defines what fields *are not*.

On functions, the function is used to split the fields out of the line/record and its return is passed through.

`field_pattern: regex|nil`

Defines what fields **are** (as opposed to `field_separator` that defines what fields **are not**). It must be a regex. Check `regex.patsplit()` for details.

`trim_record: boolean|string = false`

Whether to strip linear whitespace (if string is given, then it'll define the list of whitespace characters) from the beginning and end of each record.

`buffer_size_hint: integer|nil`

The initial size for the buffer. As is the case for every hint, it might be ignored.

`max_record_size: integer = unspecified`

The maximum size for each record/buffer.

`with_awk_defaults(read_stream) → scanner`

Returns a scanner acting on `stream` that has the semantics from AWK defaults:

`record_separator`

`"\n"`

`trim_record`

`true`

`field_separator`

A regex that describes a sequence of linear whitespace.

`get_line(self) → byte_span|byte_span[]`

Reads next record buffering any bytes as required and returns it. If `field_separator`, or `field_pattern` were set, the record's extracted fields are returned.

It also sets `self.record_terminator` to the record separator just read. On end of streams that don't include the record separator, `self.record_terminator` will be set to an empty `byte_span` (or an empty string if record separator was specified as a string).

It also increments `self.record_number` by one on success (it is initially zero).

`buffered_line(self) → byte_span`

Returns current buffered record without extracting its fields. It works like AWK's `$0` variable.

Precondition

A record must be present in the buffer from a previous call to `get_line()`.

`remove_line(self)`

Removes current record from the buffer and sets `self.record_terminator` to `nil`.

Precondition

A record must be present in the buffer from a previous call to `get_line()`.

`buffer(self) → byte_span, integer`

Returns the buffer + the offset where the read data begins.

The returned buffer's capacity may be greater than its length.

`set_buffer(self, buf: byte_span[, offset: integer = 1])`

Set `buf` as the new internal buffer.

`buf`'s capacity will indicate the usable part of the buffer for IO ops and `buf`'s length (after slicing from `offset`) will indicate the buffered data.

Previously buffered record and `self.record_terminator` are discarded.

Example

```
local buffered_data = buf:slice(offset)
scanner:set_buffer(buf, offset)
```

system.arguments

Synopsis

```
local system = require "system"
system.arguments: string[]
```

Description

Arguments passed on the CLI (a.k.a. ARGV). First element in the table is emilua binary path. Second element is the script path. Rest of the elements are anything passed after `"--"` in the command line.

system.environment

Synopsis

```
local system = require "system"
system.environment: { [string]: string }
```

Description

The environment variables.

system.in_

Synopsis

```
local system = require "system"
system.in_
```

Functions

`read_some(self, buffer: byte_span) → integer`

Read data from stdin and blocks current fiber until it completes or errs.

Returns the number of bytes read.

First argument is ignored and it's only there to make it have a stream-like interface.

`dup(self) → file_descriptor`

Creates a new file descriptor that refers to `STDIN_FILENO`.

`isatty(self) → boolean`

See isatty(3).

`tcgetpgrp(self) → integer`

See tcgetpgrp(3).

`tcsetpgrp(self, pgid_id: integer)`

See tcsetpgrp(3).

system.out

Synopsis

```
local system = require "system"
system.out
```

Functions

`write_some(self, buffer: byte_span) → integer`

Write data to stdout and blocks current fiber until it completes or errs.

Returns the number of bytes written.

First argument is ignored and it's only there to make it have a stream-like interface.

`dup(self) → file_descriptor`

Creates a new file descriptor that refers to `STDOUT_FILENO`.

`isatty(self) → boolean`

See isatty(3).

`tcgetpgrp(self) → integer`

See tcgetpgrp(3).

`tcsetpgrp(self, pgid_id: integer)`

See tcsetpgrp(3).

system.err

Synopsis

```
local system = require "system"
system.err
```

Functions

write_some(self, buffer: byte_span) → integer

Write data to stderr and blocks current fiber until it completes or errs.

Returns the number of bytes written.

First argument is ignored and it's only there to make it have a stream-like interface.

dup(self) → file_descriptor

Creates a new file descriptor that refers to `STDERR_FILENO`.

isatty(self) → boolean

See isatty(3).

tcgetpgrp(self) → integer

See tcgetpgrp(3).

tcsetpgrp(self, pgid_id: integer)

See tcsetpgrp(3).

system.exit

Synopsis

```
local system = require "system"
system.exit([code: integer = 0 [, opts: table]])
```

Description

Exit the VM. Other VMs in the process are not stopped.

Parameters

code

If caller is the main VM, `code` is used as the application exit code.

opts

If caller is the main VM, then `opts` is a table that accepts the following options:

`force: 0|1|2|"abort" = 0`

`0`
: Nothing.

`1`
: Not implemented yet.

`2`
: Exit the process forcefully (little to none cleanup steps are performed).

`"abort"`
: Exit the process even more forcefully (equivalent to the C function `abort()`).

system.signal

Synopsis

```
local system = require "system"
system.signal: table
```

Constants

- SIGABRT.
- SIGFPE.
- SIGILL.
- SIGINT.
- SIGSEGV.
- SIGTERM.

UNIX constants

Availability depending on the host system.

- SIGALRM.
- SIGBUS.
- SIGCHLD.
- SIGCONT.
- SIGHUP.
- SIGIO.
- SIGKILL.
- SIGPIPE.
- SIGPROF.
- SIGQUIT.
- SIGSTOP.
- SIGSYS.
- SIGTRAP.
- SIGTSTP.
- SIGTTIN.
- SIGTTOU.

- SIGURG.
- SIGUSR1.
- SIGUSR2.
- SIGVTALRM.
- SIGWINCH.
- SIGXCPU.
- SIGXFSZ.

Windows constants

Availability depending on the host system.

- SIGBREAK.

Signal handling also works on Windows, as the Microsoft Visual C++ runtime library maps console events like Ctrl+C to the equivalent signal.

system.signal.raise

Synopsis

```
local system = require "system"
system.signal.raise(signal: integer)
```

Description

Sends a signal to the calling process.

system.signal.set

```
local set = system.signal.set.new(system.signal.SIGTERM, system.signal.SIGINT)
set:wait()
```

This class provides the ability to wait for one or more signals to occur.

Multiple registration of signals

As in Boost.Asio (translated to fibers/emilua lingo):

> The same signal number may be registered with different [`set`] objects. When the signal occurs, one [signal notification is queued] for each [`set`] object.

Functions

`new([sig1: integer, ⋯]) → system.signal.set`

Constructor.

Arguments are treated as signals to be added to the set.

Only the main VM on the process may create new `set` objects. If the VM elects another VM to be the new main VM, its old `set` objects will remain valid and working, but the VM won't be able to create new `set` objects.

`add(self, signal: integer)`

Add a signal to the set.

Only the master VM is allowed to use this function.

`remove(self, signal: integer)`

Remove a signal from the set.

`clear(self)`

Remove all signals from the set.

`cancel(self)`

Cancel all operations associated with the set.

`wait(self) → integer`

Wait for a signal to be delivered. The function will return when:

- One of the registered signals in the set occurs; or
- The set was cancelled, in which case the function will raise the exception `boost::asio::error::operation_aborted`.

A number is returned to indicate which signal occurred.

Queueing of signal notifications

As in Boost.Asio (translated to fibers/emilua lingo):

> If a signal is registered with a [`set`], and the signal occurs when there are no [calls to `wait()`], then the signal notification is queued. The next [call to `wait()` on that `set`] will dequeue the notification. If multiple notifications are queued, subsequent [`wait()` calls] dequeue them one at a time. Signal notifications are dequeued in order of ascending signal number.
>
> If a signal number is removed from a [`set`] (using the [`remove()` member function]) then any queued notifications for that signal are discarded.

system.signal.ignore

Synopsis

```
local system = require "system"
system.signal.ignore(signal: integer)
```

Description

Ignore signal.

This function will fail if you try to ignore a signal for which a `system.signal.set` object exists.

Only the master VM is allowed to use this function.

This function is only available to POSIX systems.

system.signal.default

Synopsis

```
local system = require "system"
system.signal.default(signal: integer)
```

Description

Reset `signal`'s handling to the system's default.

There's no need to set the default handlers at the start of the program. The Emilua runtime will already do that for you.

This function will fail if you try to reset a signal for which a `system.signal.set` object exists.

Only the master VM is allowed to use this function.

This function is only available to POSIX systems.

system.spawn

Synopsis

```
local system = require "system"
system.spawn(opts: table) -> subprocess
```

Description

Creates a new process.

Named parameters

`program: string|filesystem.path|file_descriptor`

`string`

A simple filename. The system searches for this file in the list of directories specified by PATH (in the same way as for execvp(3)).

`filesystem.path`

The path (which can be absolute or relative) of the executable.

`file_descriptor`

A file descriptor to the executable. See fexecve(3).

`arguments: string[]|nil`

A table of strings that will be used as the created process' `argv`. On `nil`, an empty `argv` will be used.

Don't forget to include the name of the program as the first argument.

`environment: { [string]: string }|nil`

A table of strings that will be used as the created process' `envp`. On `nil`, an empty `envp` will be used.

`stdin,stdout,stderr: "share"|file_descriptor|nil`

`"share"`

The spawned process will share the specified standard handle (`stdin`, `stdout`, or `stderr`) with the caller process.

`file_descriptor`

Use the file descriptor as the specified standard handle (`stdin`, `stdout`, or `stderr`) for the spawned process.

`nil`

Create and use a closed pipe end as the specified standard handle (`stdin`, `stdout`, or `stderr`) for the spawned process.

On Windows, it's unspecified (will vary depending on whether any redirection is done at all, `dwCreationFlags`'s value, etc).

`extra_fds: { [integer]: file_descriptor }|nil`

Extra file descriptors for the child to inherit. Parent and child processes don't need to share the same numeric value reference for a given file description. The file descriptor number used in the child process will be the one specified in the key portion of the dictionary argument. Only file descriptors numbered from `3` to `9` are acceptable (i.e. the same limitations of low fds that you're likely to face on older UNIX shells). If you need to pass more than 10 file descriptors — stdin, stdout, stderr, plus these extra 7 file descriptors — use another interface (e.g. `SCM_RIGHTS`).

Not available on Windows.

`signal_on_gcreaper: integer = system.signal.SIGTERM`

Each process is responsible for reaping its own children. A process that fails to reap its children will soon exhaust its OS-provided resources. For short-lived programs that's hardly a problem given the process quits and its children are re-parented to the next subreaper in the chain (usually the init process). However for a concurrency runtime such as Emilua we expect other concurrent tasks to remain unaffected by the one failing task (be it a single fiber or the whole VM). Emilua will then transparently reap any child process for which its handle has been GC'ed. `signal_on_gcreaper` allows the user to specify a signal to be sent to the child that's about to be reaped at this occasion.

By default, the signal `system.signal.SIGTERM` will be sent to the child and then the main Emilua process will — indefinitely, non-blockingly, and non-pollingly — await for all of its children to finish even if there's no longer any Lua program being executed. Use the more dangerous `system.signal.SIGKILL` if you don't want the main Emilua process to wait long for the child. Use `0` if you don't want the Emilua reaper to send any signal before awaiting for the child.

Ideally the system kernel would expose some re-parent syscall, but until then (if ever), `signal_on_gcreaper` will be necessary.

Only available on Linux.

`pd_daemon: boolean = see-below`

Instead of the default terminate-on-close behaviour, allow the process to live until it is explicitly killed with kill(2).

By default, it's `true` unless the parent process is in capability mode (see cap_enter(2)).

Only available on FreeBSD.

`scheduler.policy: string|nil`

Values acceptable on Linux for non-real-time policies are:

`"other"`

See `SCHED_OTHER`.

`"batch"`

See `SCHED_BATCH`.

`"idle"`

See `SCHED_IDLE`.

Values acceptable on Linux for real-time policies are:

`"fifo"`

See `SCHED_FIFO`. Must also set `scheduler.priority`.

`"rr"`

See `SCHED_RR`. Must also set `scheduler.priority`.

Not available on Windows.

`scheduler.priority: integer|nil`

The interpretation of this parameter is dependant on `scheduler.policy`.

Not available on Windows.

`scheduler.reset_on_fork: boolean = false`

If `true`, grandchildren created as a result of a call to fork(2) from the direct child will not inherit privileged scheduling policies. If set, must also set `scheduler.policy`.

Not available on Windows.

`start_new_session: boolean = false`

Whether to create a new session and become the session leader. On `true`, calls `setsid()` on the child.

On Windows, `DETACHED_PROCESS|CREATE_NEW_PROCESS_GROUP` is used in creation flags.

`set_ctty: file_descriptor|nil`

Set the controlling terminal for the child. It is an error to specify `set_ctty`, but omit `start_new_session`.

It's an error to specify both `set_ctty` and `foreground`.

Not available on Windows.

`process_group: integer|nil`

Set the process group (it calls `setpgid()` on the child). On 0, the child's process group ID is made the same as its process ID.

On Windows, only `0` is supported (`CREATE_NEW_PROCESS_GROUP` is used in creation flags).

`foreground: "stdin"|"stdout"|"stderr"|file_descriptor|nil`

Make the child be the foreground job for the specified controlling terminal by calling `tcsetpgrp()` (`SIGTTOU` will be blocked for the duration of the call). It is an error to specify `foreground`, but omit `process_group`.

`"stdin"`, `"stdout"`, and `"stderr"` can only be specified if parent and child share the same file for the specified standard handle.

It's an error to specify both `foreground` and `set_ctty`.

Not available on Windows.

`ruid: integer|nil`

Set the real user ID.

Not available on Windows.

`euid: integer|nil`

Set the effective user ID. If the set-user-ID permission bit is enabled on the executable file, its effect will override this setting (see execve(2)).

Not available on Windows.

`rgid: integer|nil`

Set the real group ID.

Not available on Windows.

`egid: integer|nil`

Set the effective group ID. If the set-group-ID permission bit is enabled on the executable file, its effect will override this setting (see execve(2)).

Not available on Windows.

`extra_groups: integer[]|nil`

Set the supplementary group IDs.

Not available on Windows.

`umask: integer|nil`

See umask(3p).

Not available on Windows.

`working_directory: filesystem.path|file_descriptor|nil`

Sets the working directory for the spawned program.

`pdeathsig: integer|nil`

Signal that the process will get when its parent dies. If the executable file contains set-user-ID, set-group-ID, or contains associated capabilities, `pdeathsig` will be cleared.

"Parent" is a difficult term to define here. For Linux, that's not the process, but the thread. For Emilua, the thread will exist for at least as long as the calling Lua VM exists (even if the Lua VM might jump between threads). The thread will also exist for even longer, for as long as other Lua VMs are using it.

Not available on Windows.

`nsenter_user: file_descriptor|nil`

Enter in this Linux user namespace. When `nsenter_user` is specified, Emilua always enter in the user namespace before any other namespace.

Only available on Linux.

`nsenter_mount: file_descriptor|nil`

Enter in this Linux mount namespace.

Only available on Linux.

`nsenter_uts: file_descriptor|nil`

Enter in this Linux UTS namespace.

Only available on Linux.

`nsenter_ipc: file_descriptor|nil`

Enter in this Linux IPC namespace.

Only available on Linux.

`nsenter_net: file_descriptor|nil`

Enter in this Linux net namespace.

Only available on Linux.

`show_window: "hide"|"shownormal"|"normal"|"showminimized"|"showmaximized"|"maximize"|"shownoactivate"|"show"|"minimize"|"showminnoactive"|"showna"|"restore"|"forceminimize"|nil`

If present, `STARTUPINFO.dwFlags` will include `STARTF_USESHOWWINDOW`, and `STARTUPINFO.wShowWindow` will be initialized with the indicated value.

Only available on Windows.

`create_breakaway_from_job: boolean = false`

Only available on Windows.

`create_new_console: boolean = false`

Only available on Windows.

`create_no_window: boolean = false`

Only available on Windows.

`detached_process: boolean = false`

Only available on Windows.

`subprocess` functions

`wait(self)`

Wait for the process to finish, and then reap it. Information regarding termination status is stored in `exit_code` and `exit_signal`.

If your code fails to call `wait()`, the Emilua runtime will reap the process in your stead as soon as the GC collects `self` and the underlying subprocess finishes. It's important to reap children processes to free OS-associated resources.

`kill(self, signal: integer)`

Send a signal to the process.

You may specify `0` (the null signal) to not send any signal, but still let the OS to perform permission checks (reported as raised errors).

`cap_get(self) → system.linux_capabilities`

See cap_get_pid(3).

`subprocess` properties

`exit_code: integer`

The process return code as passed to exit(3). If the process was terminated by a signal, this will be `128 + exit_signal` (as done in BASH).

You can only access this field for `wait()`'ed processes.

`exit_signal: integer|nil`

The signal that terminated the process. If the process was **not** terminated by a signal, this will be `nil`.

You can only access this field for `wait()`'ed processes.

`pid: integer`

The process id used by the OS to represent this child process (e.g. the number that shows up in `/proc` on some UNIX systems).

You can only access this field for non-`wait()`'ed processes.

Bugs

Windows properly supports line-breaks in `arguments`. However if you're running a `.bat` or a `.cmd` file, there's a bug in `CMD.exe` that stops parsing the command line at the line-break. This is a bug in Windows. To fix this bug, you need to install TCC-RT from JP Software (or another `CMD.exe` replacement such as wineconsole) and set `COMSPEC` to this new interpreter. Microsoft won't fix this bug.

system.getresuid

Synopsis

```
local system = require "system"
system.getresuid() -> integer, integer, integer
```

Description

Returns the real UID, the effective UID, and the saved set-user-ID of the calling process, respectively.

system.getresgid

Synopsis

```
local system = require "system"
system.getresgid() -> integer, integer, integer
```

Description

Returns the real GID, the effective GID, and the saved set-group-ID of the calling process, respectively.

system.setresuid

Synopsis

```
local system = require "system"
system.setresuid(ruid: integer, euid: integer, suid: integer)
```

Description

Sets the real UID, the effective UID, and the saved set-user-ID of the calling process.

If one of the arguments equals `-1`, the corresponding value is not changed.

Only the master VM is allowed to use this function.

system.setresgid

Synopsis

```
local system = require "system"
system.setresgid(rgid: integer, egid: integer, sgid: integer)
```

Description

Sets the real GID, the effective GID, and the saved set-group-ID of the calling process.

If one of the arguments equals `-1`, the corresponding value is not changed.

Only the master VM is allowed to use this function.

system.getgroups

Synopsis

```
local system = require "system"
system.getgroups() -> integer[]
```

Description

Returns the current supplementary group IDs of the calling process. It is unspecified whether `getgroups()` also returns the effective group ID in the list.

system.setgroups

Synopsis

```
local system = require "system"
system.setgroups(groups: integer[])
```

Description

Sets the supplementary group IDs for the calling process.

Only the master VM is allowed to use this function.

system.linux_capabilities

```
local system = require "system"
local caps = system.cap_init()
caps:set_proc()
system.cap_reset_ambient()
```

Functions

cap_get_proc() → linux_capabilities

See cap_get_proc(3).

cap_init() → linux_capabilities

See cap_init(3).

cap_from_text(caps: string) → linux_capabilities

See cap_from_text(3).

cap_get_bound(cap: string) → boolean

See cap_get_bound(3).

cap_drop_bound(cap: string)

See cap_drop_bound(3).

Only the master VM is allowed to use this function.

cap_get_ambient(cap: string) → boolean

See cap_get_ambient(3).

cap_set_ambient(cap: string, value: boolean)

See cap_set_ambient(3).

Only the master VM is allowed to use this function.

cap_reset_ambient()

See cap_reset_ambient(3).

Only the master VM is allowed to use this function.

`cap_get_secbits() → integer`

See cap_get_secbits(3).

`cap_set_secbits(bits: integer)`

See cap_set_secbits(3).

The securebits flag constants are available from the `system` table:

- `SECBIT_NOROOT`
- `SECBIT_NOROOT_LOCKED`
- `SECBIT_NO_SETUID_FIXUP`
- `SECBIT_NO_SETUID_FIXUP_LOCKED`
- `SECBIT_KEEP_CAPS`
- `SECBIT_KEEP_CAPS_LOCKED`
- `SECBIT_NO_CAP_AMBIENT_RAISE`
- `SECBIT_NO_CAP_AMBIENT_RAISE_LOCKED`

Only the master VM is allowed to use this function.

`dup(self) → linux_capabilities`

See cap_dup(3).

`clear(self)`

See cap_clear(3).

`clear_flag(self, flag: string)`

See cap_clear_flag(3).

`get_flag(self, cap: string, flag: string) → boolean`

See cap_get_flag(3).

`set_flag(self, flag: string, caps: string[], value: boolean)`

See cap_set_flag(3).

`fill_flag(self, to: string, ref: linux_capabilities, from: string)`

See cap_fill_flag(3).

`fill(self, to: string, from: string)`

See cap_fill(3).

`set_proc(self)`

See cap_set_proc(3).

Only the master VM is allowed to use this function.

`get_nsowner(self) → integer`

See cap_get_nsowner(3).

`set_nsowner(self, rootuid: integer)`

See cap_set_nsowner(3).

Metamethods

`__tostring()`

See cap_to_text(3).

system.getpid

Synopsis

```
local system = require "system"
system.getpid() -> integer
```

Description

Returns the process ID of the calling process.

system.getppid

Synopsis

```
local system = require "system"
system.getppid() -> integer
```

Description

Returns the parent process ID of the calling process.

system.kill

Synopsis

```
local system = require "system"
system.kill(pid: integer, sig: integer)
```

Description

See kill(2).

Only the master VM is allowed to use this function.

system.getpgrp

Synopsis

```
local system = require "system"
system.getpgrp() -> integer
```

Description

See getpgrp(3).

system.getpgid

Synopsis

```
local system = require "system"
system.getpgid(pid: integer) -> integer
```

Description

See getpgid(3).

system.setpgid

Synopsis

```
local system = require "system"
system.setpgid(pid: integer, pgid: integer)
```

Description

See setpgid(3).

Only the master VM is allowed to use this function.

system.getsid

Synopsis

```
local system = require "system"
system.getsid(pid: integer) -> integer
```

Description

See getsid(3).

system.setsid

Synopsis

```
local system = require "system"
system.setsid() -> integer
```

Description

See setsid(3).

Only the master VM is allowed to use this function.

system.jail_set

Synopsis

```
local system = require "system"
system.jail_set(params: { [string]: string|boolean }, flags: string[]|nil) -> integer
```

Description

Create or modify a jail.

Jail parameters are given as strings and they'll be transparently converted to the native format accepted by the kernel.

`flags` may contain the following values:

- `"create"`
- `"update"`
- `"dying"`

See jail(8) for more information on the core jail parameters.

system.jail_get

Synopsis

```
local system = require "system"
system.jail_get(params: table, flags: string[]|nil) -> integer, { [string]: string }
```

Description

Retrieves jail parameters.

`params` specify — as a list of strings — which parameters are desired in the returned value.

`params` also specify — in the same format as used by system.jail_set() — which jail to read values from. Usually `"jid"` or `"name"` are used as filters. The special parameter `"lastjid"` can be used to retrieve a list of all jails.

`flags` may contain the following values:

- `"dying"`

Example

Retrieve the hostname and path of jail "foo":

```
local jid, params = system.jail_get {
    "host.hostname",
    "path",
    ["name"] = "foo"
}

print(jid)
print(params["host.hostname"])
print(params.path)
```

system.jail_remove

Synopsis

```
local system = require "system"
system.jail_remove(jid: integer)
```

Description

Removes the jail identified by `jid`.

system.jailparam_all

Synopsis

```
local system = require "system"
system.jailparam_all() -> string[]
```

Description

Returns a list of all known jail parameters.

tls.context

Functions

`new(method: string) → tls.context`

Constructor.

`method` must be one of:

- `"sslv2"`
- `"sslv2_client"`
- `"sslv2_server"`
- `"sslv3"`
- `"sslv3_client"`
- `"sslv3_server"`
- `"tlsv1"`
- `"tlsv1_client"`
- `"tlsv1_server"`
- `"sslv23"`
- `"sslv23_client"`
- `"sslv23_server"`
- `"tlsv11"`
- `"tlsv11_client"`
- `"tlsv11_server"`
- `"tlsv12"`
- `"tlsv12_client"`
- `"tlsv12_server"`
- `"tlsv13"`
- `"tlsv13_client"`
- `"tlsv13_server"`
- `"tls"`
- `"tls_client"`
- `"tls_server"`

`add_certificate_authority(self, data: byte_span)`

Add certification authority for performing verification.

add_verify_path(self, path: filesystem.path)

Add a directory containing certificate authority files to be used for performing verification.

clear_options(self, flags: integer)

Clear options on the context.

load_verify_file(self, filename: filesystem.path)

Load a certification authority file for performing verification.

set_default_verify_paths(self)

Configures the context to use the default directories for finding certification authority certificates.

set_options(self, flags: integer)

Set options on the context.

set_password_callback(self, callback: function)

Set the password callback.

`callback`'s signature must be:

```
function callback(max_length: integer, purpose: string) -> string
```

`purpose` will be either `"for_reading"` or `"for_writing"`.

The function `callback` will be called from an unspecified fiber where IO/blocking operations are disabled.

set_verify_callback(self, callback: string[, callback_options…])

Set the callback used to verify peer certificates.

For now only one callback is supported:

`"host_name_verification"`
: `callback_options` will be a single string containing the host name.

set_verify_depth(self, depth: integer)

Set the peer verification depth.

set_verify_mode(self, mode: string)

Set the peer verification mode.

`mode` might be one of the following:

- `"none"`.
- `"peer"`.
- `"fail_if_no_peer_cert"`.
- `"client_once"`.

use_certificate(self, data: byte_span, fmt: string)

Use a certificate from a memory buffer.

`fmt` might be one of the following:

`"asn1"`
: ASN.1 file.

`"pem"`
: PEM file.

use_certificate_chain(self, data: byte_span)

Use a certificate chain from a memory buffer.

use_certificate_chain_file(self, filename: filesystem.path)

Use a certificate chain from a file.

use_certificate_file(self, filename: filesystem.path, fmt: string)

Use a certificate from a file.

`fmt` might be one of the following:

`"asn1"`
: ASN.1 file.

`"pem"`
: PEM file.

use_private_key(self, data: byte_span, fmt: string)

Use a private key from a memory buffer.

`fmt` might be one of the following:

`"asn1"`
: ASN.1 file.

`"pem"`
: PEM file.

use_private_key_file(self, filename: filesystem.path, fmt: string)

Use a private key from a file.

`fmt` might be one of the following:

`"asn1"`
: ASN.1 file.

`"pem"`
: PEM file.

use_rsa_private_key(self, data: byte_span, fmt: string)

Use an RSA private key from a memory buffer.

`fmt` might be one of the following:

`"asn1"`
: ASN.1 file.

`"pem"`
: PEM file.

use_rsa_private_key_file(self, filename: filesystem.path, fmt: string)

Use an RSA private key from a file.

`fmt` might be one of the following:

`"asn1"`
: ASN.1 file.

`"pem"`
: PEM file.

use_tmp_dh(self, data: byte_span)

Use the specified memory buffer to obtain the temporary Diffie-Hellman parameters.

use_tmp_dh_file(self, filename: filesystem.path)

Use the specified file to obtain the temporary Diffie-Hellman parameters.

tls.context_flag

This module contains SSL options flag constants.

default_workarounds

The flag with same name in Boost.Asio:

> Implement various bug workarounds.

no_compression

The flag with same name in Boost.Asio:

> Disable compression. Compression is disabled by default.

no_sslv2

The flag with same name in Boost.Asio:

> Disable SSL v2.

no_sslv3

The flag with same name in Boost.Asio:

> Disable SSL v3.

no_tlsv1

The flag with same name in Boost.Asio:

> Disable TLS v1.

no_tlsv1_1

The flag with same name in Boost.Asio:

> Disable TLS v1.1.

no_tlsv1_2

The flag with same name in Boost.Asio:

> Disable TLS v1.2.

no_tlsv1_3

The flag with same name in Boost.Asio:

> Disable TLS v1.3.

single_dh_use

The flag with same name in Boost.Asio:

> Always create a new key when using tmp_dh parameters.

tls.socket

```
tls_ctx = tls.context.new('tlsv13')

local s = ip.tcp.socket.new()
ip.connect(s, ip.tcp.get_address_info('www.example.com', 'https'))
s = tls.socket.new(s, tls_ctx)
s:client_handshake()
s = http.socket.new(s)

local req = http.request.new()
local res = http.response.new()
req.headers.host = 'www.example.com'

s:write_request(req)
s:read_response(res)
```

Functions

new(sock: ip.tcp.socket, tls_ctx: tls.context) → tls.socket

Constructor.

client_handshake(self)

Perform the TLS client handshake and suspend current fiber until operation finishes.

server_handshake(self)

Perform the TLS server handshake and suspend current fiber until operation finishes.

read_some(self, buffer: byte_span) → integer

Read data from the stream socket and blocks current fiber until it completes or errs.

Returns the number of bytes read.

write_some(self, buffer: byte_span) → integer

Write data to the stream socket and blocks current fiber until it completes or errs.

Returns the number of bytes written.

set_server_name(self, hostname: string)

Sets the server name indication.

set_verify_callback(self, callback: string[, callback_options…])

Set the callback used to verify peer certificates.

For now only one callback is supported:

`"host_name_verification"`
: `callback_options` will be a single string containing the host name.

set_verify_depth(self, depth: integer)

Set the peer verification depth.

set_verify_mode(self, mode: string)

Set the peer verification mode.

`mode` might be one of the following:

- `"none"`.
- `"peer"`.
- `"fail_if_no_peer_cert"`.
- `"client_once"`.

unix.datagram_socket

```
local sock = unix.datagram_socket.new()
sock.open()
sock.bind(filesystem.path.new('/tmp/9Lq7BNBnBycd6nxy.socket'))

local buf = byte_span.new(1024)
local nread = sock:receive(buf)
print(buf:slice(1, nread))
```

Functions

new() → unix.datagram_socket

```
new()                    ①
new(fd: file_descriptor) ②
```

① Default constructor.

② Converts a file descriptor into an `unix.datagram_socket` object.

pair() → unix.datagram_socket, unix.datagram_socket

Create a pair of connected sockets.

open(self)

Open the socket.

bind(self, pathname: filesystem.path)

Bind the socket to the given local endpoint.

connect(self, pathname: filesystem.path)

Set the default destination address so datagrams can be sent using `send()` without specifying a destination address.

disconnect(self)

Dissolve the socket's association by resetting the socket's peer address (i.e. connect(3) will be called with an `AF_UNSPEC` address).

close(self)

Close the socket.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `boost::asio::error::operation_aborted` error.

`shutdown(self, what: string)`

Disable sends or receives on the socket.

`what` can be one of the following:

`"receive"`
: Shutdown the receive side of the socket.

`"send"`
: Shutdown the send side of the socket.

`"both"`
: Shutdown both send and receive on the socket.

`cancel(self)`

Cancel all asynchronous operations associated with the acceptor.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, fd: file_descriptor)`

Assign an existing native socket to `self`.

`release(self) -> file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native socket is then transferred to the caller.

`receive(self, buffer: byte_span[, flags: integer]) -> integer`

Receive a datagram and blocks current fiber until it completes or errs.

Returns the number of bytes read.

`flags` is `0` or an or-combination of values from unix.message_flag(3em).

`receive_from(self, buffer: byte_span[, flags: integer]) → integer, filesystem.path`

Receive a datagram and blocks current fiber until it completes or errs.

Returns the number of bytes read plus the pathname of the remote sender of the datagram.

`flags` is `0` or an or-combination of values from unix.message_flag(3em).

`send(self, buffer: byte_span[, flags: integer]) → integer`

Send data on the datagram socket and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`flags` is `0` or an or-combination of values from unix.message_flag(3em).

The `send` operation can only be used with a connected socket. Use the `send_to` function to send data on an unconnected datagram socket.

`send_to(self, buffer: byte_span, pathname: filesystem.path[, flags: integer]) → integer`

Send a datagram to the specified remote endpoint and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`flags` is `0` or an or-combination of values from unix.message_flag(3em).

`receive_with_fds(self, buffer: byte_span, maxfds: integer) → integer, file_descriptor[]`

Receive a datagram and blocks current fiber until it completes or errs.

Returns the number of bytes read plus the table containing the `fds` read.

`receive_from_with_fds(self, buffer: byte_span, maxfds: integer) → integer, filesystem.path, file_descriptor[]`

Receive a datagram and blocks current fiber until it completes or errs.

Returns the number of bytes read plus the pathname of the remote sender of the datagram plus the table containing the `fds` read.

`send_with_fds(self, buffer: byte_span, fds: file_descriptor[]) → integer`

Send data on the datagram socket and blocks current fiber until it completes or errs.

Returns the number of bytes written.

The `send` operation can only be used with a connected socket. Use the `send_to` function to send data on an unconnected datagram socket.

`send_to_with_fds(self, buffer: byte_span, pathname: filesystem.path, fds: file_descriptor[]) → integer`

Send a datagram to the specified remote endpoint and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`set_option(self, opt: string, val)`

Set an option on the socket.

Currently available options are:

`"debug"`
: Check Boost.Asio documentation.

`"send_buffer_size"`
: Check Boost.Asio documentation.

`"receive_buffer_size"`
: Check Boost.Asio documentation.

`get_option(self, opt: string) → value`

Get an option from the socket.

Currently available options are:

`"debug"`
: Check Boost.Asio documentation.

`"send_buffer_size"`
: Check Boost.Asio documentation.

`"receive_buffer_size"`
: Check Boost.Asio documentation.

`io_control(self, command: string[, ⋯])`

Perform an IO control command on the socket.

Currently available commands are:

`"bytes_readable"`

Expects no arguments. Get the amount of data that can be read without blocking. Implements the `FIONREAD` IO control command.

Properties

`is_open: boolean`

Whether the socket is open.

`local_path: filesystem.path`

The local address endpoint of the socket.

`remote_path: filesystem.path`

The remote address endpoint of the socket.

unix.message_flag

This module contains flag constants useful to send/receive operations.

peek

The flag with same name in Boost.Asio:

> Peek at incoming data without removing it from the input queue.

unix.stream_acceptor

```
local a = unix.stream_acceptor.new()
a:open()
a:bind(filesystem.path.new('/tmp/9Lq7BNBnBycd6nxy.socket'))
a:listen()

while true do
    local s = a:accept()
    spawn(function()
        my_client_handler(s)
    end)
end
```

Functions

new() → unix.stream_acceptor

```
new()                    ①
new(fd: file_descriptor) ②
```

① Default constructor.

② Converts a file descriptor into an `unix.stream_acceptor` object.

open(self)

Open the acceptor.

set_option(self, opt: string, val)

Set an option on the acceptor.

Currently available options are:

`"enable_connection_aborted"`
: Check Boost.Asio documentation.

`"debug"`
: Check Boost.Asio documentation.

get_option(self, opt: string) → value

Get an option from the acceptor.

Currently available options are:

`"enable_connection_aborted"`

Check Boost.Asio documentation.

`"debug"`

Check Boost.Asio documentation.

`bind(self, pathname: filesystem.path)`

Bind the acceptor to the given local endpoint.

`listen(self [, backlog: integer])`

Place the acceptor into the state where it will listen for new connections.

`backlog` is the maximum length of the queue of pending connections. If not provided, an implementation defined maximum length will be used.

`accept(self) -> unix.stream_socket`

Initiate an accept operation and blocks current fiber until it completes or errs.

`close(self)`

Close the acceptor.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous accept operations will be cancelled immediately.
>
> A subsequent call to open() is required before the acceptor can again be used to again perform socket accept operations.

`cancel(self)`

Cancel all asynchronous operations associated with the acceptor.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, fd: file_descriptor)`

Assign an existing native acceptor to `self`.

`release(self) -> file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous accept operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native acceptor is then transferred to the caller.

Properties

`is_open: boolean`

Whether the acceptor is open.

`local_path: filesystem.path`

The local address of the acceptor.

unix.stream_socket

```
local a, b = unix.stream_socket.pair()

spawn(function()
    local buf = byte_span.new(1024)
    local nread = b:read_some(buf)
    print(buf:slice(1, nread))
end):detach()

local nwritten = stream.write_all(a, 'Hello World')
print(nwritten)
```

Functions

new() → unix.stream_socket

```
new()                    ①
new(fd: file_descriptor) ②
```

① Default constructor.

② Converts a file descriptor into an `unix.stream_socket` object.

pair() → unix.stream_socket, unix.stream_socket

Create a pair of connected sockets.

open(self)

Open the socket.

bind(self, pathname: filesystem.path)

Bind the socket to the given local endpoint.

close(self)

Close the socket.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `boost::asio::error::operation_aborted` error.

`cancel(self)`

Cancel all asynchronous operations associated with the acceptor.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, fd: file_descriptor)`

Assign an existing native socket to `self`.

`release(self) -> file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native socket is then transferred to the caller.

`io_control(self, command: string[, ...])`

Perform an IO control command on the socket.

Currently available commands are:

`"bytes_readable"`
: Expects no arguments. Get the amount of data that can be read without blocking. Implements the `FIONREAD` IO control command.

`shutdown(self, what: string)`

Disable sends or receives on the socket.

`what` can be one of the following:

`"receive"`
: Shutdown the receive side of the socket.

`"send"`
: Shutdown the send side of the socket.

`"both"`
: Shutdown both send and receive on the socket.

`connect(self, pathname: filesystem.path)`

Initiate a connect operation and blocks current fiber until it completes or errs.

`disconnect(self)`

Dissolve the socket's association by resetting the socket's peer address (i.e. connect(3) will be called with an `AF_UNSPEC` address).

`read_some(self, buffer: byte_span) → integer`

Read data from the stream socket and blocks current fiber until it completes or errs.

Returns the number of bytes read.

`write_some(self, buffer: byte_span) → integer`

Write data to the stream socket and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`receive_with_fds(self, buffer: byte_span, maxfds: integer) → integer, file_descriptor[]`

Read data from the stream socket and blocks current fiber until it completes or errs.

Returns the number of bytes read + the table containing the `fds` read.

`send_with_fds(self, buffer: byte_span, fds: file_descriptor[]) → integer`

Write data to the stream socket and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`fds` are not closed and can be re-converted to some Emilua IO object if so one wishes.

`set_option(self, opt: string, val)`

Set an option on the socket.

Currently available options are:

`"send_low_watermark"`
: Check Boost.Asio documentation.

`"send_buffer_size"`
: Check Boost.Asio documentation.

`"receive_low_watermark"`
: Check Boost.Asio documentation.

`"receive_buffer_size"`
: Check Boost.Asio documentation.

`"debug"`
: Check Boost.Asio documentation.

`get_option(self, opt: string) → value`

Get an option from the socket.

Currently available options are:

`"send_low_watermark"`
: Check Boost.Asio documentation.

`"send_buffer_size"`
: Check Boost.Asio documentation.

`"receive_low_watermark"`
: Check Boost.Asio documentation.

`"receive_buffer_size"`
: Check Boost.Asio documentation.

`"debug"`
: Check Boost.Asio documentation.

`"remote_credentials": { uid: integer, groups: integer[], pid: integer }`
: Returns the credentials from the remote process.

On Linux, `groups` don't include the supplementary group list.

`pid` is racy and you shouldn't use it for anything but debugging purposes.

`"remote_security_labels": { [string]: string }|string|nil`
: (FreeBSD only) Returns the security labels associated with each policy for the remote process.

  Optionally one may pass an extra argument to `get_option()` with either a list of strings for the policies of interest, or just a single string in case there's only one policy of interest.

`"remote_security_label": string`
: (Linux only) Returns the SELinux security label associated with the remote process.

Properties

`is_open: boolean`

Whether the socket is open.

`local_path: filesystem.path`

The local address of the socket.

`remote_path: filesystem.path`

The remote address of the socket.

unix.seqpacket_acceptor

```
local a = unix.seqpacket_acceptor.new()
a:open()
a:bind(filesystem.path.new('/tmp/9Lq7BNBnBycd6nxy.socket'))
a:listen()

while true do
    local s = a:accept()
    spawn(function()
        my_client_handler(s)
    end)
end
```

Functions

new() → unix.seqpacket_acceptor

```
new()                    ①
new(fd: file_descriptor) ②
```

① Default constructor.

② Converts a file descriptor into an `unix.seqpacket_acceptor` object.

open(self)

Open the acceptor.

set_option(self, opt: string, val)

Set an option on the acceptor.

Currently available options are:

`"enable_connection_aborted"`

Check Boost.Asio documentation.

`"debug"`

Check Boost.Asio documentation.

get_option(self, opt: string) → value

Get an option from the acceptor.

Currently available options are:

`"enable_connection_aborted"`

Check Boost.Asio documentation.

`"debug"`

Check Boost.Asio documentation.

`bind(self, pathname: filesystem.path)`

Bind the acceptor to the given local endpoint.

`listen(self [, backlog: integer])`

Place the acceptor into the state where it will listen for new connections.

`backlog` is the maximum length of the queue of pending connections. If not provided, an implementation defined maximum length will be used.

`accept(self) -> unix.seqpacket_socket`

Initiate an accept operation and blocks current fiber until it completes or errs.

`close(self)`

Close the acceptor.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous accept operations will be cancelled immediately.
>
> A subsequent call to open() is required before the acceptor can again be used to again perform socket accept operations.

`cancel(self)`

Cancel all asynchronous operations associated with the acceptor.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, fd: file_descriptor)`

Assign an existing native acceptor to `self`.

`release(self) -> file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous accept operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native acceptor is then transferred to the caller.

Properties

`is_open: boolean`

Whether the acceptor is open.

`local_path: filesystem.path`

The local address of the acceptor.

unix.seqpacket_socket

```
local sock = unix.seqpacket_socket.new()
sock.open()
sock.bind(filesystem.path.new('/tmp/9Lq7BNBnBycd6nxy.socket'))

local buf = byte_span.new(1024)
local nread = sock:receive(buf)
print(buf:slice(1, nread))
```

A note on 0-sized packets

`AF_UNIX+SOCK_SEQPACKET` sockets behave just the same on Linux and BSD systems. It's safe to use them as IPC primitives in your system. However there are a few caveats related to the idea of what `SOCK_SEQPACKET` were supposed to mean originally.

> seems SEQPACKET is too exotic thing that everyone implements it in own manner, because i've tested SCTP seqpacket implementation, and found [...]
>
> — Arseny Krasnov, https://lore.kernel.org/netdev/8bd80d3f-3e00-5e31-42a1-300ff29100ae@kaspersky.com/

The API for general `SOCK_SEQPACKET` sockets exposes a few incompatible mechanisms to tell EOF apart from 0-sized messages. These mechanisms are not found in `AF_UNIX` sockets.

As for `AF_UNIX+SOCK_SEQPACKET`, 0-sized payloads are valid and indistinguishable from the end of the stream.

According to POSIX the behaviour for Linux and BSD is wrong, but pointing to POSIX or changing the behaviour of current systems is useless (even harmful) at this point.

Emilua will just report EOF whenever a 0-sized read occurs.

If you control both sides of the communication channel, just avoid sending any 0-sized datagram and you're safe.

If you don't control the sending side, you might receive 0-sized datagrams that are in reality an attack to the system. If your program is the only receiver there's hardly any harm. However if you need to make sure the connection is closed when your program deems it as so, just call `shutdown("receive")` or `shutdown("both")` to make sure the connection is closed to every associated handle.

However don't let this small note scare you. `AF_UNIX+SOCK_SEQPACKET` sockets are a powerful IPC primitive that will save you from way worse concerns if your application needs a socket that is connection-oriented, preserves message boundaries, and delivers messages in the order that they were sent. `SOCK_STREAM` and `SOCK_DGRAM` will have their own caveats.

Functions

`new() → unix.seqpacket_socket`

```
new()                    ①
new(fd: file_descriptor) ②
```

① Default constructor.

② Converts a file descriptor into an `unix.seqpacket_socket` object.

`pair() → unix.seqpacket_socket, unix.seqpacket_socket`

Create a pair of connected sockets.

`open(self)`

Open the socket.

`bind(self, pathname: filesystem.path)`

Bind the socket to the given local endpoint.

`close(self)`

Close the socket.

Forward the call to the function with same name in Boost.Asio:

> Any asynchronous send, receive or connect operations will be cancelled immediately, and will complete with the `boost::asio::error::operation_aborted` error.

`cancel(self)`

Cancel all asynchronous operations associated with the socket.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error.

`assign(self, fd: file_descriptor)`

Assign an existing native socket to `self`.

`release(self) → file_descriptor`

Release ownership of the native descriptor implementation.

Forward the call to the function with same name in Boost.Asio:

> This function causes all outstanding asynchronous connect, send and receive operations to finish immediately, and the handlers for cancelled operations will be passed the `boost::asio::error::operation_aborted` error. Ownership of the native socket is then transferred to the caller.

`shutdown(self, what: string)`

Disable sends or receives on the socket.

`what` can be one of the following:

`"receive"`
: Shutdown the receive side of the socket.

`"send"`
: Shutdown the send side of the socket.

`"both"`
: Shutdown both send and receive on the socket.

`connect(self, pathname: filesystem.path)`

Initiate a connect operation and blocks current fiber until it completes or errs.

`disconnect(self)`

Dissolve the socket's association by resetting the socket's peer address (i.e. connect(3) will be called with an `AF_UNSPEC` address).

`receive(self, buffer: byte_span[, flags: integer]) → integer`

Receive a datagram and blocks current fiber until it completes or errs.

Returns the number of bytes read.

`flags` is `0` or an or-combination of values from unix.message_flag(3em).

`send(self, buffer: byte_span[, flags: integer]) → integer`

Send data on the seqpacket socket and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`flags` is `0` or an or-combination of values from unix.message_flag(3em).

`receive_with_fds(self, buffer: byte_span, maxfds: integer) → integer, file_descriptor[]`

Receive a datagram and blocks current fiber until it completes or errs.

Returns the number of bytes read plus the table containing the `fds` read.

`send_with_fds(self, buffer: byte_span, fds: file_descriptor[]) → integer`

Send data on the seqpacket socket and blocks current fiber until it completes or errs.

Returns the number of bytes written.

`set_option(self, opt: string, val)`

Set an option on the socket.

Currently available options are:

`"debug"`
: Check Boost.Asio documentation.

`"send_buffer_size"`
: Check Boost.Asio documentation.

`"receive_buffer_size"`
: Check Boost.Asio documentation.

`get_option(self, opt: string) → value`

Get an option from the socket.

Currently available options are:

`"debug"`
: Check Boost.Asio documentation.

`"send_buffer_size"`
: Check Boost.Asio documentation.

`"receive_buffer_size"`
: Check Boost.Asio documentation.

`"remote_credentials": { uid: integer, groups: integer[], pid: integer }`
: Returns the credentials from the remote process.

On Linux, `groups` don't include the supplementary group list.

`pid` is racy and you shouldn't use it for anything but debugging purposes.

`"remote_security_labels": { [string]: string }|string|nil`

(FreeBSD only) Returns the security labels associated with each policy for the remote process.

Optionally one may pass an extra argument to `get_option()` with either a list of strings for the policies of interest, or just a single string in case there's only one policy of interest.

`"remote_security_label": string`

(Linux only) Returns the SELinux security label associated with the remote process.

`io_control(self, command: string[, ···])`

Perform an IO control command on the socket.

Currently available commands are:

`"bytes_readable"`

Expects no arguments. Get the amount of data that can be read without blocking. Implements the `FIONREAD` IO control command.

Properties

`is_open: boolean`

Whether the socket is open.

`local_path: filesystem.path`

The local address endpoint of the socket.

`remote_path: filesystem.path`

The remote address endpoint of the socket.

file_descriptor

A file descriptor.

It cannot be created directly.

On Windows, `file_descriptor` is only implemented for pipes and `file.stream`.

Functions

`close(self)`

Closes the file descriptor w/o waiting for the GC.

It can only be called once.

`dup(self) → file_descriptor`

Creates a new file descriptor that refers to the same open file description.

`cap_get(self) → system.linux_capabilities`

See cap_get_fd(3).

`cap_set(self, caps: system.linux_capabilities)`

See cap_set_fd(3).

`cap_rights_limit(self, rights: string[])`

See cap_rights_limit(2).

Parameters:

- `rights: string[]`
  - `"accept"`
  - `"acl_check"`
  - `"acl_delete"`
  - `"acl_get"`
  - `"acl_set"`
  - `"bind"`
  - `"bindat"`
  - `"chflagsat"`
  - `"connect"`

- "connectat"
- "create"
- "event"
- "extattr_delete"
- "extattr_get"
- "extattr_list"
- "extattr_set"
- "fchdir"
- "fchflags"
- "fchmod"
- "fchmodat"
- "fchown"
- "fchownat"
- "fcntl"
- "fexecve"
- "flock"
- "fpathconf"
- "fsck"
- "fstat"
- "fstatat"
- "fstatfs"
- "fsync"
- "ftruncate"
- "futimes"
- "futimesat"
- "getpeername"
- "getsockname"
- "getsockopt"
- "ioctl"
- "kqueue"
- "kqueue_change"
- "kqueue_event"
- "linkat_source"
- "linkat_target"
- "listen"

- "lookup"
- "mac_get"
- "mac_set"
- "mkdirat"
- "mkfifoat"
- "mknodat"
- "mmap"
- "mmap_r"
- "mmap_rw"
- "mmap_rwx"
- "mmap_rx"
- "mmap_w"
- "mmap_wx"
- "mmap_x"
- "pdgetpid"
- "pdkill"
- "peeloff"
- "pread"
- "pwrite"
- "read"
- "recv"
- "renameat_source"
- "renameat_target"
- "seek"
- "sem_getvalue"
- "sem_post"
- "sem_wait"
- "send"
- "setsockopt"
- "shutdown"
- "symlinkat"
- "ttyhook"
- "unlinkat"
- "write"

`cap_ioctls_limit(self, cmds: integer[])`

See cap_ioctls_limit(2).

`cap_fcntls_limit(self, fcntlrights: string[])`

See cap_fcntls_limit(2).

Parameters:

- `fcntlrights: string[]`
  - `"getfl"`
  - `"setfl"`
  - `"getown"`
  - `"setown"`

Metamethods

`__tostring()`

Produces a string in the format `"/dev/fd/%i"` where `"%i"` is the integer value as seen by the OS.