LibUV in Lua

The luv project provides access to the multi-platform support library libuv in Lua code. It was primarily developed for the luvit project as the built-in `uv` module, but can be used in other Lua environments.

More information about the core libuv library can be found at the original libuv documentation page.

TCP Echo Server Example

Here is a small example showing a TCP echo server:

```lua
local uv = require("luv") -- "luv" when stand-alone, "uv" in luvi apps

local server = uv.new_tcp()
server:bind("127.0.0.1", 1337)
server:listen(128, function (err)
  assert(not err, err)
  local client = uv.new_tcp()
  server:accept(client)
  client:read_start(function (err, chunk)
    assert(not err, err)
    if chunk then
      client:write(chunk)
    else
      client:shutdown()
      client:close()
    end
  end)
end)
print("TCP server listening at 127.0.0.1 port 1337")
uv.run() -- an explicit run call is necessary outside of luvit
```

Module Layout

The luv library contains a single Lua module referred to hereafter as `uv` for simplicity. This module consists mostly of functions with names corresponding to their original libuv versions. For example, the libuv function `uv_tcp_bind` has a luv version at `uv.tcp_bind`. Currently, only two non-function fields exists: `uv.constants` and `uv.errno`, which are tables.

Functions vs Methods

In addition to having simple functions, luv provides an optional method-style API. For example, `uv.tcp_bind(server, host, port)` can alternatively be called as `server:bind(host, port)`. Note that the first argument `server` becomes the object and `tcp_` is removed from the function name. Method forms are documented below where they exist.

Synchronous vs Asynchronous Functions

Functions that accept a callback are asynchronous. These functions may immediately return results to the caller to indicate their initial status, but their final execution is deferred until at least the next libuv loop iteration. After completion, their callbacks are executed with any results passed to it.

Functions that do not accept a callback are synchronous. These functions immediately return their results to the caller.

Some (generally FS and DNS) functions can behave either synchronously or asynchronously. If a callback is provided to these functions, they behave asynchronously; if no callback is provided, they behave synchronously.

Pseudo-Types

Some unique types are defined. These are not actual types in Lua, but they are used here to facilitate documenting consistent behavior:

- `fail`: an assertable `nil, string, string` tuple (see Error handling)
- `callable`: a `function`; or a `table` or `userdata` with a `__call` metamethod
- `buffer`: a `string` or a sequential `table` of `string`s
- `threadargs`: variable arguments (`...`) of type `nil`, `boolean`, `number`, `string`, or `userdata`, numbers of argument limited to 9.

Contents

This documentation is mostly a retelling of the libuv API documentation within the context of luv's Lua API. Low-level implementation details and unexposed C functions and types are not documented here except for when they are relevant to behavior seen in the Lua module.

    - `uv_stream_t` — Stream handle
        - `uv_tcp_t` — TCP handle
        - `uv_pipe_t` — Pipe handle
        - `uv_tty_t` — TTY handle
    - `uv_udp_t` — UDP handle
    - `uv_fs_event_t` — FS Event handle
    - `uv_fs_poll_t` — FS Poll handle
- File system operations
- Thread pool work scheduling
- DNS utility functions
- Threading and synchronization utilities
- Miscellaneous utilities
- Metrics operations

Constants

As a Lua library, luv supports and encourages the use of lowercase strings to represent options. For example:

```
-- signal start with string input
uv.signal_start("sigterm", function(signame)
  print(signame) -- string output: "sigterm"
end)
```

However, luv also superficially exposes libuv constants in a Lua table at `uv.constants` where its keys are uppercase constant names and their associated values are integers defined internally by libuv. The values from this table may be supported as function arguments, but their use may not change the output type. For example:

```
-- signal start with integer input
uv.signal_start(uv.constants.SIGTERM, function(signame)
  print(signame) -- string output: "sigterm"
end)
```

The uppercase constants defined in `uv.constants` that have associated lowercase option strings are listed below.

Address Families

- `AF_UNIX`: "unix"
- `AF_INET`: "inet"
- `AF_INET6`: "inet6"
- `AF_IPX`: "ipx"
- `AF_NETLINK`: "netlink"

- `AF_X25`: "x25"
- `AF_AX25`: "as25"
- `AF_ATMPVC`: "atmpvc"
- `AF_APPLETALK`: "appletalk"
- `AF_PACKET`: "packet"

Signals

- `SIGHUP`: "sighup"
- `SIGINT`: "sigint"
- `SIGQUIT`: "sigquit"
- `SIGILL`: "sigill"
- `SIGTRAP`: "sigtrap"
- `SIGABRT`: "sigabrt"
- `SIGIOT`: "sigiot"
- `SIGBUS`: "sigbus"
- `SIGFPE`: "sigfpe"
- `SIGKILL`: "sigkill"
- `SIGUSR1`: "sigusr1"
- `SIGSEGV`: "sigsegv"
- `SIGUSR2`: "sigusr2"
- `SIGPIPE`: "sigpipe"
- `SIGALRM`: "sigalrm"
- `SIGTERM`: "sigterm"
- `SIGCHLD`: "sigchld"
- `SIGSTKFLT`: "sigstkflt"
- `SIGCONT`: "sigcont"
- `SIGSTOP`: "sigstop"
- `SIGTSTP`: "sigtstp"
- `SIGBREAK`: "sigbreak"
- `SIGTTIN`: "sigttin"
- `SIGTTOU`: "sigttou"
- `SIGURG`: "sigurg"
- `SIGXCPU`: "sigxcpu"
- `SIGXFSZ`: "sigxfsz"
- `SIGVTALRM`: "sigvtalrm"
- `SIGPROF`: "sigprof"
- `SIGWINCH`: "sigwinch"

- `SIGIO`: "sigio"
- `SIGPOLL`: "sigpoll"
- `SIGLOST`: "siglost"
- `SIGPWR`: "sigpwr"
- `SIGSYS`: "sigsys"

Socket Types

- `SOCK_STREAM`: "stream"
- `SOCK_DGRAM`: "dgram"
- `SOCK_SEQPACKET`: "seqpacket"
- `SOCK_RAW`: "raw"
- `SOCK_RDM`: "rdm"

TTY Modes

- `TTY_MODE_NORMAL`: "normal"
- `TTY_MODE_RAW`: "raw"
- `TTY_MODE_IO`: "io"

Error Handling

In libuv, errors are represented by negative numbered constants. While these constants are made available in the `uv.errno` table, they are not returned by luv funtions and the libuv functions used to handle them are not exposed. Instead, if an internal error is encountered, the failing luv function will return to the caller an assertable `nil, err, name` tuple:

- `nil` idiomatically indicates failure
- `err` is a string with the format `{name}: {message}`
  - `{name}` is the error name provided internally by `uv_err_name`
  - `{message}` is a human-readable message provided internally by `uv_strerror`
- `name` is the same string used to construct `err`

This tuple is referred to below as the `fail` pseudo-type.

When a function is called successfully, it will return either a value that is relevant to the operation of the function, or the integer `0` to indicate success, or sometimes nothing at all. These cases are documented below.

Below is a list of known error names and error strings. See libuv's error constants page for an original source.

- `E2BIG`: argument list too long.
- `EACCES`: permission denied.
- `EADDRINUSE`: address already in use.
- `EADDRNOTAVAIL`: address not available.

- `EAFNOSUPPORT`: address family not supported.
- `EAGAIN`: resource temporarily unavailable.
- `EAI_ADDRFAMILY`: address family not supported.
- `EAI_AGAIN`: temporary failure.
- `EAI_BADFLAGS`: bad ai_flags value.
- `EAI_BADHINTS`: invalid value for hints.
- `EAI_CANCELED`: request canceled.
- `EAI_FAIL`: permanent failure.
- `EAI_FAMILY`: ai_family not supported.
- `EAI_MEMORY`: out of memory.
- `EAI_NODATA`: no address.
- `EAI_NONAME`: unknown node or service.
- `EAI_OVERFLOW`: argument buffer overflow.
- `EAI_PROTOCOL`: resolved protocol is unknown.
- `EAI_SERVICE`: service not available for socket type.
- `EAI_SOCKTYPE`: socket type not supported.
- `EALREADY`: connection already in progress.
- `EBADF`: bad file descriptor.
- `EBUSY`: resource busy or locked.
- `ECANCELED`: operation canceled.
- `ECHARSET`: invalid Unicode character.
- `ECONNABORTED`: software caused connection abort.
- `ECONNREFUSED`: connection refused.
- `ECONNRESET`: connection reset by peer.
- `EDESTADDRREQ`: destination address required.
- `EEXIST`: file already exists.
- `EFAULT`: bad address in system call argument.
- `EFBIG`: file too large.
- `EHOSTUNREACH`: host is unreachable.
- `EINTR`: interrupted system call.
- `EINVAL`: invalid argument.
- `EIO`: i/o error.
- `EISCONN`: socket is already connected.
- `EISDIR`: illegal operation on a directory.
- `ELOOP`: too many symbolic links encountered.
- `EMFILE`: too many open files.
- `EMSGSIZE`: message too long.

- `ENAMETOOLONG`: name too long.
- `ENETDOWN`: network is down.
- `ENETUNREACH`: network is unreachable.
- `ENFILE`: file table overflow.
- `ENOBUFS`: no buffer space available.
- `ENODEV`: no such device.
- `ENOENT`: no such file or directory.
- `ENOMEM`: not enough memory.
- `ENONET`: machine is not on the network.
- `ENOPROTOOPT`: protocol not available.
- `ENOSPC`: no space left on device.
- `ENOSYS`: function not implemented.
- `ENOTCONN`: socket is not connected.
- `ENOTDIR`: not a directory.
- `ENOTEMPTY`: directory not empty.
- `ENOTSOCK`: socket operation on non-socket.
- `ENOTSUP`: operation not supported on socket.
- `EOVERFLOW`: value too large for defined data type.
- `EPERM`: operation not permitted.
- `EPIPE`: broken pipe.
- `EPROTO`: protocol error.
- `EPROTONOSUPPORT`: protocol not supported.
- `EPROTOTYPE`: protocol wrong type for socket.
- `ERANGE`: result too large.
- `EROFS`: read-only file system.
- `ESHUTDOWN`: cannot send after transport endpoint shutdown.
- `ESPIPE`: invalid seek.
- `ESRCH`: no such process.
- `ETIMEDOUT`: connection timed out.
- `ETXTBSY`: text file is busy.
- `EXDEV`: cross-device link not permitted.
- `UNKNOWN`: unknown error.
- `EOF`: end of file.
- `ENXIO`: no such device or address.
- `EMLINK`: too many links.
- `ENOTTY`: inappropriate ioctl for device.
- `EFTYPE`: inappropriate file type or format.

- `EILSEQ`: illegal byte sequence.
- `ESOCKTNOSUPPORT`: socket type not supported.

Version Checking

`uv.version()`

Returns the libuv version packed into a single integer. 8 bits are used for each component, with the patch number stored in the 8 least significant bits. For example, this would be 0x010203 in libuv 1.2.3.

Returns: `integer`

`uv.version_string()`

Returns the libuv version number as a string. For example, this would be "1.2.3" in libuv 1.2.3. For non-release versions, the version suffix is included.

Returns: `string`

`uv_loop_t` — Event loop

The event loop is the central part of libuv's functionality. It takes care of polling for I/O and scheduling callbacks to be run based on different sources of events.

In luv, there is an implicit uv loop for every Lua state that loads the library. You can use this library in an multi-threaded environment as long as each thread has it's own Lua state with its corresponding own uv loop. This loop is not directly exposed to users in the Lua module.

`uv.loop_close()`

Closes all internal loop resources. In normal execution, the loop will automatically be closed when it is garbage collected by Lua, so it is not necessary to explicitly call `loop_close()`. Call this function only after the loop has finished executing and all open handles and requests have been closed, or it will return `EBUSY`.

Returns: `0` or `fail`

`uv.run([mode])`

Parameters:

- `mode`: `string` or `nil` (default: `"default"`)

This function runs the event loop. It will act differently depending on the specified mode:

- `"default"`: Runs the event loop until there are no more active and referenced handles or requests. Returns `true` if `uv.stop()` was called and there are still active handles or requests. Returns `false` in all other cases.

- `"once"`: Poll for I/O once. Note that this function blocks if there are no pending callbacks. Returns `false` when done (no active handles or requests left), or `true` if more callbacks are expected (meaning you should run the event loop again sometime in the future).
- `"nowait"`: Poll for I/O once but don't block if there are no pending callbacks. Returns `false` if done (no active handles or requests left), or `true` if more callbacks are expected (meaning you should run the event loop again sometime in the future).

Returns: `boolean` or `fail`

Note: Luvit will implicitly call `uv.run()` after loading user code, but if you use the luv bindings directly, you need to call this after registering your initial set of event callbacks to start the event loop.

`uv.loop_configure(option, ...)`

Parameters:

- `option`: `string`
- `...`: depends on `option`, see below

Set additional loop options. You should normally call this before the first call to uv_run() unless mentioned otherwise.

Supported options:

- `"block_signal"`: Block a signal when polling for new events. The second argument to loop_configure() is the signal name (as a lowercase string) or the signal number. This operation is currently only implemented for `"sigprof"` signals, to suppress unnecessary wakeups when using a sampling profiler. Requesting other signals will fail with `EINVAL`.
- `"metrics_idle_time"`: Accumulate the amount of idle time the event loop spends in the event provider. This option is necessary to use `metrics_idle_time()`.

An example of a valid call to this function is:

```
uv.loop_configure("block_signal", "sigprof")
```

Returns: `0` or `fail`

Note: Be prepared to handle the `ENOSYS` error; it means the loop option is not supported by the platform.

`uv.loop_mode()`

If the loop is running, returns a string indicating the mode in use. If the loop is not running, `nil` is returned instead.

Returns: `string` or `nil`

`uv.loop_alive()`

Returns `true` if there are referenced active handles, active requests, or closing handles in the loop; otherwise, `false`.

Returns: `boolean` or `fail`

`uv.stop()`

Stop the event loop, causing `uv.run()` to end as soon as possible. This will happen not sooner than the next loop iteration. If this function was called before blocking for I/O, the loop won't block for I/O on this iteration.

Returns: Nothing.

`uv.backend_fd()`

Get backend file descriptor. Only kqueue, epoll, and event ports are supported.

This can be used in conjunction with `uv.run("nowait")` to poll in one thread and run the event loop's callbacks in another

Returns: `integer` or `nil`

Note: Embedding a kqueue fd in another kqueue pollset doesn't work on all platforms. It's not an error to add the fd but it never generates events.

`uv.backend_timeout()`

Get the poll timeout. The return value is in milliseconds, or -1 for no timeout.

Returns: `integer`

`uv.now()`

Returns the current timestamp in milliseconds. The timestamp is cached at the start of the event loop tick, see `uv.update_time()` for details and rationale.

The timestamp increases monotonically from some arbitrary point in time. Don't make assumptions about the starting point, you will only get disappointed.

Returns: `integer`

Note: Use `uv.hrtime()` if you need sub-millisecond granularity.

`uv.update_time()`

Update the event loop's concept of "now". Libuv caches the current time at the start of the event loop tick in order to reduce the number of time-related system calls.

You won't normally need to call this function unless you have callbacks that block the event loop for longer periods of time, where "longer" is somewhat subjective but probably on the order of a millisecond or more.

Returns: Nothing.

`uv.walk(callback)`

Parameters:

- `callback`: `callable`
  - `handle`: `userdata` for sub-type of `uv_handle_t`

Walk the list of handles: `callback` will be executed with each handle.

Returns: Nothing.

```lua
-- Example usage of uv.walk to close all handles that aren't already closing.
uv.walk(function (handle)
  if not handle:is_closing() then
    handle:close()
  end
end)
```

`uv_req_t` — Base request

`uv_req_t` is the base type for all libuv request types.

`uv.cancel(req)`

> method form `req:cancel()`

Parameters:

- `req`: `userdata` for sub-type of `uv_req_t`

Cancel a pending request. Fails if the request is executing or has finished executing. Only cancellation of `uv_fs_t`, `uv_getaddrinfo_t`, `uv_getnameinfo_t` and `uv_work_t` requests is currently supported.

Returns: `0` or `fail`

`uv.req_get_type(req)`

> method form `req:get_type()`

Parameters:

- `req`: `userdata` for sub-type of `uv_req_t`

Returns the name of the struct for a given request (e.g. `"fs"` for `uv_fs_t`) and the libuv enum integer for the request's type (`uv_req_type`).

Returns: `string, integer`

`uv_handle_t` — Base handle

`uv_handle_t` is the base type for all libuv handle types. All API functions defined here work with any handle type.

`uv.is_active(handle)`

> method form `handle:is_active()`

Parameters:

- `handle`: `userdata` for sub-type of `uv_handle_t`

Returns `true` if the handle is active, `false` if it's inactive. What "active" means depends on the type of handle:

- A `uv_async_t` handle is always active and cannot be deactivated, except by closing it with `uv.close()`.
- A `uv_pipe_t`, `uv_tcp_t`, `uv_udp_t`, etc. handle - basically any handle that deals with I/O - is active when it is doing something that involves I/O, like reading, writing, connecting, accepting new connections, etc.
- A `uv_check_t`, `uv_idle_t`, `uv_timer_t`, etc. handle is active when it has been started with a call to `uv.check_start()`, `uv.idle_start()`, `uv.timer_start()` etc. until it has been stopped with a call to its respective stop function.

Returns: `boolean` or `fail`

`uv.is_closing(handle)`

> method form `handle:is_closing()`

Parameters:

- `handle`: `userdata` for sub-type of `uv_handle_t`

Returns `true` if the handle is closing or closed, `false` otherwise.

Returns: `boolean` or `fail`

Note: This function should only be used between the initialization of the handle and the arrival of the close callback.

`uv.close(handle, [callback])`

> method form `handle:close([callback])`

Parameters:

- `handle`: `userdata` for sub-type of `uv_handle_t`
- `callback`: `callable` or `nil`

Request handle to be closed. `callback` will be called asynchronously after this call. This MUST be called on each handle before memory is released.

Handles that wrap file descriptors are closed immediately but `callback` will still be deferred to the next iteration of the event loop. It gives you a chance to free up any resources associated with the handle.

In-progress requests, like `uv_connect_t` or `uv_write_t`, are cancelled and have their callbacks called asynchronously with `ECANCELED`.

Returns: Nothing.

`uv.ref(handle)`

> method form `handle:ref()`

Parameters:

- `handle`: `userdata` for sub-type of `uv_handle_t`

Reference the given handle. References are idempotent, that is, if a handle is already referenced calling this function again will have no effect.

Returns: Nothing.

See Reference counting.

`uv.unref(handle)`

> method form `handle:unref()`

Parameters:

- `handle`: `userdata` for sub-type of `uv_handle_t`

Un-reference the given handle. References are idempotent, that is, if a handle is not referenced calling this function again will have no effect.

Returns: Nothing.

See Reference counting.

`uv.has_ref(handle)`

> method form `handle:has_ref()`

Parameters:

- `handle`: `userdata` for sub-type of `uv_handle_t`

Returns `true` if the handle referenced, `false` if not.

Returns: `boolean` or `fail`

See Reference counting.

`uv.send_buffer_size(handle, [size])`

> method form `handle:send_buffer_size([size])`

Parameters:

- `handle`: `userdata` for sub-type of `uv_handle_t`
- `size`: `integer` or `nil` (default: `0`)

Gets or sets the size of the send buffer that the operating system uses for the socket.

If `size` is omitted (or `0`), this will return the current send buffer size; otherwise, this will use `size` to set the new send buffer size.

This function works for TCP, pipe and UDP handles on Unix and for TCP and UDP handles on Windows.

Returns:

- `integer` or `fail` (if `size` is `nil` or `0`)
- `0` or `fail` (if `size` is not `nil` and not `0`)

Note: Linux will set double the size and return double the size of the original set value.

`uv.recv_buffer_size(handle, [size])`

> method form `handle:recv_buffer_size([size])`

Parameters:

- `handle`: `userdata` for sub-type of `uv_handle_t`
- `size`: `integer` or `nil` (default: `0`)

Gets or sets the size of the receive buffer that the operating system uses for the socket.

If `size` is omitted (or `0`), this will return the current send buffer size; otherwise, this will use `size` to set the new send buffer size.

This function works for TCP, pipe and UDP handles on Unix and for TCP and UDP handles on Windows.

Returns:

- `integer` or `fail` (if `size` is `nil` or `0`)
- `0` or `fail` (if `size` is not `nil` and not `0`)

Note: Linux will set double the size and return double the size of the original set value.

`uv.fileno(handle)`

> method form `handle:fileno()`

Parameters:

- `handle`: `userdata` for sub-type of `uv_handle_t`

Gets the platform dependent file descriptor equivalent.

The following handles are supported: TCP, pipes, TTY, UDP and poll. Passing any other handle type will fail with `EINVAL`.

If a handle doesn't have an attached file descriptor yet or the handle itself has been closed, this function will return `EBADF`.

Returns: `integer` or `fail`

Warning: Be very careful when using this function. libuv assumes it's in control of the file descriptor so any change to it may lead to malfunction.

`uv.handle_get_type(handle)`

> method form `handle:get_type()`

Parameters:

- `handle`: `userdata` for sub-type of `uv_handle_t`

Returns the name of the struct for a given handle (e.g. `"pipe"` for `uv_pipe_t`) and the libuv enum integer for the handle's type (`uv_handle_type`).

Returns: `string, integer`

Reference counting

The libuv event loop (if run in the default mode) will run until there are no active and referenced handles left. The user can force the loop to exit early by unreferencing handles which are active, for example by calling `uv.unref()` after calling `uv.timer_start()`.

A handle can be referenced or unreferenced, the refcounting scheme doesn't use a counter, so both operations are idempotent.

All handles are referenced when active by default, see `uv.is_active()` for a more detailed explanation on what being active involves.

`uv_timer_t` — Timer handle

> `uv_handle_t` functions also apply.

Timer handles are used to schedule callbacks to be called in the future.

`uv.new_timer()`

Creates and initializes a new `uv_timer_t`. Returns the Lua userdata wrapping it.

Returns: `uv_timer_t userdata` or `fail`

```lua
-- Creating a simple setTimeout wrapper
local function setTimeout(timeout, callback)
  local timer = uv.new_timer()
  timer:start(timeout, 0, function ()
    timer:stop()
    timer:close()
    callback()
  end)
  return timer
end

-- Creating a simple setInterval wrapper
local function setInterval(interval, callback)
  local timer = uv.new_timer()
```

```
  timer:start(interval, interval, function ()
    callback()
  end)
  return timer
end

-- And clearInterval
local function clearInterval(timer)
  timer:stop()
  timer:close()
end
```

`uv.timer_start(timer, timeout, repeat, callback)`

> method form `timer:start(timeout, repeat, callback)`

Parameters:

- `timer`: `uv_timer_t userdata`
- `timeout`: `integer`
- `repeat`: `integer`
- `callback`: `callable`

Start the timer. `timeout` and `repeat` are in milliseconds.

If `timeout` is zero, the callback fires on the next event loop iteration. If `repeat` is non-zero, the callback fires first after `timeout` milliseconds and then repeatedly after `repeat` milliseconds.

Returns: `0` or `fail`

`uv.timer_stop(timer)`

> method form `timer:stop()`

Parameters:

- `timer`: `uv_timer_t userdata`

Stop the timer, the callback will not be called anymore.

Returns: `0` or `fail`

`uv.timer_again(timer)`

> method form `timer:again()`

Parameters:

- `timer`: `uv_timer_t userdata`

Stop the timer, and if it is repeating restart it using the repeat value as the timeout. If the timer has never been started before it raises `EINVAL`.

Returns: `0` or `fail`

`uv.timer_set_repeat(timer, repeat)`

> method form `timer:set_repeat(repeat)`

Parameters:

- `timer`: `uv_timer_t userdata`
- `repeat`: `integer`

Set the repeat interval value in milliseconds. The timer will be scheduled to run on the given interval, regardless of the callback execution duration, and will follow normal timer semantics in the case of a time-slice overrun.

For example, if a 50 ms repeating timer first runs for 17 ms, it will be scheduled to run again 33 ms later. If other tasks consume more than the 33 ms following the first timer callback, then the callback will run as soon as possible.

Returns: Nothing.

`uv.timer_get_repeat(timer)`

> method form `timer:get_repeat()`

Parameters:

- `timer`: `uv_timer_t userdata`

Get the timer repeat value.

Returns: `integer`

`uv.timer_get_due_in(timer)`

> method form `timer:get_due_in()`

Parameters:

- `timer`: `uv_timer_t userdata`

Get the timer due value or 0 if it has expired. The time is relative to `uv.now()`.

Returns: `integer`

Note: New in libuv version 1.40.0.

`uv_prepare_t` — Prepare handle

> `uv_handle_t` functions also apply.

Prepare handles will run the given callback once per loop iteration, right before polling for I/O.

```lua
local prepare = uv.new_prepare()
prepare:start(function()
  print("Before I/O polling")
end)
```

`uv.new_prepare()`

Creates and initializes a new `uv_prepare_t`. Returns the Lua userdata wrapping it.

Returns: `uv_prepare_t userdata`

`uv.prepare_start(prepare, callback)`

> method form `prepare:start(callback)`

Parameters:

- `prepare`: `uv_prepare_t userdata`
- `callback`: `callable`

Start the handle with the given callback.

Returns: `0` or `fail`

`uv.prepare_stop(prepare)`

> method form `prepare:stop()`

Parameters:

- `prepare`: `uv_prepare_t userdata`

Stop the handle, the callback will no longer be called.

Returns: `0` or `fail`

`uv_check_t` — Check handle

> `uv_handle_t` functions also apply.

Check handles will run the given callback once per loop iteration, right after polling for I/O.

```lua
local check = uv.new_check()
check:start(function()
  print("After I/O polling")
end)
```

`uv.new_check()`

Creates and initializes a new `uv_check_t`. Returns the Lua userdata wrapping it.

Returns: `uv_check_t userdata`

`uv.check_start(check, callback)`

> method form `check:start(callback)`

Parameters:

- `check` : `uv_check_t userdata`
- `callback` : `callable`

Start the handle with the given callback.

Returns: `0` or `fail`

`uv.check_stop(check)`

> method form `check:stop()`

Parameters:

- `check` : `uv_check_t userdata`

Stop the handle, the callback will no longer be called.

Returns: `0` or `fail`

`uv_idle_t` — Idle handle

> `uv_handle_t` functions also apply.

Idle handles will run the given callback once per loop iteration, right before the `uv_prepare_t` handles.

Note: The notable difference with prepare handles is that when there are active idle handles, the loop will perform a zero timeout poll instead of blocking for I/O.

Warning: Despite the name, idle handles will get their callbacks called on every loop iteration, not when the loop is actually "idle".

```
local idle = uv.new_idle()
idle:start(function()
  print("Before I/O polling, no blocking")
end)
```

`uv.new_idle()`

Creates and initializes a new `uv_idle_t`. Returns the Lua userdata wrapping it.

Returns: `uv_idle_t userdata`

`uv.idle_start(idle, callback)`

> method form `idle:start(callback)`

Parameters:

- `idle`: `uv_idle_t userdata`
- `callback`: `callable`

Start the handle with the given callback.

Returns: `0` or `fail`

`uv.idle_stop(check)`

> method form `idle:stop()`

Parameters:

- `idle`: `uv_idle_t userdata`

Stop the handle, the callback will no longer be called.

Returns: `0` or `fail`

`uv_async_t` — Async handle

> `uv_handle_t` functions also apply.

Async handles allow the user to "wakeup" the event loop and get a callback called from another thread.

```
local async
async = uv.new_async(function()
  print("async operation ran")
  async:close()
end)

async:send()
```

`uv.new_async(callback)`

Parameters:

- `callback`: `callable`
    - `...`: `threadargs` passed to/from `uv.async_send(async, ...)`

Creates and initializes a new `uv_async_t`. Returns the Lua userdata wrapping it.

Returns: `uv_async_t userdata` or `fail`

Note: Unlike other handle initialization functions, this immediately starts the handle.

`uv.async_send(async, ...)`

> method form `async:send(...)`

Parameters:

- `async`: `uv_async_t userdata`
- `...`: `threadargs`

Wakeup the event loop and call the async handle's callback.

Returns: `0` or `fail`

Note: It's safe to call this function from any thread. The callback will be called on the loop thread.

Warning: libuv will coalesce calls to `uv.async_send(async)`, that is, not every call to it will yield an execution of the callback. For example: if `uv.async_send()` is called 5 times in a row before the callback is called, the callback will only be called once. If `uv.async_send()` is called again after the callback was called, it will be called again.

`uv_poll_t` — Poll handle

> `uv_handle_t` functions also apply.

Poll handles are used to watch file descriptors for readability and writability, similar to the purpose of poll(2).

The purpose of poll handles is to enable integrating external libraries that rely on the event loop to signal it about the socket status changes, like c-ares or libssh2. Using `uv_poll_t` for any other purpose is not recommended; `uv_tcp_t`, `uv_udp_t`, etc. provide an implementation that is faster and more scalable than what can be achieved with `uv_poll_t`, especially on Windows.

It is possible that poll handles occasionally signal that a file descriptor is readable or writable even when it isn't. The user should therefore always be prepared to handle EAGAIN or equivalent when it attempts to read from or write to the fd.

It is not okay to have multiple active poll handles for the same socket, this can cause libuv to busyloop or otherwise malfunction.

The user should not close a file descriptor while it is being polled by an active poll handle. This can cause the handle to report an error, but it might also start polling another socket. However the fd can be safely closed immediately after a call to `uv.poll_stop()` or `uv.close()`.

Note: On windows only sockets can be polled with poll handles. On Unix any file descriptor that would be accepted by poll(2) can be used.

`uv.new_poll(fd)`

Parameters:

- `fd`: `integer`

Initialize the handle using a file descriptor.

The file descriptor is set to non-blocking mode.

Returns: `uv_poll_t userdata` or `fail`

`uv.new_socket_poll(fd)`

Parameters:

- `fd`: `integer`

Initialize the handle using a socket descriptor. On Unix this is identical to `uv.new_poll()`. On windows it takes a SOCKET handle.

The socket is set to non-blocking mode.

Returns: `uv_poll_t userdata` or `fail`

`uv.poll_start(poll, events, callback)`

> method form `poll:start(events, callback)`

Parameters:

- `poll`: `uv_poll_t userdata`
- `events`: `string` or `nil` (default: `"rw"`)
- `callback`: `callable`
  - `err`: `nil` or `string`
  - `events`: `string` or `nil`

Starts polling the file descriptor. `events` are: `"r"`, `"w"`, `"rw"`, `"d"`, `"rd"`, `"wd"`, `"rwd"`, `"p"`, `"rp"`, `"wp"`, `"rwp"`, `"dp"`, `"rdp"`, `"wdp"`, or `"rwdp"` where `r` is `READABLE`, `w` is `WRITABLE`, `d` is `DISCONNECT`, and `p` is `PRIORITIZED`. As soon as an event is detected the callback will be called with status set to 0, and the detected events set on the events field.

The user should not close the socket while the handle is active. If the user does that anyway, the callback may be called reporting an error status, but this is not guaranteed.

Returns: `0` or `fail`

Note Calling `uv.poll_start()` on a handle that is already active is fine. Doing so will update the events mask that is being watched for.

`uv.poll_stop(poll)`

> method form `poll:stop()`

Parameters:

- `poll`: `uv_poll_t userdata`

Stop polling the file descriptor, the callback will no longer be called.

Returns: `0` or `fail`

`uv_signal_t` — Signal handle

> `uv_handle_t` functions also apply.

Signal handles implement Unix style signal handling on a per-event loop bases.

Windows Notes:

Reception of some signals is emulated on Windows:

- SIGINT is normally delivered when the user presses CTRL+C. However, like on Unix, it is not generated when terminal raw mode is enabled.
- SIGBREAK is delivered when the user pressed CTRL + BREAK.
- SIGHUP is generated when the user closes the console window. On SIGHUP the program is given approximately 10 seconds to perform cleanup. After that Windows will unconditionally terminate it.
- SIGWINCH is raised whenever libuv detects that the console has been resized. SIGWINCH is emulated by libuv when the program uses a uv_tty_t handle to write to the console. SIGWINCH may not always be delivered in a timely manner; libuv will only detect size changes when the cursor is being moved. When a readable `uv_tty_t` handle is used in raw mode, resizing the console buffer will also trigger a SIGWINCH signal.
- Watchers for other signals can be successfully created, but these signals are never received. These signals are: SIGILL, SIGABRT, SIGFPE, SIGSEGV, SIGTERM and SIGKILL.
- Calls to raise() or abort() to programmatically raise a signal are not detected by libuv; these will not trigger a signal watcher.

Unix Notes:

- SIGKILL and SIGSTOP are impossible to catch.
- Handling SIGBUS, SIGFPE, SIGILL or SIGSEGV via libuv results into undefined behavior.
- SIGABRT will not be caught by libuv if generated by abort(), e.g. through assert().
- On Linux SIGRT0 and SIGRT1 (signals 32 and 33) are used by the NPTL pthreads library to manage threads. Installing watchers for those signals will lead to unpredictable behavior and is strongly discouraged. Future versions of libuv may simply reject them.

```lua
-- Create a new signal handler
local signal = uv.new_signal()
-- Define a handler function
uv.signal_start(signal, "sigint", function(signame)
  print("got " .. signame .. ", shutting down")
  os.exit(1)
end)
```

`uv.new_signal()`

Creates and initializes a new `uv_signal_t`. Returns the Lua userdata wrapping it.

Returns: `uv_signal_t userdata` or `fail`

`uv.signal_start(signal, signame, callback)`

> method form `signal:start(signame, callback)`

Parameters:

- `signal`: `uv_signal_t userdata`
- `signame`: `string` or `integer`
- `callback`: `callable`
    - `signame`: `string`

Start the handle with the given callback, watching for the given signal.

See Constants for supported `signame` input and output values.

Returns: `0` or `fail`

`uv.signal_start_oneshot(signal, signame, callback)`

> method form `signal:start_oneshot(signame, callback)`

Parameters:

- `signal`: `uv_signal_t userdata`
- `signame`: `string` or `integer`
- `callback`: `callable`
    - `signame`: `string`

Same functionality as `uv.signal_start()` but the signal handler is reset the moment the signal is received.

See Constants for supported `signame` input and output values.

Returns: `0` or `fail`

`uv.signal_stop(signal)`

> method form `signal:stop()`

Parameters:

- `signal`: `uv_signal_t userdata`

Stop the handle, the callback will no longer be called.

Returns: `0` or `fail`

`uv_process_t` — Process handle

> `uv_handle_t` functions also apply.

Process handles will spawn a new process and allow the user to control it and establish communication channels with it using streams.

`uv.disable_stdio_inheritance()`

Disables inheritance for file descriptors / handles that this process inherited from its parent. The effect is that child processes spawned by this process don't accidentally inherit these handles.

It is recommended to call this function as early in your program as possible, before the inherited file descriptors can be closed or duplicated.

Returns: Nothing.

Note: This function works on a best-effort basis: there is no guarantee that libuv can discover all file descriptors that were inherited. In general it does a better job on Windows than it does on Unix.

`uv.spawn(path, options, on_exit)`

Parameters:

- `path`: `string`
- `options`: `table` (see below)
- `on_exit`: `callable`
  - `code`: `integer`
  - `signal`: `integer`

Initializes the process handle and starts the process. If the process is successfully spawned, this function will return the handle and pid of the child process.

Possible reasons for failing to spawn would include (but not be limited to) the file to execute not existing, not having permissions to use the setuid or setgid specified, or not having enough memory to allocate for the new process.

```
local stdin = uv.new_pipe()
local stdout = uv.new_pipe()
local stderr = uv.new_pipe()
```

```lua
print("stdin", stdin)
print("stdout", stdout)
print("stderr", stderr)

local handle, pid = uv.spawn("cat", {
  stdio = {stdin, stdout, stderr}
}, function(code, signal) -- on exit
  print("exit code", code)
  print("exit signal", signal)
end)

print("process opened", handle, pid)

uv.read_start(stdout, function(err, data)
  assert(not err, err)
  if data then
    print("stdout chunk", stdout, data)
  else
    print("stdout end", stdout)
  end
end)

uv.read_start(stderr, function(err, data)
  assert(not err, err)
  if data then
    print("stderr chunk", stderr, data)
  else
    print("stderr end", stderr)
  end
end)

uv.write(stdin, "Hello World")

uv.shutdown(stdin, function()
  print("stdin shutdown", stdin)
  uv.close(handle, function()
    print("process closed", handle, pid)
  end)
end)
```

The `options` table accepts the following fields:

- `options.args` - Command line arguments as a list of strings. The first string should *not* be the path to the program, since that is already provided via `path`. On Windows, this uses CreateProcess which concatenates the arguments into a string. This can cause some strange errors (see `options.verbatim` below for Windows).
- `options.stdio` - Set the file descriptors that will be made available to the child process. The convention is that the first entries are stdin, stdout, and stderr. (**Note**: On Windows, file descriptors after the third are available to the child process only if the child processes uses the MSVCRT runtime.)
- `options.env` - Set environment variables for the new process.

- `options.cwd` - Set the current working directory for the sub-process.
- `options.uid` - Set the child process' user id.
- `options.gid` - Set the child process' group id.
- `options.verbatim` - If true, do not wrap any arguments in quotes, or perform any other escaping, when converting the argument list into a command line string. This option is only meaningful on Windows systems. On Unix it is silently ignored.
- `options.detached` - If true, spawn the child process in a detached state - this will make it a process group leader, and will effectively enable the child to keep running after the parent exits. Note that the child process will still keep the parent's event loop alive unless the parent process calls `uv.unref()` on the child's process handle.
- `options.hide` - If true, hide the subprocess console window that would normally be created. This option is only meaningful on Windows systems. On Unix it is silently ignored.

The `options.stdio` entries can take many shapes.

- If they are numbers, then the child process inherits that same zero-indexed fd from the parent process.
- If `uv_stream_t` handles are passed in, those are used as a read-write pipe or inherited stream depending if the stream has a valid fd.
- Including `nil` placeholders means to ignore that fd in the child process.

When the child process exits, `on_exit` is called with an exit code and signal.

Returns: `uv_process_t userdata`, `integer`

`uv.process_kill(process, signame)`

> method form `process:kill(signame)`

Parameters:

- `process`: `uv_process_t userdata`
- `signame`: `string` or `integer` or `nil` (default: `sigterm`)

Sends the specified signal to the given process handle. Check the documentation on `uv_signal_t` for signal support, specially on Windows.

See Constants for supported `signame` input values.

Returns: `0` or `fail`

`uv.kill(pid, signame)`

Parameters:

- `pid`: `integer`
- `signame`: `string` or `integer` or `nil` (default: `sigterm`)

Sends the specified signal to the given PID. Check the documentation on `uv_signal_t` for signal support, specially on Windows.

See Constants for supported `signame` input values.

Returns: `0` or `fail`

`uv.process_get_pid(process)`

> method form `process:get_pid()`

Parameters:

- `process`: `uv_process_t userdata`

Returns the handle's pid.

Returns: `integer`

`uv_stream_t` — Stream handle

> `uv_handle_t` functions also apply.

Stream handles provide an abstraction of a duplex communication channel. `uv_stream_t` is an abstract type, libuv provides 3 stream implementations in the form of `uv_tcp_t`, `uv_pipe_t` and `uv_tty_t`.

`uv.shutdown(stream, [callback])`

> method form `stream:shutdown([callback])`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`
- `callback`: `callable` or `nil`
  - `err`: `nil` or `string`

Shutdown the outgoing (write) side of a duplex stream. It waits for pending write requests to complete. The callback is called after shutdown is complete.

Returns: `uv_shutdown_t userdata` or `fail`

`uv.listen(stream, backlog, callback)`

> method form `stream:listen(backlog, callback)`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`
- `backlog`: `integer`
- `callback`: `callable`
  - `err`: `nil` or `string`

Start listening for incoming connections. `backlog` indicates the number of connections the kernel might queue, same as `listen(2)`. When a new incoming connection is received the callback is called.

Returns: `0` or `fail`

`uv.accept(stream, client_stream)`

> method form `stream:accept(client_stream)`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`
- `client_stream`: `userdata` for sub-type of `uv_stream_t`

This call is used in conjunction with `uv.listen()` to accept incoming connections. Call this function after receiving a callback to accept the connection.

When the connection callback is called it is guaranteed that this function will complete successfully the first time. If you attempt to use it more than once, it may fail. It is suggested to only call this function once per connection call.

Returns: `0` or `fail`

```
server:listen(128, function (err)
  local client = uv.new_tcp()
  server:accept(client)
end)
```

`uv.read_start(stream, callback)`

> method form `stream:read_start(callback)`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`
- `callback`: `callable`
    - `err`: `nil` or `string`
    - `data`: `string` or `nil`

Read data from an incoming stream. The callback will be made several times until there is no more data to read or `uv.read_stop()` is called. When we've reached EOF, `data` will be `nil`.

Returns: `0` or `fail`

```
stream:read_start(function (err, chunk)
  if err then
    -- handle read error
  elseif chunk then
    -- handle data
  else
    -- handle disconnect
  end
end)
```

`uv.read_stop(stream)`

> method form `stream:read_stop()`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`

Stop reading data from the stream. The read callback will no longer be called.

This function is idempotent and may be safely called on a stopped stream.

Returns: `0` or `fail`

`uv.write(stream, data, [callback])`

> method form `stream:write(data, [callback])`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`
- `data`: `buffer`
- `callback`: `callable` or `nil`
    - `err`: `nil` or `string`

Write data to stream.

`data` can either be a Lua string or a table of strings. If a table is passed in, the C backend will use writev to send all strings in a single system call.

The optional `callback` is for knowing when the write is complete.

Returns: `uv_write_t userdata` or `fail`

`uv.write2(stream, data, send_handle, [callback])`

> method form `stream:write2(data, send_handle, [callback])`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`
- `data`: `buffer`
- `send_handle`: `userdata` for sub-type of `uv_stream_t`
- `callback`: `callable` or `nil`
    - `err`: `nil` or `string`

Extended write function for sending handles over a pipe. The pipe must be initialized with `ipc` option `true`.

Returns: `uv_write_t userdata` or `fail`

Note: `send_handle` must be a TCP socket or pipe, which is a server or a connection (listening or connected state). Bound sockets or pipes will be assumed to be servers.

`uv.try_write(stream, data)`

> method form `stream:try_write(data)`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`
- `data`: `buffer`

Same as `uv.write()`, but won't queue a write request if it can't be completed immediately.

Will return number of bytes written (can be less than the supplied buffer size).

Returns: `integer` or `fail`

`uv.try_write2(stream, data, send_handle)`

> method form `stream:try_write2(data, send_handle)`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`
- `data`: `buffer`
- `send_handle`: `userdata` for sub-type of `uv_stream_t`

Like `uv.write2()`, but with the properties of `uv.try_write()`. Not supported on Windows, where it returns `UV_EAGAIN`.

Will return number of bytes written (can be less than the supplied buffer size).

Returns: `integer` or `fail`

`uv.is_readable(stream)`

> method form `stream:is_readable()`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`

Returns `true` if the stream is readable, `false` otherwise.

Returns: `boolean`

`uv.is_writable(stream)`

> method form `stream:is_writable()`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`

Returns `true` if the stream is writable, `false` otherwise.

Returns: `boolean`

`uv.stream_set_blocking(stream, blocking)`

> method form `stream:set_blocking(blocking)`

Parameters:

- `stream`: `userdata` for sub-type of `uv_stream_t`
- `blocking`: `boolean`

Enable or disable blocking mode for a stream.

When blocking mode is enabled all writes complete synchronously. The interface remains unchanged otherwise, e.g. completion or failure of the operation will still be reported through a callback which is made asynchronously.

Returns: `0` or `fail`

Warning: Relying too much on this API is not recommended. It is likely to change significantly in the future. Currently this only works on Windows and only for `uv_pipe_t` handles. Also libuv currently makes no ordering guarantee when the blocking mode is changed after write requests have already been submitted. Therefore it is recommended to set the blocking mode immediately after opening or creating the stream.

`uv.stream_get_write_queue_size()`

> method form `stream:get_write_queue_size()`

Returns the stream's write queue size.

Returns: `integer`

`uv_tcp_t` — TCP handle

> `uv_handle_t` and `uv_stream_t` functions also apply.

TCP handles are used to represent both TCP streams and servers.

`uv.new_tcp([flags])`

Parameters:

- `flags`: `string` or `integer` or `nil`

Creates and initializes a new `uv_tcp_t`. Returns the Lua userdata wrapping it.

If set, `flags` must be a valid address family. See Constants for supported address family input values.

Returns: `uv_tcp_t userdata` or `fail`

`uv.tcp_open(tcp, sock)`

> method form `tcp:open(sock)`

Parameters:

- `tcp`: `uv_tcp_t userdata`

- `sock`: `integer`

Open an existing file descriptor or SOCKET as a TCP handle.

Returns: `0` or `fail`

Note: The passed file descriptor or SOCKET is not checked for its type, but it's required that it represents a valid stream socket.

`uv.tcp_nodelay(tcp, enable)`

> method form `tcp:nodelay(enable)`

Parameters:

- `tcp`: `uv_tcp_t userdata`
- `enable`: `boolean`

Enable / disable Nagle's algorithm.

Returns: `0` or `fail`

`uv.tcp_keepalive(tcp, enable, [delay])`

> method form `tcp:keepalive(enable, [delay])`

Parameters:

- `tcp`: `uv_tcp_t userdata`
- `enable`: `boolean`
- `delay`: `integer` or `nil`

Enable / disable TCP keep-alive. `delay` is the initial delay in seconds, ignored when enable is `false`.

Returns: `0` or `fail`

`uv.tcp_simultaneous_accepts(tcp, enable)`

> method form `tcp:simultaneous_accepts(enable)`

Parameters:

- `tcp`: `uv_tcp_t userdata`
- `enable`: `boolean`

Enable / disable simultaneous asynchronous accept requests that are queued by the operating system when listening for new TCP connections.

This setting is used to tune a TCP server for the desired performance. Having simultaneous accepts can significantly improve the rate of accepting connections (which is why it is enabled by default) but may lead to uneven load distribution in multi-process setups.

Returns: `0` or `fail`

`uv.tcp_bind(tcp, host, port, [flags])`

> method form `tcp:bind(host, port, [flags])`

Parameters:

- `tcp` : `uv_tcp_t userdata`
- `host` : `string`
- `port` : `integer`
- `flags` : `table` or `nil`
  - `ipv6only` : `boolean`

Bind the handle to an host and port. `host` should be an IP address and not a domain name. Any `flags` are set with a table with field `ipv6only` equal to `true` or `false`.

When the port is already taken, you can expect to see an `EADDRINUSE` error from either `uv.tcp_bind()`, `uv.listen()` or `uv.tcp_connect()`. That is, a successful call to this function does not guarantee that the call to `uv.listen()` or `uv.tcp_connect()` will succeed as well.

Use a port of `0` to let the OS assign an ephemeral port. You can look it up later using `uv.tcp_getsockname()`.

Returns: `0` or `fail`

`uv.tcp_getpeername(tcp)`

> method form `tcp:getpeername()`

Parameters:

- `tcp` : `uv_tcp_t userdata`

Get the address of the peer connected to the handle.

See Constants for supported address `family` output values.

Returns: `table` or `fail`

- `ip` : `string`
- `family` : `string`
- `port` : `integer`

`uv.tcp_getsockname(tcp)`

> method form `tcp:getsockname()`

Parameters:

- `tcp` : `uv_tcp_t userdata`

Get the current address to which the handle is bound.

See Constants for supported address `family` output values.

Returns: `table` or `fail`

- `ip` : `string`
- `family` : `string`
- `port` : `integer`

`uv.tcp_connect(tcp, host, port, callback)`

> method form `tcp:connect(host, port, callback)`

Parameters:

- `tcp` : `uv_tcp_t userdata`
- `host` : `string`
- `port` : `integer`
- `callback` : `callable`
  - `err` : `nil` or `string`

Establish an IPv4 or IPv6 TCP connection.

Returns: `uv_connect_t userdata` or `fail`

```lua
local client = uv.new_tcp()
client:connect("127.0.0.1", 8080, function (err)
  -- check error and carry on.
end)
```

`uv.tcp_write_queue_size(tcp)`

> method form `tcp:write_queue_size()`

Deprecated: Please use `uv.stream_get_write_queue_size()` instead.

`uv.tcp_close_reset([callback])`

> method form `tcp:close_reset([callback])`

Parameters:

- `tcp` : `uv_tcp_t userdata`
- `callback` : `callable` or `nil`

Resets a TCP connection by sending a RST packet. This is accomplished by setting the SO_LINGER socket option with a linger interval of zero and then calling `uv.close()`. Due to some platform inconsistencies, mixing of `uv.shutdown()` and `uv.tcp_close_reset()` calls is not allowed.

Returns: `0` or `fail`

`uv.socketpair([socktype], [protocol], [flags1], [flags2])`

Parameters:

- `socktype` : `string` , `integer` or `nil` (default: `stream`)
- `protocol` : `string` , `integer` or `nil` (default: 0)
- `flags1` : `table` or `nil`
  - `nonblock` : `boolean` (default: `false`)
- `flags2` : `table` or `nil`
  - `nonblock` : `boolean` (default: `false`)

Create a pair of connected sockets with the specified properties. The resulting handles can be passed to `uv.tcp_open` , used with `uv.spawn` , or for any other purpose.

See Constants for supported `socktype` input values.

When `protocol` is set to 0 or nil, it will be automatically chosen based on the socket's domain and type. When `protocol` is specified as a string, it will be looked up using the `getprotobyname(3)` function (examples: `"ip"` , `"icmp"` , `"tcp"` , `"udp"` , etc).

Flags:

- `nonblock` : Opens the specified socket handle for `OVERLAPPED` or `FIONBIO` / `O_NONBLOCK` I/O usage. This is recommended for handles that will be used by libuv, and not usually recommended otherwise.

Equivalent to `socketpair(2)` with a domain of `AF_UNIX` .

Returns: `table` or `fail`

- `[1, 2]` : `integer` (file descriptor)

```lua
-- Simple read/write with tcp
local fds = uv.socketpair(nil, nil, {nonblock=true}, {nonblock=true})

local sock1 = uv.new_tcp()
sock1:open(fds[1])

local sock2 = uv.new_tcp()
sock2:open(fds[2])

sock1:write("hello")
sock2:read_start(function(err, chunk)
  assert(not err, err)
  print(chunk)
end)
```

`uv_pipe_t` — Pipe handle

> `uv_handle_t` and `uv_stream_t` functions also apply.

Pipe handles provide an abstraction over local domain sockets on Unix and named pipes on Windows.

```lua
local pipe = uv.new_pipe(false)

pipe:bind('/tmp/sock.test')

pipe:listen(128, function()
  local client = uv.new_pipe(false)
  pipe:accept(client)
  client:write("hello!\n")
  client:close()
end)
```

`uv.new_pipe([ipc])`

Parameters:

- `ipc`: `boolean` or `nil` (default: `false`)

Creates and initializes a new `uv_pipe_t`. Returns the Lua userdata wrapping it. The `ipc` argument is a boolean to indicate if this pipe will be used for handle passing between processes.

Returns: `uv_pipe_t userdata` or `fail`

`uv.pipe_open(pipe, fd)`

> method form `pipe:open(fd)`

Parameters:

- `pipe`: `uv_pipe_t userdata`
- `fd`: `integer`

Open an existing file descriptor or `uv_handle_t` as a pipe.

Returns: `0` or `fail`

Note: The file descriptor is set to non-blocking mode.

`uv.pipe_bind(pipe, name)`

> method form `pipe:bind(name)`

Parameters:

- `pipe`: `uv_pipe_t userdata`
- `name`: `string`

Bind the pipe to a file path (Unix) or a name (Windows).

Returns: `0` or `fail`

Note: Paths on Unix get truncated to sizeof(sockaddr_un.sun_path) bytes, typically between 92 and 108 bytes.

`uv.pipe_connect(pipe, name, [callback])`

> method form `pipe:connect(name, [callback])`

Parameters:

- `pipe`: `uv_pipe_t userdata`
- `name`: `string`
- `callback`: `callable` or `nil`
    - `err`: `nil` or `string`

Connect to the Unix domain socket or the named pipe.

Returns: `uv_connect_t userdata` or `fail`

Note: Paths on Unix get truncated to sizeof(sockaddr_un.sun_path) bytes, typically between 92 and 108 bytes.

`uv.pipe_getsockname(pipe)`

> method form `pipe:getsockname()`

Parameters:

- `pipe`: `uv_pipe_t userdata`

Get the name of the Unix domain socket or the named pipe.

Returns: `string` or `fail`

`uv.pipe_getpeername(pipe)`

> method form `pipe:getpeername()`

Parameters:

- `pipe`: `uv_pipe_t userdata`

Get the name of the Unix domain socket or the named pipe to which the handle is connected.

Returns: `string` or `fail`

`uv.pipe_pending_instances(pipe, count)`

> method form `pipe:pending_instances(count)`

Parameters:

- `pipe`: `uv_pipe_t userdata`
- `count`: `integer`

Set the number of pending pipe instance handles when the pipe server is waiting for connections.

Returns: Nothing.

Note: This setting applies to Windows only.

`uv.pipe_pending_count(pipe)`

> method form `pipe:pending_count()`

Parameters:

- `pipe`: `uv_pipe_t userdata`

Returns the pending pipe count for the named pipe.

Returns: `integer`

`uv.pipe_pending_type(pipe)`

> method form `pipe:pending_type()`

Parameters:

- `pipe`: `uv_pipe_t userdata`

Used to receive handles over IPC pipes.

First - call `uv.pipe_pending_count()`, if it's > 0 then initialize a handle of the given type, returned by `uv.pipe_pending_type()` and call `uv.accept(pipe, handle)`.

Returns: `string`

`uv.pipe_chmod(pipe, flags)`

> method form `pipe:chmod(flags)`

Parameters:

- `pipe`: `uv_pipe_t userdata`
- `flags`: `string`

Alters pipe permissions, allowing it to be accessed from processes run by different users. Makes the pipe writable or readable by all users. `flags` are: `"r"`, `"w"`, `"rw"`, or `"wr"` where `r` is `READABLE` and `w` is `WRITABLE`. This function is blocking.

Returns: `0` or `fail`

`uv.pipe(read_flags, write_flags)`

Parameters:

- `read_flags`: `table` or `nil`
  - `nonblock`: `boolean` (default: `false`)
- `write_flags`: `table` or `nil`
  - `nonblock`: `boolean` (default: `false`)

Create a pair of connected pipe handles. Data may be written to the `write` fd and read from the `read` fd. The resulting handles can be passed to `pipe_open`, used with `spawn`, or for any other purpose.

Flags:

- `nonblock` : Opens the specified socket handle for `OVERLAPPED` or `FIONBIO` / `O_NONBLOCK` I/O usage. This is recommended for handles that will be used by libuv, and not usually recommended otherwise.

Equivalent to `pipe(2)` with the `O_CLOEXEC` flag set.

Returns: `table` or `fail`

- `read` : `integer` (file descriptor)
- `write` : `integer` (file descriptor)

```lua
-- Simple read/write with pipe_open
local fds = uv.pipe({nonblock=true}, {nonblock=true})

local read_pipe = uv.new_pipe()
read_pipe:open(fds.read)

local write_pipe = uv.new_pipe()
write_pipe:open(fds.write)

write_pipe:write("hello")
read_pipe:read_start(function(err, chunk)
  assert(not err, err)
  print(chunk)
end)
```

`uv.pipe_bind2(pipe, name, [flags])`

> method form `pipe:bind2(name, [flags])`

Parameters:

- `pipe` : `uv_pipe_t userdata`
- `name` : `string`
- `flags` : `integer` or `table` or `nil` (default: 0)

Bind the pipe to a file path (Unix) or a name (Windows).

`Flags` :

- If `type(flags)` is `number`, it must be `0` or `uv.constants.PIPE_NO_TRUNCATE`.
- If `type(flags)` is `table`, it must be `{}` or `{ no_truncate = true|false }`.
- If `type(flags)` is `nil`, it use default value `0`.
- Returns `EINVAL` for unsupported flags without performing the bind operation.

Supports Linux abstract namespace sockets. namelen must include the leading '\0' byte but not the trailing nul byte.

Returns: `0` or `fail`

Note:

1. Paths on Unix get truncated to sizeof(sockaddr_un.sun_path) bytes,
   typically between 92 and 108 bytes.

2. New in version 1.46.0.

`uv.pipe_connect2(pipe, name, [flags], [callback])`

> method form `pipe:connect2(name, [flags], [callback])`

Parameters:

- `pipe`: `uv_pipe_t userdata`
- `name`: `string`
- `flags`: `integer` or `table` or `nil` (default: 0)
- `callback`: `callable` or `nil`
    - `err`: `nil` or `string`

Connect to the Unix domain socket or the named pipe.

`Flags`:

- If `type(flags)` is `number`, it must be `0` or `uv.constants.PIPE_NO_TRUNCATE`.
- If `type(flags)` is `table`, it must be `{}` or `{ no_truncate = true|false }`.
- If `type(flags)` is `nil`, it use default value `0`.
- Returns `EINVAL` for unsupported flags without performing the bind operation.

Supports Linux abstract namespace sockets. namelen must include the leading nul byte but not the trailing nul byte.

Returns: `uv_connect_t userdata` or `fail`

Note:

1. Paths on Unix get truncated to sizeof(sockaddr_un.sun_path) bytes, typically between 92 and 108 bytes.
2. New in version 1.46.0.

`uv_tty_t` — TTY handle

> `uv_handle_t` and `uv_stream_t` functions also apply.

TTY handles represent a stream for the console.

```lua
-- Simple echo program
local stdin = uv.new_tty(0, true)
local stdout = uv.new_tty(1, false)

stdin:read_start(function (err, data)
  assert(not err, err)
  if data then
    stdout:write(data)
  else
    stdin:close()
    stdout:close()
  end
end)
```

`uv.new_tty(fd, readable)`

Parameters:

- `fd`: `integer`
- `readable`: `boolean`

Initialize a new TTY stream with the given file descriptor. Usually the file descriptor will be:

- 0 - stdin
- 1 - stdout
- 2 - stderr

On Unix this function will determine the path of the fd of the terminal using ttyname_r(3), open it, and use it if the passed file descriptor refers to a TTY. This lets libuv put the tty in non-blocking mode without affecting other processes that share the tty.

This function is not thread safe on systems that don't support ioctl TIOCGPTN or TIOCPTYGNAME, for instance OpenBSD and Solaris.

Returns: `uv_tty_t userdata` or `fail`

Note: If reopening the TTY fails, libuv falls back to blocking writes.

`uv.tty_set_mode(tty, mode)`

> method form `tty:set_mode(mode)`

Parameters:

- `tty`: `uv_tty_t userdata`
- `mode`: `string` or `integer`

Set the TTY using the specified terminal mode.

See Constants for supported TTY mode input values.

Returns: `0` or `fail`

`uv.tty_reset_mode()`

To be called when the program exits. Resets TTY settings to default values for the next process to take over.

This function is async signal-safe on Unix platforms but can fail with error code `EBUSY` if you call it when execution is inside `uv.tty_set_mode()`.

Returns: `0` or `fail`

`uv.tty_get_winsize(tty)`

> method form `tty:get_winsize()`

Parameters:

- `tty`: `uv_tty_t userdata`

Gets the current Window width and height.

Returns: `integer, integer` or `fail`

`uv.tty_set_vterm_state(state)`

Parameters:

- `state`: `string`

Controls whether console virtual terminal sequences are processed by libuv or console. Useful in particular for enabling ConEmu support of ANSI X3.64 and Xterm 256 colors. Otherwise Windows10 consoles are usually detected automatically. State should be one of: `"supported"` or `"unsupported"`.

This function is only meaningful on Windows systems. On Unix it is silently ignored.

Returns: none

`uv.tty_get_vterm_state()`

Get the current state of whether console virtual terminal sequences are handled by libuv or the console. The return value is `"supported"` or `"unsupported"`.

This function is not implemented on Unix, where it returns `ENOTSUP`.

Returns: `string` or `fail`

`uv_udp_t` — UDP handle

> `uv_handle_t` functions also apply.

UDP handles encapsulate UDP communication for both clients and servers.

`uv.new_udp([flags])`

Parameters:

- `flags`: `table` or `nil`
  - `family`: `string` or `nil`
  - `mmsgs`: `integer` or `nil` (default: `1`)

Creates and initializes a new `uv_udp_t`. Returns the Lua userdata wrapping it. The actual socket is created lazily.

See Constants for supported address `family` input values.

When specified, `mmsgs` determines the number of messages able to be received at one time via `recvmmsg(2)` (the allocated buffer will be sized to be able to fit the specified number of max size dgrams). Only has an effect on platforms that support `recvmmsg(2)`.

Note: For backwards compatibility reasons, `flags` can also be a string or integer. When it is a string, it will be treated like the `family` key above. When it is an integer, it will be used directly as the `flags` parameter when calling `uv_udp_init_ex`.

Returns: `uv_udp_t userdata` or `fail`

`uv.udp_get_send_queue_size()`

> method form `udp:get_send_queue_size()`

Returns the handle's send queue size.

Returns: `integer`

`uv.udp_get_send_queue_count()`

> method form `udp:get_send_queue_count()`

Returns the handle's send queue count.

Returns: `integer`

`uv.udp_open(udp, fd)`

> method form `udp:open(fd)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `fd`: `integer`

Opens an existing file descriptor or Windows SOCKET as a UDP handle.

Unix only: The only requirement of the sock argument is that it follows the datagram contract (works in unconnected mode, supports sendmsg()/recvmsg(), etc). In other words, other datagram-type sockets like raw sockets or netlink sockets can also be passed to this function.

The file descriptor is set to non-blocking mode.

Note: The passed file descriptor or SOCKET is not checked for its type, but it's required that it represents a valid datagram socket.

Returns: `0` or `fail`

`uv.udp_bind(udp, host, port, [flags])`

> method form `udp:bind(host, port, [flags])`

Parameters:

- `udp`: `uv_udp_t userdata`
- `host`: `string`
- `port`: `number`
- `flags`: `table` or `nil`
  - `ipv6only`: `boolean`
  - `reuseaddr`: `boolean`

Bind the UDP handle to an IP address and port. Any `flags` are set with a table with fields `reuseaddr` or `ipv6only` equal to `true` or `false`.

Returns: `0` or `fail`

`uv.udp_getsockname(udp)`

> method form `udp:getsockname()`

Parameters:

- `udp`: `uv_udp_t userdata`

Get the local IP and port of the UDP handle.

Returns: `table` or `fail`

- `ip` : `string`
- `family` : `string`
- `port` : `integer`

`uv.udp_getpeername(udp)`

> method form `udp:getpeername()`

Parameters:

- `udp`: `uv_udp_t userdata`

Get the remote IP and port of the UDP handle on connected UDP handles.

Returns: `table` or `fail`

- `ip` : `string`
- `family` : `string`

- `port` : `integer`

`uv.udp_set_membership(udp, multicast_addr, interface_addr, membership)`

> method form `udp:set_membership(multicast_addr, interface_addr, membership)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `multicast_addr`: `string`
- `interface_addr`: `string` or `nil`
- `membership`: `string`

Set membership for a multicast address. `multicast_addr` is multicast address to set membership for. `interface_addr` is interface address. `membership` can be the string `"leave"` or `"join"`.

Returns: `0` or `fail`

`uv.udp_set_source_membership(udp, multicast_addr, interface_addr, source_addr, membership)`

> method form `udp:set_source_membership(multicast_addr, interface_addr, source_addr, membership)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `multicast_addr`: `string`
- `interface_addr`: `string` or `nil`
- `source_addr`: `string`
- `membership`: `string`

Set membership for a source-specific multicast group. `multicast_addr` is multicast address to set membership for. `interface_addr` is interface address. `source_addr` is source address. `membership` can be the string `"leave"` or `"join"`.

Returns: `0` or `fail`

`uv.udp_set_multicast_loop(udp, on)`

> method form `udp:set_multicast_loop(on)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `on`: `boolean`

Set IP multicast loop flag. Makes multicast packets loop back to local sockets.

Returns: `0` or `fail`

`uv.udp_set_multicast_ttl(udp, ttl)`

> method form `udp:set_multicast_ttl(ttl)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `ttl`: `integer`

Set the multicast ttl.

`ttl` is an integer 1 through 255.

Returns: `0` or `fail`

`uv.udp_set_multicast_interface(udp, interface_addr)`

> method form `udp:set_multicast_interface(interface_addr)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `interface_addr`: `string`

Set the multicast interface to send or receive data on.

Returns: `0` or `fail`

`uv.udp_set_broadcast(udp, on)`

> method form `udp:set_broadcast(on)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `on`: `boolean`

Set broadcast on or off.

Returns: `0` or `fail`

`uv.udp_set_ttl(udp, ttl)`

> method form `udp:set_ttl(ttl)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `ttl`: `integer`

Set the time to live.

`ttl` is an integer 1 through 255.

Returns: `0` or `fail`

`uv.udp_send(udp, data, host, port, callback)`

> method form `udp:send(data, host, port, callback)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `data`: `buffer`
- `host`: `string`
- `port`: `integer`
- `callback`: `callable`
  - `err`: `nil` or `string`

Send data over the UDP socket. If the socket has not previously been bound with `uv.udp_bind()` it will be bound to `0.0.0.0` (the "all interfaces" IPv4 address) and a random port number.

Returns: `uv_udp_send_t userdata` or `fail`

`uv.udp_try_send(udp, data, host, port)`

> method form `udp:try_send(data, host, port)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `data`: `buffer`
- `host`: `string`
- `port`: `integer`

Same as `uv.udp_send()`, but won't queue a send request if it can't be completed immediately.

Returns: `integer` or `fail`

`uv.udp_recv_start(udp, callback)`

> method form `udp:recv_start(callback)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `callback`: `callable`
  - `err`: `nil` or `string`
  - `data`: `string` or `nil`
  - `addr`: `table` or `nil`
    - `ip`: `string`
    - `port`: `integer`
    - `family`: `string`

- `flags`: `table`
  - `partial`: `boolean` or `nil`
  - `mmsg_chunk`: `boolean` or `nil`

Prepare for receiving data. If the socket has not previously been bound with `uv.udp_bind()` it is bound to `0.0.0.0` (the "all interfaces" IPv4 address) and a random port number.

See Constants for supported address `family` output values.

Returns: `0` or `fail`

`uv.udp_recv_stop(udp)`

> method form `udp:recv_stop()`

Parameters:

- `udp`: `uv_udp_t userdata`

Stop listening for incoming datagrams.

Returns: `0` or `fail`

`uv.udp_connect(udp, host, port)`

> method form `udp:connect(host, port)`

Parameters:

- `udp`: `uv_udp_t userdata`
- `host`: `string`
- `port`: `integer`

Associate the UDP handle to a remote address and port, so every message sent by this handle is automatically sent to that destination. Calling this function with a NULL addr disconnects the handle. Trying to call `uv.udp_connect()` on an already connected handle will result in an `EISCONN` error. Trying to disconnect a handle that is not connected will return an `ENOTCONN` error.

Returns: `0` or `fail`

`uv_fs_event_t` — FS Event handle

> `uv_handle_t` functions also apply.

FS Event handles allow the user to monitor a given path for changes, for example, if the file was renamed or there was a generic change in it. This handle uses the best backend for the job on each platform.

`uv.new_fs_event()`

Creates and initializes a new `uv_fs_event_t`. Returns the Lua userdata wrapping it.

Returns: `uv_fs_event_t userdata` or `fail`

`uv.fs_event_start(fs_event, path, flags, callback)`

> method form `fs_event:start(path, flags, callback)`

Parameters:

- `fs_event`: `uv_fs_event_t userdata`
- `path`: `string`
- `flags`: `table`
  - `watch_entry`: `boolean` or `nil` (default: `false`)
  - `stat`: `boolean` or `nil` (default: `false`)
  - `recursive`: `boolean` or `nil` (default: `false`)
- `callback`: `callable`
  - `err`: `nil` or `string`
  - `filename`: `string`
  - `events`: `table`
    - `change`: `boolean` or `nil`
    - `rename`: `boolean` or `nil`

Start the handle with the given callback, which will watch the specified path for changes.

Returns: `0` or `fail`

`uv.fs_event_stop()`

> method form `fs_event:stop()`

Stop the handle, the callback will no longer be called.

Returns: `0` or `fail`

`uv.fs_event_getpath()`

> method form `fs_event:getpath()`

Get the path being monitored by the handle.

Returns: `string` or `fail`

`uv_fs_poll_t` — FS Poll handle

> `uv_handle_t` functions also apply.

FS Poll handles allow the user to monitor a given path for changes. Unlike `uv_fs_event_t`, fs poll handles use `stat` to detect when a file has changed so they can work on file systems where fs event handles can't.

`uv.new_fs_poll()`

Creates and initializes a new `uv_fs_poll_t`. Returns the Lua userdata wrapping it.

Returns: `uv_fs_poll_t userdata` or `fail`

`uv.fs_poll_start(fs_poll, path, interval, callback)`

> method form `fs_poll:start(path, interval, callback)`

Parameters:

- `fs_poll`: `uv_fs_poll_t userdata`
- `path`: `string`
- `interval`: `integer`
- `callback`: `callable`
  - `err`: `nil` or `string`
  - `prev`: `table` or `nil` (see `uv.fs_stat`)
  - `curr`: `table` or `nil` (see `uv.fs_stat`)

Check the file at `path` for changes every `interval` milliseconds.

Note: For maximum portability, use multi-second intervals. Sub-second intervals will not detect all changes on many file systems.

Returns: `0` or `fail`

`uv.fs_poll_stop()`

> method form `fs_poll:stop()`

Stop the handle, the callback will no longer be called.

Returns: `0` or `fail`

`uv.fs_poll_getpath()`

> method form `fs_poll:getpath()`

Get the path being monitored by the handle.

Returns: `string` or `fail`

File system operations

Most file system functions can operate synchronously or asynchronously. When a synchronous version is called (by omitting a callback), the function will immediately return the results of the FS call. When an asynchronous version is called (by providing a callback), the function will immediately return a `uv_fs_t userdata` and asynchronously execute its callback; if an error is encountered, the first and only argument passed to the callback will be the `err` error string; if the operation completes successfully, the first argument will be `nil` and the remaining arguments will be the results of the FS call.

Synchronous and asynchronous versions of `readFile` (with naive error handling) are implemented below as an example:

```lua
local function readFileSync(path)
  local fd = assert(uv.fs_open(path, "r", 438))
  local stat = assert(uv.fs_fstat(fd))
  local data = assert(uv.fs_read(fd, stat.size, 0))
  assert(uv.fs_close(fd))
  return data
end

local data = readFileSync("main.lua")
print("synchronous read", data)
```

```lua
local function readFile(path, callback)
  uv.fs_open(path, "r", 438, function(err, fd)
    assert(not err, err)
    uv.fs_fstat(fd, function(err, stat)
      assert(not err, err)
      uv.fs_read(fd, stat.size, 0, function(err, data)
        assert(not err, err)
        uv.fs_close(fd, function(err)
          assert(not err, err)
          return callback(data)
        end)
      end)
    end)
  end)
end

readFile("main.lua", function(data)
  print("asynchronous read", data)
end)
```

`uv.fs_close(fd, [callback])`

Parameters:

- `fd` : `integer`
- `callback` : `callable` (async version) or `nil` (sync version)
    - `err` : `nil` or `string`

- `success`: `boolean` or `nil`

Equivalent to `close(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_open(path, flags, mode, [callback])`

Parameters:

- `path`: `string`
- `flags`: `string` or `integer`
- `mode`: `integer` (octal `chmod(1)` mode, e.g. `tonumber('644', 8)`)
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `fd`: `integer` or `nil`

Equivalent to `open(2)`. Access `flags` may be an integer or one of: `"r"`, `"rs"`, `"sr"`, `"r+"`, `"rs+"`, `"sr+"`, `"w"`, `"wx"`, `"xw"`, `"w+"`, `"wx+"`, `"xw+"`, `"a"`, `"ax"`, `"xa"`, `"a+"`, `"ax+"`, or "`xa+`".

Returns (sync version): `integer` or `fail`

Returns (async version): `uv_fs_t userdata`

Note: On Windows, libuv uses `CreateFileW` and thus the file is always opened in binary mode. Because of this, the `O_BINARY` and `O_TEXT` flags are not supported.

`uv.fs_read(fd, size, [offset], [callback])`

Parameters:

- `fd`: `integer`
- `size`: `integer`
- `offset`: `integer` or `nil`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `data`: `string` or `nil`

Equivalent to `preadv(2)`. Returns any data. An empty string indicates EOF.

If `offset` is nil or omitted, it will default to `-1`, which indicates 'use and update the current file offset.'

Note: When `offset` is >= 0, the current file offset will not be updated by the read.

Returns (sync version): `string` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_unlink(path, [callback])`

Parameters:

- `path`: `string`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `unlink(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_write(fd, data, [offset], [callback])`

Parameters:

- `fd`: `integer`
- `data`: `buffer`
- `offset`: `integer` or `nil`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `bytes`: `integer` or `nil`

Equivalent to `pwritev(2)`. Returns the number of bytes written.

If `offset` is nil or omitted, it will default to `-1`, which indicates 'use and update the current file offset.'

Note: When `offset` is >= 0, the current file offset will not be updated by the write.

Returns (sync version): `integer` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_mkdir(path, mode, [callback])`

Parameters:

- `path`: `string`
- `mode`: `integer` (octal `chmod(1)` mode, e.g. `tonumber('755', 8)`)
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `mkdir(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_mkdtemp(template, [callback])`

Parameters:

- `template`: `string`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `path`: `string` or `nil`

Equivalent to `mkdtemp(3)`.

Returns (sync version): `string` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_mkstemp(template, [callback])`

Parameters:

- `template`: `string`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `fd`: `integer` or `nil`
  - `path`: `string` or `nil`

Equivalent to `mkstemp(3)`. Returns a temporary file handle and filename.

Returns (sync version): `integer, string` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_rmdir(path, [callback])`

Parameters:

- `path`: `string`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `rmdir(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_scandir(path, [callback])`

Parameters:

- `path`: `string`
- `callback`: `callable`
  - `err`: `nil` or `string`

    - `success`: `uv_fs_t userdata` or `nil`

Equivalent to `scandir(3)`, with a slightly different API. Returns a handle that the user can pass to `uv.fs_scandir_next()`.

Note: This function can be used synchronously or asynchronously. The request userdata is always synchronously returned regardless of whether a callback is provided and the same userdata is passed to the callback if it is provided.

Returns: `uv_fs_t userdata` or `fail`

`uv.fs_scandir_next(fs)`

Parameters:

- `fs`: `uv_fs_t userdata`

Called on a `uv_fs_t` returned by `uv.fs_scandir()` to get the next directory entry data as a `name, type` pair. When there are no more entries, `nil` is returned.

Note: This function only has a synchronous version. See `uv.fs_opendir` and its related functions for an asynchronous version.

Returns: `string, string` or `nil` or `fail`

`uv.fs_stat(path, [callback])`

Parameters:

- `path`: `string`
- `callback`: `callable` (async version) or `nil` (sync version)
    - `err`: `nil` or `string`
    - `stat`: `table` or `nil` (see below)

Equivalent to `stat(2)`.

Returns (sync version): `table` or `fail`

- `dev` : `integer`
- `mode` : `integer`
- `nlink` : `integer`
- `uid` : `integer`
- `gid` : `integer`
- `rdev` : `integer`
- `ino` : `integer`
- `size` : `integer`
- `blksize` : `integer`
- `blocks` : `integer`
- `flags` : `integer`
- `gen` : `integer`

- `atime` : `table`
  - `sec` : `integer`
  - `nsec` : `integer`
- `mtime` : `table`
  - `sec` : `integer`
  - `nsec` : `integer`
- `ctime` : `table`
  - `sec` : `integer`
  - `nsec` : `integer`
- `birthtime` : `table`
  - `sec` : `integer`
  - `nsec` : `integer`
- `type` : `string`

Returns (async version): `uv_fs_t userdata`

`uv.fs_fstat(fd, [callback])`

Parameters:

- `fd` : `integer`
- `callback` : `callable` (async version) or `nil` (sync version)
  - `err` : `nil` or `string`
  - `stat` : `table` or `nil` (see `uv.fs_stat`)

Equivalent to `fstat(2)`.

Returns (sync version): `table` or `fail` (see `uv.fs_stat`)

Returns (async version): `uv_fs_t userdata`

`uv.fs_lstat(path, [callback])`

Parameters:

- `path` : `string`
- `callback` : `callable` (async version) or `nil` (sync version)
  - `err` : `nil` or `string`
  - `stat` : `table` or `nil` (see `uv.fs_stat`)

Equivalent to `lstat(2)`.

Returns (sync version): `table` or `fail` (see `uv.fs_stat`)

Returns (async version): `uv_fs_t userdata`

`uv.fs_rename(path, new_path, [callback])`

Parameters:

- `path`: `string`
- `new_path`: `string`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `rename(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_fsync(fd, [callback])`

Parameters:

- `fd`: `integer`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `fsync(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_fdatasync(fd, [callback])`

Parameters:

- `fd`: `integer`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `fdatasync(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_ftruncate(fd, offset, [callback])`

Parameters:

- `fd`: `integer`
- `offset`: `integer`
- `callback`: `callable` (async version) or `nil` (sync version)

- `err`: `nil` or `string`
- `success`: `boolean` or `nil`

Equivalent to `ftruncate(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_sendfile(out_fd, in_fd, in_offset, size, [callback])`

Parameters:

- `out_fd`: `integer`
- `in_fd`: `integer`
- `in_offset`: `integer`
- `size`: `integer`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `bytes`: `integer` or `nil`

Limited equivalent to `sendfile(2)`. Returns the number of bytes written.

Returns (sync version): `integer` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_access(path, mode, [callback])`

Parameters:

- `path`: `string`
- `mode`: `string` (a combination of the `'r'`, `'w'` and `'x'` characters denoting the symbolic mode as per `chmod(1)`)
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `permission`: `boolean` or `nil`

Equivalent to `access(2)` on Unix. Windows uses `GetFileAttributesW()`. Access `mode` can be an integer or a string containing `"R"` or `"W"` or `"X"`.
Returns `true` or `false` indicating access permission.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_chmod(path, mode, [callback])`

Parameters:

- `path`: `string`
- `mode`: `integer` (octal `chmod(1)` mode, e.g. `tonumber('644', 8)`)
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `chmod(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_fchmod(fd, mode, [callback])`

Parameters:

- `fd`: `integer`
- `mode`: `integer`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `fchmod(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_utime(path, atime, mtime, [callback])`

Parameters:

- `path`: `string`
- `atime`: `number`
- `mtime`: `number`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `utime(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_futime(fd, atime, mtime, [callback])`

Parameters:

- `fd`: `integer`
- `atime`: `number`
- `mtime`: `number`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `futime(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_lutime(path, atime, mtime, [callback])`

Parameters:

- `path`: `string`
- `atime`: `number`
- `mtime`: `number`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `lutime(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_link(path, new_path, [callback])`

Parameters:

- `path`: `string`
- `new_path`: `string`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `link(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_symlink(path, new_path, [flags], [callback])`

Parameters:

- `path`: `string`
- `new_path`: `string`
- `flags`: `table`, `integer`, or `nil`
  - `dir`: `boolean`
  - `junction`: `boolean`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `symlink(2)`. If the `flags` parameter is omitted, then the 3rd parameter will be treated as the `callback`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_readlink(path, [callback])`

Parameters:

- `path`: `string`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `path`: `string` or `nil`

Equivalent to `readlink(2)`.

Returns (sync version): `string` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_realpath(path, [callback])`

Parameters:

- `path`: `string`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `path`: `string` or `nil`

Equivalent to `realpath(3)`.

Returns (sync version): `string` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_chown(path, uid, gid, [callback])`

Parameters:

- `path`: `string`
- `uid`: `integer`
- `gid`: `integer`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `chown(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_fchown(fd, uid, gid, [callback])`

Parameters:

- `fd`: `integer`
- `uid`: `integer`
- `gid`: `integer`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `fchown(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_lchown(fd, uid, gid, [callback])`

Parameters:

- `fd`: `integer`
- `uid`: `integer`
- `gid`: `integer`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Equivalent to `lchown(2)`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_copyfile(path, new_path, [flags], [callback])`

Parameters:

- `path`: `string`
- `new_path`: `string`
- `flags`: `table`, `integer`, or `nil`
  - `excl`: `boolean`
  - `ficlone`: `boolean`
  - `ficlone_force`: `boolean`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `success`: `boolean` or `nil`

Copies a file from path to new_path. If the `flags` parameter is omitted, then the 3rd parameter will be treated as the `callback`.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_opendir(path, [callback, [entries]])`

Parameters:

- `path`: `string`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`
  - `dir`: `luv_dir_t userdata` or `nil`
- `entries`: `integer` or `nil`

Opens path as a directory stream. Returns a handle that the user can pass to `uv.fs_readdir()`. The `entries` parameter defines the maximum number of entries that should be returned by each call to `uv.fs_readdir()`.

Returns (sync version): `luv_dir_t userdata` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_readdir(dir, [callback])`

> method form `dir:readdir([callback])`

Parameters:

- `dir`: `luv_dir_t userdata`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `string`

- `entries` : `table` or `nil` (see below)

Iterates over the directory stream `luv_dir_t` returned by a successful `uv.fs_opendir()` call. A table of data tables is returned where the number of entries `n` is equal to or less than the `entries` parameter used in the associated `uv.fs_opendir()` call.

Returns (sync version): `table` or `fail`

- `[1, 2, 3, ..., n]` : `table`
  - `name` : `string`
  - `type` : `string`

Returns (async version): `uv_fs_t userdata`

`uv.fs_closedir(dir, [callback])`

> method form `dir:closedir([callback])`

Parameters:

- `dir` : `luv_dir_t userdata`
- `callback` : `callable` (async version) or `nil` (sync version)
  - `err` : `nil` or `string`
  - `success` : `boolean` or `nil`

Closes a directory stream returned by a successful `uv.fs_opendir()` call.

Returns (sync version): `boolean` or `fail`

Returns (async version): `uv_fs_t userdata`

`uv.fs_statfs(path, [callback])`

Parameters:

- `path` : `string`
- `callback` : `callable` (async version) or `nil` (sync version)
  - `err` : `nil` or `string`
  - `table` or `nil` (see below)

Equivalent to `statfs(2)`.

Returns `table` or `nil`

- `type` : `integer`
- `bsize` : `integer`
- `blocks` : `integer`
- `bfree` : `integer`
- `bavail` : `integer`
- `files` : `integer`

- `ffree` : `integer`

Thread pool work scheduling

Libuv provides a threadpool which can be used to run user code and get notified in the loop thread. This threadpool is internally used to run all file system operations, as well as `getaddrinfo` and `getnameinfo` requests.

```lua
local function work_callback(a, b)
  return a + b
end

local function after_work_callback(c)
  print("The result is: " .. c)
end

local work = uv.new_work(work_callback, after_work_callback)

work:queue(1, 2)

-- output: "The result is: 3"
```

`uv.new_work(work_callback, after_work_callback)`

Parameters:

- `work_callback` : `function` or `string`
  - `...` : `threadargs` passed to/from `uv.queue_work(work_ctx, ...)`
- `after_work_callback` : `function`
  - `...` : `threadargs` returned from `work_callback`

Creates and initializes a new `luv_work_ctx_t` (not `uv_work_t`).
`work_callback` is a Lua function or a string containing Lua code or bytecode dumped from a function.
Returns the Lua userdata wrapping it.

Returns: `luv_work_ctx_t userdata`

`uv.queue_work(work_ctx, ...)`

> method form `work_ctx:queue(...)`

Parameters:

- `work_ctx` : `luv_work_ctx_t userdata`
- `...` : `threadargs`

Queues a work request which will run `work_callback` in a new Lua state in a thread from the threadpool with any additional arguments from `...`. Values returned from `work_callback` are passed to `after_work_callback`, which is called in the main loop thread.

Returns: `boolean` or `fail`

DNS utility functions

`uv.getaddrinfo(host, service, [hints, [callback]])`

Parameters:

- `host` : `string` or `nil`
- `service` : `string` or `nil`
- `hints` : `table` or `nil`
  - `family` : `string` or `integer` or `nil`
  - `socktype` : `string` or `integer` or `nil`
  - `protocol` : `string` or `integer` or `nil`
  - `addrconfig` : `boolean` or `nil`
  - `v4mapped` : `boolean` or `nil`
  - `all` : `boolean` or `nil`
  - `numerichost` : `boolean` or `nil`
  - `passive` : `boolean` or `nil`
  - `numericserv` : `boolean` or `nil`
  - `canonname` : `boolean` or `nil`
- `callback` : `callable` (async version) or `nil` (sync version)
  - `err` : `nil` or `string`
  - `addresses` : `table` or `nil` (see below)

Equivalent to `getaddrinfo(3)`. Either `node` or `service` may be `nil` but not both.

See Constants for supported address `family` input and output values.

See Constants for supported `socktype` input and output values.

When `protocol` is set to 0 or nil, it will be automatically chosen based on the socket's domain and type. When `protocol` is specified as a string, it will be looked up using the `getprotobyname(3)` function. Examples: `"ip"`, `"icmp"`, `"tcp"`, `"udp"`, etc.

Returns (sync version): `table` or `fail`

- `[1, 2, 3, ..., n]` : `table`
  - `addr` : `string`
  - `family` : `string`
  - `port` : `integer` or `nil`
  - `socktype` : `string`
  - `protocol` : `string`
  - `canonname` : `string` or `nil`

Returns (async version): `uv_getaddrinfo_t userdata` or `fail`

`uv.getnameinfo(address, [callback])`

Parameters:

- `address`: `table`
  - `ip`: `string` or `nil`
  - `port`: `integer` or `nil`
  - `family`: `string` or `integer` or `nil`
- `callback`: `callable` (async version) or `nil` (sync version)
  - `err`: `nil` or `sring`
  - `host`: `string` or `nil`
  - `service`: `string` or `nil`

Equivalent to `getnameinfo(3)`.

See Constants for supported address `family` input values.

Returns (sync version): `string, string` or `fail`

Returns (async version): `uv_getnameinfo_t userdata` or `fail`

Threading and synchronization utilities

Libuv provides cross-platform implementations for multiple threading an
synchronization primitives. The API largely follows the pthreads API.

`uv.new_thread([options], entry, ...)`

Parameters:

- `options`: `table` or `nil`
  - `stack_size`: `integer` or `nil`
- `entry`: `function` or `string`
- `...`: `threadargs` passed to `entry`

Creates and initializes a `luv_thread_t` (not `uv_thread_t`). Returns the Lua
userdata wrapping it and asynchronously executes `entry`, which can be either
a Lua function or a string containing Lua code or bytecode dumped from a function. Additional
arguments `...`
are passed to the `entry` function and an optional `options` table may be
provided. Currently accepted `option` fields are `stack_size`.

Returns: `luv_thread_t userdata` or `fail`

Note: unsafe, please make sure the thread end of life before Lua state close.

`uv.thread_equal(thread, other_thread)`

> method form `thread:equal(other_thread)`

Parameters:

- `thread`: `luv_thread_t userdata`
- `other_thread`: `luv_thread_t userdata`

Returns a boolean indicating whether two threads are the same. This function is equivalent to the `__eq` metamethod.

Returns: `boolean`

`uv.thread_setaffinity(thread, affinity, [get_old_affinity])`

> method form `thread:setaffinity(affinity, [get_old_affinity])`

Parameters:

- `thread`: `luv_thread_t userdata`
- `affinity`: `table`
  - `[1, 2, 3, ..., n]`: `boolean`
- `get_old_affinity`: `boolean`

Sets the specified thread's affinity setting.

`affinity` must be a table where each of the keys are a CPU number and the values are booleans that represent whether the `thread` should be eligible to run on that CPU. If the length of the `affinity` table is not greater than or equal to `uv.cpumask_size()`, any CPU numbers missing from the table will have their affinity set to `false`. If setting the affinity of more than `uv.cpumask_size()` CPUs is desired, `affinity` must be an array-like table with no gaps, since `#affinity` will be used as the `cpumask_size` if it is greater than `uv.cpumask_size()`.

If `get_old_affinity` is `true`, the previous affinity settings for the `thread` will be returned. Otherwise, `true` is returned after a successful call.

Note: Thread affinity setting is not atomic on Windows. Unsupported on macOS.

Returns: `table` or `boolean` or `fail`

- `[1, 2, 3, ..., n]`: `boolean`

`uv.thread_getaffinity(thread, [mask_size])`

> method form `thread:getaffinity([mask_size])`

Parameters:

- `thread`: `luv_thread_t userdata`
- `mask_size`: `integer`

Gets the specified thread's affinity setting.

If `mask_size` is provided, it must be greater than or equal to `uv.cpumask_size()`. If the `mask_size` parameter is omitted, then the return of `uv.cpumask_size()` will be used. Returns an array-like table where each of the keys correspond to a CPU number and the values are booleans that represent whether the `thread` is eligible to run on that CPU.

Note: Thread affinity getting is not atomic on Windows. Unsupported on macOS.

Returns: `table` or `fail`

- `[1, 2, 3, ..., n]` : `boolean`

`uv.thread_getcpu()`

Gets the CPU number on which the calling thread is running.

Note: The first CPU will be returned as the number 1, not 0. This allows for the number to correspond with the table keys used in `uv.thread_getaffinity` and `uv.thread_setaffinity`.

Returns: `integer` or `fail`

`uv.thread_setpriority(thread, priority)`

> method form `thread:setpriority(priority)`

Parameters:

- `thread`: `luv_thread_t userdata`
- `priority`: `number`

Sets the specified thread's scheduling priority setting. It requires elevated privilege to set specific priorities on some platforms.

The priority can be set to the following constants.

- uv.constants.THREAD_PRIORITY_HIGHEST
- uv.constants.THREAD_PRIORITY_ABOVE_NORMAL
- uv.constants.THREAD_PRIORITY_NORMAL
- uv.constants.THREAD_PRIORITY_BELOW_NORMAL
- uv.constants.THREAD_PRIORITY_LOWEST

Returns: `boolean` or `fail`

`uv.thread_getpriority(thread)

> method form `thread:getpriority()`

Parameters:

- `thread`: `luv_thread_t userdata`

Gets the thread's priority setting.

Retrieves the scheduling priority of the specified thread. The returned priority value is platform dependent.

For Linux, when schedule policy is SCHED_OTHER (default), priority is 0.

Returns: `number` or `fail`

`uv.thread_self()`

Returns the handle for the thread in which this is called.

Returns: `luv_thread_t`

`uv.thread_join(thread)`

> method form `thread:join()`

Parameters:

- `thread`: `luv_thread_t userdata`

Waits for the `thread` to finish executing its entry function.

Returns: `boolean` or `fail`

`uv.sleep(msec)`

Parameters:

- `msec`: `integer`

Pauses the thread in which this is called for a number of milliseconds.

Returns: Nothing.

Miscellaneous utilities

`uv.exepath()`

Returns the executable path.

Returns: `string` or `fail`

`uv.cwd()`

Returns the current working directory.

Returns: `string` or `fail`

`uv.chdir(cwd)`

Parameters:

- `cwd`: `string`

Sets the current working directory with the string `cwd`.

Returns: `0` or `fail`

`uv.get_process_title()`

Returns the title of the current process.

Returns: `string` or `fail`

`uv.set_process_title(title)`

Parameters:

- `title`: `string`

Sets the title of the current process with the string `title`.

Returns: `0` or `fail`

`uv.get_total_memory()`

Returns the current total system memory in bytes.

Returns: `number`

`uv.get_free_memory()`

Returns the current free system memory in bytes.

Returns: `number`

`uv.get_constrained_memory()`

Gets the amount of memory available to the process in bytes based on limits imposed by the OS. If there is no such constraint, or the constraint is unknown, 0 is returned. Note that it is not unusual for this value to be less than or greater than the total system memory.

Returns: `number`

`uv.get_available_memory()`

Gets the amount of free memory that is still available to the process (in bytes). This differs from `uv.get_free_memory()` in that it takes into account any limits imposed by the OS. If there is no such constraint, or the constraint is unknown, the amount returned will be identical to `uv.get_free_memory()`.

Returns: `number`

`uv.resident_set_memory()`

Returns the resident set size (RSS) for the current process.

Returns: `integer` or `fail`

`uv.getrusage()`

Returns the resource usage.

Returns: `table` or `fail`

- `utime` : `table` (user CPU time used)
  - `sec` : `integer`
  - `usec` : `integer`
- `stime` : `table` (system CPU time used)
  - `sec` : `integer`
  - `usec` : `integer`
- `maxrss` : `integer` (maximum resident set size)
- `ixrss` : `integer` (integral shared memory size)
- `idrss` : `integer` (integral unshared data size)
- `isrss` : `integer` (integral unshared stack size)
- `minflt` : `integer` (page reclaims (soft page faults))
- `majflt` : `integer` (page faults (hard page faults))
- `nswap` : `integer` (swaps)
- `inblock` : `integer` (block input operations)
- `oublock` : `integer` (block output operations)
- `msgsnd` : `integer` (IPC messages sent)
- `msgrcv` : `integer` (IPC messages received)
- `nsignals` : `integer` (signals received)
- `nvcsw` : `integer` (voluntary context switches)
- `nivcsw` : `integer` (involuntary context switches)

`uv.available_parallelism()`

Returns an estimate of the default amount of parallelism a program should use. Always returns a non-zero value.

On Linux, inspects the calling thread's CPU affinity mask to determine if it has been pinned to specific CPUs.

On Windows, the available parallelism may be underreported on systems with more than 64 logical CPUs.

On other platforms, reports the number of CPUs that the operating system considers to be online.

Returns: `integer`

`uv.cpu_info()`

Returns information about the CPU(s) on the system as a table of tables for each CPU found.

Returns: `table` or `fail`

- `[1, 2, 3, ..., n]` : `table`
  - `model` : `string`
  - `speed` : `number`
  - `times` : `table`
    - `user` : `number`
    - `nice` : `number`
    - `sys` : `number`
    - `idle` : `number`
    - `irq` : `number`

`uv.cpumask_size()`

Returns the maximum size of the mask used for process/thread affinities, or `ENOTSUP` if affinities are not supported on the current platform.

Returns: `integer` or `fail`

`uv.getpid()`

Deprecated: Please use `uv.os_getpid()` instead.

`uv.getuid()`

Returns the user ID of the process.

Returns: `integer`

Note: This is not a libuv function and is not supported on Windows.

`uv.getgid()`

Returns the group ID of the process.

Returns: `integer`

Note: This is not a libuv function and is not supported on Windows.

`uv.setuid(id)`

Parameters:

- `id` : `integer`

Sets the user ID of the process with the integer `id`.

Returns: Nothing.

Note: This is not a libuv function and is not supported on Windows.

`uv.setgid(id)`

Parameters:

- `id`: `integer`

Sets the group ID of the process with the integer `id`.

Returns: Nothing.

Note: This is not a libuv function and is not supported on Windows.

`uv.hrtime()`

Returns a current high-resolution time in nanoseconds as a number. This is relative to an arbitrary time in the past. It is not related to the time of day and therefore not subject to clock drift. The primary use is for measuring time between intervals.

Returns: `number`

`uv.clock_gettime(clock_id)`

Parameters:

- `clock_id`: `string`

Obtain the current system time from a high-resolution real-time or monotonic clock source. `clock_id` can be the string `"monotonic"` or `"realtime"`.

The real-time clock counts from the UNIX epoch (1970-01-01) and is subject to time adjustments; it can jump back in time.

The monotonic clock counts from an arbitrary point in the past and never jumps back in time.

Returns: `table` or `fail`

- `sec`: `integer`
- `nsec`: `integer`

`uv.uptime()`

Returns the current system uptime in seconds.

Returns: `number` or `fail`

`uv.print_all_handles()`

Prints all handles associated with the main loop to stderr. The format is `[flags] handle-type handle-address`. Flags are `R` for referenced, `A` for active and `I` for internal.

Returns: Nothing.

Note: This is not available on Windows.

Warning: This function is meant for ad hoc debugging, there are no API/ABI stability guarantees.

`uv.print_active_handles()`

The same as `uv.print_all_handles()` except only active handles are printed.

Returns: Nothing.

Note: This is not available on Windows.

Warning: This function is meant for ad hoc debugging, there are no API/ABI stability guarantees.

`uv.guess_handle(fd)`

Parameters:

- `fd` : `integer`

Used to detect what type of stream should be used with a given file descriptor `fd`. Usually this will be used during initialization to guess the type of the stdio streams.

Returns: `string`

`uv.gettimeofday()`

Cross-platform implementation of `gettimeofday(2)`. Returns the seconds and microseconds of a unix time as a pair.

Returns: `integer, integer` or `fail`

`uv.interface_addresses()`

Returns address information about the network interfaces on the system in a table. Each table key is the name of the interface while each associated value is an array of address information where fields are `ip`, `family`, `netmask`, `internal`, and `mac`.

See Constants for supported address `family` output values.

Returns: `table`

- `[name(s)]` : `table`
  - `ip` : `string`
  - `family` : `string`
  - `netmask` : `string`
  - `internal` : `boolean`
  - `mac` : `string`

`uv.if_indextoname(ifindex)`

Parameters:

- `ifindex`: `integer`

IPv6-capable implementation of `if_indextoname(3)`.

Returns: `string` or `fail`

`uv.if_indextoiid(ifindex)`

Parameters:

- `ifindex`: `integer`

Retrieves a network interface identifier suitable for use in an IPv6 scoped address. On Windows, returns the numeric `ifindex` as a string. On all other platforms, `uv.if_indextoname()` is used.

Returns: `string` or `fail`

`uv.loadavg()`

Returns the load average as a triad. Not supported on Windows.

Returns: `number, number, number`

`uv.os_uname()`

Returns system information.

Returns: `table`

- `sysname` : `string`
- `release` : `string`
- `version` : `string`
- `machine` : `string`

`uv.os_gethostname()`

Returns the hostname.

Returns: `string`

`uv.os_getenv(name, [size])`

Parameters:

- `name`: `string`
- `size`: `integer` (default = `LUAL_BUFFERSIZE`)

Returns the environment variable specified by `name` as string. The internal buffer size can be set by defining `size`. If omitted, `LUAL_BUFFERSIZE` is used. If the environment variable exceeds the storage available in the internal buffer, `ENOBUFS` is returned. If no matching environment variable exists, `ENOENT` is returned.

Returns: `string` or `fail`

Warning: This function is not thread safe.

`uv.os_setenv(name, value)`

Parameters:

- `name`: `string`
- `value`: `string`

Sets the environmental variable specified by `name` with the string `value`.

Returns: `boolean` or `fail`

Warning: This function is not thread safe.

`uv.os_unsetenv(name)`

Parameters:

- `name`: `string`

Unsets the environmental variable specified by `name`.

Returns: `boolean` or `fail`

Warning: This function is not thread safe.

`uv.os_environ()`

Returns all environmental variables as a dynamic table of names associated with their corresponding values.

Returns: `table`

Warning: This function is not thread safe.

`uv.os_homedir()`

Returns: `string` or `fail`

Warning: This function is not thread safe.

`uv.os_tmpdir()`

Returns: `string` or `fail`

Warning: This function is not thread safe.

`uv.os_get_passwd()`

Returns password file information.

Returns: `table`

- `username` : `string`
- `uid` : `integer`
- `gid` : `integer`
- `shell` : `string`
- `homedir` : `string`

`uv.os_getpid()`

Returns the current process ID.

Returns: `number`

`uv.os_getppid()`

Returns the parent process ID.

Returns: `number`

`uv.os_getpriority(pid)`

Parameters:

- `pid`: `integer`

Returns the scheduling priority of the process specified by `pid`.

Returns: `number` or `fail`

`uv.os_setpriority(pid, priority)`

Parameters:

- `pid`: `integer`
- `priority`: `integer`

Sets the scheduling priority of the process specified by `pid`. The `priority` range is between -20 (high priority) and 19 (low priority).

Returns: `boolean` or `fail`

`uv.random(len, flags, [callback])`

Parameters:

- `len`: `integer`
- `flags`: `nil` (see below)
- `callback`: `callable` (async version) or `nil` (sync version)
    - `err`: `nil` or `string`

- `bytes` : `string` or `nil`

Fills a string of length `len` with cryptographically strong random bytes acquired from the system CSPRNG. `flags` is reserved for future extension and must currently be `nil` or `0` or `{}`.

Short reads are not possible. When less than `len` random bytes are available, a non-zero error value is returned or passed to the callback. If the callback is omitted, this function is completed synchronously.

The synchronous version may block indefinitely when not enough entropy is available. The asynchronous version may not ever finish when the system is low on entropy.

Returns (sync version): `string` or `fail`

Returns (async version): `0` or `fail`

`uv.translate_sys_error(errcode)`

Parameters:

- `errcode` : `integer`

Returns the libuv error message and error name (both in string form, see `err` and `name` in Error Handling) equivalent to the given platform dependent error code: POSIX error codes on Unix (the ones stored in errno), and Win32 error codes on Windows (those returned by GetLastError() or WSAGetLastError()).

Returns: `string, string` or `nil`

Metrics operations

`uv.metrics_idle_time()`

Retrieve the amount of time the event loop has been idle in the kernel's event provider (e.g. `epoll_wait`). The call is thread safe.

The return value is the accumulated time spent idle in the kernel's event provider starting from when the `uv_loop_t` was configured to collect the idle time.

Note: The event loop will not begin accumulating the event provider's idle time until calling `loop_configure` with `"metrics_idle_time"`.

Returns: `number`

`uv.metrics_info()`

Get the metrics table from current set of event loop metrics. It is recommended to retrieve these metrics in a `prepare` callback (see `uv.new_prepare`, `uv.prepare_start`) in order to make sure there are no inconsistencies with the metrics counters.

Returns: `table`

- `loop_count` : `integer`

- `events` : `integer`
- `events_waiting` : `integer`
