PyUV Documentation

Release 1.4.0

Saúl Ibarra Corretgé

Jul 08, 2017

Contents

Python interface for libuv, a high performance asynchronous networking and platform abstraction library.

Note: pyuv's source code is hosted on GitHub

CHAPTER 1

Features:

- Non-blocking TCP sockets
- Non-blocking named pipes
- UDP support
- Timers
- Child process spawning
- Asynchronous DNS resolution (getaddrinfo & getnameinfo)
- Asynchronous file system APIs
- Thread pool scheduling
- High resolution time
- System memory information
- System CPUs information
- Network interfaces information
- ANSI escape code controlled TTY
- File system events
- IPC and TCP socket sharing between processes
- Arbitrary file descriptor polling
- Thread synchronization primitives

See also:

libuv's source code

See also:

Official libuv documentation

CHAPTER 2

Contents

pyuv — Python interface to libuv.

See also:

libuv's source code.

Objects

Loop — Event loop

class pyuv.**Loop**

Instantiate a new event loop. The instantiated loop is *not* the default event loop. In order to instantiate the default event loop `default_loop` classmethod should be used.

classmethod default_loop()

Create the *default* event loop. Most applications should use this event loop if only a single loop is needed.

run([*mode*])

Parameters **mode** (*int*) – Specifies the mode in which the loop will run. It can take 3 different values:

- `UV_RUN_DEFAULT`: Default mode. Run the event loop until there are no active handles or requests.
- `UV_RUN_ONCE`: Run a single event loop iteration.
- `UV_RUN_NOWAIT`: Run a single event loop iteration, but don't block for io.

Run the event loop. Returns True if there are pending operations and run should be called again or False otherwise.

stop()

Stops a running event loop. The action won't happen immediately, it will happen the next loop iteration,

but if stop was called before blocking for i/o a 0 timeout poll will be performed, so the loop will not block for i/o on that iteration.

now()

update_time()
Manage event loop time. `now` will return the current event loop time in milliseconds. It expresses the time when the event loop began to process events.

After an operation which blocks the event loop for a long time the event loop may have lost track of time. In that case `update_time` should be called to query the kernel for the time.

This are advanced functions not be used in standard applications.

queue_work(*work_callback*[, *done_callback*])

Parameters

- **work_callback** (*callable*) – Function that will be called in the thread pool.
- **done_callback** (*callable*) – Function that will be called in the caller thread after the given function has run in the thread pool.

  Callback signature: `done_callback(errorno)`. Errorno indicates if the request was cancelled (UV_ECANCELLED) or None, if it was actually executed.

Run the given function in a thread from the internal thread pool. A *WorkRequest* object is returned, which has a *cancel()* method that can be called to avoid running the request, in case it didn't already run.

Unix only: The size of the internal threadpool can be controlled with the *UV_THREADPOOL_SIZE* environment variable, which needs to be set before the first call to this function. The default size is 4 threads.

excepthook(*type*, *value*, *traceback*)
This function prints out a given traceback and exception to sys.stderr.

When an exception is raised and uncaught, the interpreter calls loop.excepthook with three arguments, the exception class, exception instance, and a traceback object. The handling of such top-level exceptions can be customized by assigning another three-argument function to loop.excepthook.

fileno()
Returns the file descriptor of the polling backend.

get_timeout()
Returns the poll timeout.

handles
Read only

List of handles in this loop.

alive
Read only

Checks whether the reference count, that is, the number of active handles or requests left in the event loop is non-zero aka if the loop is currently running.

`Handle` — Handle base class

class `pyuv.`**Handle**
Handle is an internal base class from which all handles inherit in pyuv. It provides all handles with a number of methods which are common for all.

close([*callback*])

Parameters **callback** (*callable*) – Function that will be called after the handle is closed.

Close the handle. After a handle has been closed no other operations can be performed on it, they will raise *HandleClosedError*.

Callback signature: `callback(handle)`

ref
Reference/unreference this handle. If running the event loop in default mode (UV_RUN_DEFAULT) the loop will exit when there are no more ref'd active handles left. Setting ref to True on a handle will ensure that the loop is maintained alive while the handle is active. Likewise, if all handles are unref'd, the loop would finish enven if they were all active.

loop
Read only

Loop object where this handle runs.

active
Read only

Indicates if this handle is active.

closed
Read only

Indicates if this handle is closing or already closed.

Timer — Timer handle

class pyuv.**Timer**(*loop*)

Parameters **loop** (*Loop*) – loop object where this handle runs (accessible through `Timer.loop`).

A `Timer` handle will run the supplied callback after the specified amount of seconds.

start(*callback*, *timeout*, *repeat*)

Parameters

- **callback** (*callable*) – Function that will be called when the `Timer` handle is run by the event loop.
- **timeout** (*float*) – The `Timer` will start after the specified amount of time.
- **repeat** (*float*) – The `Timer` will run again after the specified amount of time.

Start the `Timer` handle.

Callback signature: `callback(timer_handle)`.

stop()
Stop the `Timer` handle.

again()
Stop the `Timer`, and if it is repeating restart it using the repeat value as the timeout.

repeat
Get/set the repeat value. Note that if the repeat value is set from a timer callback it does not immediately take effect. If the timer was non-repeating before, it will have been stopped. If it was repeating, then the old repeat value will have been used to schedule the next timeout.

TCP — TCP handle

class pyuv.**TCP**(*loop*[, *family*])

Parameters

- **loop** (*Loop*) – loop object where this handle runs (accessible through TCP.loop).
- **family** (*int*) – Optionally specify the socket family. If specified and not AF_UNSPEC the socket will be created early.

The TCP handle provides asynchronous TCP functionality both as a client and server.

bind(*(ip, port, [flowinfo, [scope_id]]), [flags]*)

Parameters

- **ip** (*string*) – IP address to bind to.
- **port** (*int*) – Port number to bind to.
- **flowinfo** (*int*) – Flow info, used only for IPv6. Defaults to 0.
- **scope_id** (*int*) – Scope ID, used only for IPv6. Defaults to 0.
- **flags** (*int*) – Binding flags. Only pyuv.UV_TCP_IPV6ONLY is supported at the moment, which disables dual stack support on IPv6 handles.

Bind to the specified IP address and port.

listen(*callback*[, *backlog*])

Parameters

- **callback** (*callable*) – Callback to be called on every new connection. *accept()* should be called in that callback in order to accept the incoming connection.
- **backlog** (*int*) – Indicates the length of the queue of incoming connections. It defaults to 511.

Start listening for new connections.

Callback signature: `callback(tcp_handle, error)`.

accept(*client*)

Parameters **client** (*object*) – Client object where to accept the connection.

Accept a new incoming connection which was pending. This function needs to be called in the callback given to the *listen()* function.

connect(*(ip, port, [flowinfo, [scope_id]]), callback*)

Parameters

- **ip** (*string*) – IP address to connect to.
- **port** (*int*) – Port number to connect to.
- **flowinfo** (*int*) – Flow info, used only for IPv6. Defaults to 0.
- **scope_id** (*int*) – Scope ID, used only for IPv6. Defaults to 0.
- **callback** (*callable*) – Callback to be called when the connection to the remote endpoint has been made.

Initiate a client connection to the specified IP address and port.

Callback signature: `callback(tcp_handle, error)`.

open(*fd*)

Parameters **fd** (*int*) – File descriptor to be opened.

Open the given file descriptor (or SOCKET in Windows) as a TCP handle.

Note: The file descriptor will be closed when the *TCP* handle is closed, so if it was tasken from a Python socket object, it will be useless afterwards.

Note: Once a file desctiptor has been passed to the open function, the handle 'owns' it. When calling close() on the handle, the file descriptor will be closed. If you'd like to keep using it afterwards it's recommended to duplicate it (using os.dup) before passing it to this function.

Note: The fd won't be put in non-blocking mode, the user is responsible for doing it.

---

getsockname()

Return tuple containing IP address and port of the local socket. In case of IPv6 sockets, it also returns the flow info and scope ID (a 4 element tuple).

getpeername()

Return tuple containing IP address and port of the remote endpoint's socket. In case of IPv6 sockets, it also returns the flow info and scope ID (a 4 element tuple).

shutdown([*callback*])

Parameters **callback** (*callable*) – Callback to be called after shutdown has been performed.

Shutdown the outgoing (write) direction of the TCP connection.

Callback signature: `callback(tcp_handle, error)`.

write(*data*[, *callback*])

Parameters

- **data** (*object*) – Data to be written on the TCP connection. It can be any Python object conforming to the buffer interface or a sequence of such objects.
- **callback** (*callable*) – Callback to be called after the write operation has been performed.

Write data on the TCP connection.

Callback signature: `callback(tcp_handle, error)`.

try_write(*data*)

Parameters **data** (*object*) – Data to be written on the TCP connection. It can be any Python object conforming to the buffer interface.

Try to write data on the TCP connection. It will raise an exception (with UV_EAGAIN errno) if data cannot be written immediately or return a number indicating the amount of data written.

start_read(*callback*)

Parameters **callback** (*callable*) – Callback to be called when data is read from the remote endpoint.

Start reading for incoming data from the remote endpoint.

Callback signature: `callback(tcp_handle, data, error)`.

stop_read()
Stop reading data from the remote endpoint.

nodelay(*enable*)

Parameters **enable** (*boolean*) – Enable / disable nodelay option.

Enable / disable Nagle's algorithm.

keepalive(*enable*, *delay*)

Parameters

- **enable** (*boolean*) – Enable / disable keepalive option.
- **delay** (*int*) – Initial delay, in seconds.

Enable / disable TCP keep-alive.

simultaneous_accepts(*enable*)

Parameters **enable** (*boolean*) – Enable / disable simultaneous accepts.

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.

fileno()
Return the internal file descriptor (or SOCKET in Windows) used by the `TCP` handle.

> **Warning:** libuv expects you not to modify the file descriptor in any way, and if you do, things will very likely break.

send_buffer_size
Gets / sets the send buffer size.

receive_buffer_size
Gets / sets the receive buffer size.

family
Read only

Returns the socket address family.

write_queue_size
Read only

Returns the size of the write queue.

readable
Read only

Indicates if this handle is readable.

writable
Read only

Indicates if this handle is writable.

UDP — UDP handle

class pyuv.**UDP**(*loop*[, *family*])

Parameters

- **loop** (*Loop*) – loop object where this handle runs (accessible through UDP.loop).
- **family** (*int*) – Optionally specify the socket family. If specified and not AF_UNSPEC the socket will be created early.

The UDP handle provides asynchronous UDP functionality both as a client and server.

bind(*(ip, port, [flowinfo, [scope_id]]), [flags]*)

Parameters

- **ip** (*string*) – IP address to bind to.
- **port** (*int*) – Port number to bind to.
- **flowinfo** (*int*) – Flow info, used only for IPv6. Defaults to 0.
- **scope_id** (*int*) – Scope ID, used only for IPv6. Defaults to 0.
- **flags** (*int*) – Binding flags. Only pyuv.UV_UDP_IPV6ONLY is supported at the moment, which disables dual stack support on IPv6 handles.

Bind to the specified IP address and port. This function needs to be called always, both when acting as a client and as a server. It sets the local IP address and port from which the data will be sent.

open(*fd*)

Parameters **fd** (*int*) – File descriptor to be opened.

Open the given file descriptor (or SOCKET in Windows) as a UDP handle.

Note: The file descriptor will be closed when the *UDP* handle is closed, so if it was tasken from a Python socket object, it will be useless afterwards.

Note: Once a file desctiptor has been passed to the open function, the handle 'owns' it. When calling close() on the handle, the file descriptor will be closed. If you'd like to keep using it afterwards it's recommended to duplicate it (using os.dup) before passing it to this function.

Note: The fd won't be put in non-blocking mode, the user is responsible for doing it.

---

getsockname()

Return tuple containing IP address and port of the local socket. In case of IPv6 sockets, it also returns the flow info and scope ID (a 4 element tuple).

send(*(ip, port, [flowinfo, [scope_id]]), data, [callback]*)

Parameters

- **ip** (*string*) – IP address where data will be sent.
- **port** (*int*) – Port number where data will be sent.
- **flowinfo** (*int*) – Flow info, used only for IPv6. Defaults to 0.

- **scope_id** (*int*) – Scope ID, used only for IPv6. Defaults to 0.
- **data** (*object*) – Data to be sent over the UDP connection. It can be any Python object conforming to the buffer interface or a sequence of such objects.
- **callback** (*callable*) – Callback to be called after the send operation has been performed.

Send data over the UDP connection.

Callback signature: `callback(udp_handle, error)`.

try_send(*(ip, port)*, *data*)

Parameters **data** (*object*) – Data to be written on the UDP connection. It can be any Python object conforming to the buffer interface.

Try to send data on the UDP connection. It will raise an exception (with UV_EAGAIN errno) if data cannot be written immediately or return a number indicating the amount of data written.

start_recv(*callback*)

Parameters **callback** (*callable*) – Callback to be called when data is received on the bount IP address and port.

Start receiving data on the bound IP address and port.

Callback signature: `callback(udp_handle, (ip, port), flags, data, error)`. The flags attribute can only contain pyuv.UV_UDP_PARTIAL, in case the UDP packet was truncated.

stop_recv()
Stop receiving data.

set_membership(*multicast_address*, *membership*[, *interface*])

Parameters

- **multicast_address** (*string*) – Multicast group to join / leave.
- **membership** (*int*) – Flag indicating if the operation is join or leave. Flags: `pyuv.UV_JOIN_GROUP` and `pyuv.UV_LEAVE_GROUP`.
- **interface** (*string*) – Local interface address to use to join or leave the specified multicast group.

Join or leave a multicast group.

set_multicast_ttl(*ttl*)

Parameters **ttl** (*int*) – TTL value to be set.

Set the multicast Time To Live (TTL).

set_multicast_loop(*enable*)

Parameters **enable** (*boolean*) – On /off.

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

set_broadcast(*enable*)

Parameters **enable** (*boolean*) – On /off.

Set broadcast on or off.

set_ttl(*ttl*)

Parameters **ttl** (*int*) – TTL value to be set.

Set the Time To Live (TTL).

fileno()
Return the internal file descriptor (or SOCKET in Windows) used by the `UDP` handle.

> **Warning:** libuv expects you not to modify the file descriptor in any way, and if you do, things will very likely break.

send_buffer_size
Gets / sets the send buffer size.

receive_buffer_size
Gets / sets the receive buffer size.

family
Read only

Returns the socket address family.

`Pipe` — Named pipe handle

class `pyuv.`**Pipe**(*loop*, *ipc*)

Parameters

- **loop** (*Loop*) – loop object where this handle runs (accessible through `Pipe.loop`).
- **ipc** (*boolean*) – Indicates if this `Pipe` will be used for sharing handles.

The `Pipe` handle provides asynchronous named pipe functionality both as a client and server, supporting cross-process communication and handle sharing.

bind(*name*)

Parameters **name** (*string*) – Name of the pipe to bind to.

Bind to the specified pipe name. The `Pipe` handle is acting as a server in this case.

listen(*callback*[, *backlog*])

Parameters

- **callback** (*callable*) – Callback to be called on every new connection. *accept()* should be called in that callback in order to accept the incoming connection.
- **backlog** (*int*) – Indicates the length of the queue of incoming connections. It defaults to 511.

Start listening for new connections.

Callback signature: `callback(pipe_handle, error)`.

open(*fd*)

Parameters **fd** (*int*) – File descriptor to be opened.

Open the given file descriptor (or HANDLE in Windows) as a `Pipe`.

> **Note:** The file descriptor will be closed when the *Pipe* handle is closed, so if it was tasken from a Python socket object, it will be useless afterwards.

Note: Once a file desctiptor has been passed to the open function, the handle 'owns' it. When calling close() on the handle, the file descriptor will be closed. If you'd like to keep using it afterwards it's recommended to duplicate it (using os.dup) before passing it to this function.

Note: The fd won't be put in non-blocking mode, the user is responsible for doing it.

accept (*client*)

Parameters **client** (*object*) – Client object where to accept the connection.

Accept a new incoming connection which was pending. This function needs to be called in the callback given to the *listen()* function, or when the remote endpoint has shared a handle using the *handle* argument of *write()*. In either case the method *pending_handle_type()* tells you the type of handle to pass as the *client* argument.

connect (*name*, *callback*)

Parameters

- **name** (*string*) – Name of the pipe to connect to.
- **callback** (*callable*) – Callback to be called when the connection to the remote endpoint has been made.

Initiate a client connection to the specified named pipe.

Callback signature: `callback(pipe_handle, error)`.

shutdown ([*callback*])

Parameters **callback** (*callable*) – Callback to be called after shutdown has been performed.

Shutdown the outgoing (write) direction of the `Pipe` connection.

Callback signature: `callback(pipe_handle, error)`.

write (*data*[, *callback*[, *handle*]])

Parameters

- **data** (*object*) – Data to be written on the `Pipe` connection. It can be any Python object conforming to the buffer interface or a sequence of such objects.
- **callback** (*callable*) – Callback to be called after the write operation has been performed.
- **handle** (*object*) – Handle to send over the `Pipe`. Currently only `TCP`, `UDP` and `Pipe` handles are supported.

Write data on the `Pipe` connection.

Callback signature: `callback(tcp_handle, error)`.

try_write (*data*)

Parameters **data** (*object*) – Data to be written on the `Pipe` connection. It can be any Python object conforming to the buffer interface.

Try to write data on the `Pipe` connection. It will raise an exception if data cannot be written immediately or a number indicating the amount of data written.

start_read(*callback*)

Parameters **callback** (*callable*) – Callback to be called when data is read from the remote endpoint.

Start reading for incoming data from the remote endpoint.

Callback signature: `callback(pipe_handle, data, error)`.

stop_read()
Stop reading data from the remote endpoint.

pending_instances(*count*)

Parameters **count** (*int*) – Number of pending instances.

This setting applies to Windows only. Set the number of pending pipe instance handles when the pipe server is waiting for connections.

pending_handle_type()
Return the type of handle that is pending. The possible return values are `UV_TCP`, `UV_UDP` and `UV_NAMED_PIPE`, corresponding to a *TCP*, a *UDP* and a *Pipe* handle respectively. The special value `UV_UNKNOWN_HANDLE` means no handle is pending.

There are two situations when a handle becomes pending: a new connection is available on a listening socket, or a handle was shared by the remote endpoint using the *handle* argument to *write()*.

fileno()
Return the internal file descriptor (or HANDLE in Windows) used by the `Pipe` handle.

> **Warning:** libuv expects you not to modify the file descriptor in any way, and if you do, things will very likely break.

ipc
Read only

Indicates if this pipe is enabled for sharing handles.

send_buffer_size
Gets / sets the send buffer size.

receive_buffer_size
Gets / sets the receive buffer size.

write_queue_size
Read only

Returns the size of the write queue.

readable
Read only

Indicates if this handle is readable.

writable
Read only

Indicates if this handle is writable.

TTY — TTY controlling handle

class pyuv.**TTY**(*loop*, *fd*, *readable*)

Parameters

- **loop** (*Loop*) – loop object where this handle runs (accessible through TTY.loop).
- **fd** (*int*) – File descriptor to be opened as a TTY.
- **readable** (*bool*) – Specifies if the given fd is readable.

The TTY handle provides asynchronous stdin / stdout functionality.

shutdown([*callback*])

Parameters **callback** (*callable*) – Callback to be called after shutdown has been performed.

Shutdown the outgoing (write) direction of the TTY connection.

Callback signature: callback(tty_handle).

write(*data*[, *callback*])

Parameters

- **data** (*object*) – Data to be written on the TTY connection. It can be any Python object conforming to the buffer interface or a sequence of such objects.
- **callback** (*callable*) – Callback to be called after the write operation has been performed.

Write data on the TTY connection.

Callback signature: callback(tcp_handle, error).

try_write(*data*)

Parameters **data** (*object*) – Data to be written on the TTY connection. It can be any Python object conforming to the buffer interface.

Try to write data on the TTY connection. It will raise an exception if data cannot be written immediately or a number indicating the amount of data written.

start_read(*callback*)

Parameters **callback** (*callable*) – Callback to be called when data is read.

Start reading for incoming data.

Callback signature: callback(status_handle, data).

stop_read()
Stop reading data.

set_mode(*mode*)

Parameters **mode** (*int*) – TTY mode. 0 for normal, 1 for raw.

Set TTY mode.

get_winsize()
Get terminal window size.

fileno()
Return the internal file descriptor (or HANDLE in Windows) used by the TTY handle.

> **Warning:** libuv expects you not to modify the file descriptor in any way, and if you do, things will very likely break.

classmethod **reset_mode**()
Reset TTY settings. To be called when program exits.

write_queue_size
Read only

Returns the size of the write queue.

readable
Read only

Indicates if this handle is readable.

writable
Read only

Indicates if this handle is writable.

Poll — Poll handle

class pyuv.**Poll**(*loop*, *fd*)

Parameters

- **loop** (*Loop*) – loop object where this handle runs (accessible through Poll.loop).
- **fd** (*int*) – File descriptor to be monitored for readability or writability.

Poll handles can be used to monitor an arbitrary file descriptor for readability or writability. On Unix any file descriptor is supported but on Windows only sockets are supported.

Note: (From the libuv documentation) The uv_poll watcher is used to watch file descriptors for readability and writability, similar to the purpose of poll(2).

The purpose of uv_poll 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 other purpose is not recommended; uv_tcp_t, uv_udp_t, etc. provide an implementation that is much faster and more scalable than what can be achieved with uv_poll_t, especially on Windows.

It is possible that uv_poll occasionally signals 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.

The user should not close a file descriptor while it is being polled by an active uv_poll watcher. This can cause the poll watcher 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().

On windows only sockets can be polled with uv_poll. On unix any file descriptor that would be accepted by poll(2) can be used with uv_poll.

IMPORTANT: It is not okay to have multiple active uv_poll watchers for the same socket. This can cause libuv to assert. See this issue: https://github.com/saghul/pyuv/issues/54

---

start(*events*, *callback*)

Parameters

- **events** (*int*) – Mask of events that will be detected. The possible events are *pyuv.UV_READABLE pyuv.UV_WRITABLE*, or *pyuv.UV_DISCONNECT*.
- **callback** (*callable*) – Function that will be called when the `Poll` handle receives events.

Start or update the event mask of the `Poll` handle.

Callback signature: `callback(poll_handle, events, errorno)`.

Note: The `UV_DISCONNECT` event might not be triggered on all platforms.

stop()
Stop the `Poll` handle.

fileno()
Returns the file descriptor being monitored or -1 if the handle is closed.

`Process` — Child process spawning handle

`Process` handles allow spawning child processes which can be controlled (their stdin and stdout) with `Pipe` handles within an event loop.

classmethod `pyuv.`**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.

Note that 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.

classmethod `pyuv.`**spawn**(*loop*, *args*[, *executable*[, *env*[, *cwd*[, *uid*[, *gid*[, *flags*[, *stdio*[, *exit_callback*]]]]]]]])

Param Loop loop: *pyuv.Loop* instance where this handle belongs.

Parameters

- **args** (*list*) – Arguments for the new process. In case it's just the executable, it's possible to specify it as a string instead of a single element list.
- **executable** (*string*) – File to be executed. args[0] is taken in case it's not specified.
- **exit_callback** (*callable*) – Callback to be called when the process exits.
- **env** (*dict*) – Overrides the environment for the child process. If none is specified the one from the parent is used.
- **cwd** (*string*) – Specifies the working directory where the child process will be executed.
- **uid** (*int*) – UID of the user to be used if flag `UV_PROCESS_SETUID` is used.
- **gid** (*int*) – GID of the group to be used if flag `UV_PROCESS_SETGID` is used.
- **flags** (*int*) – Available flags:

- UV_PROCESS_SETUID: set child UID
- UV_PROCESS_SETGID: set child GID
- UV_PROCESS_WINDOWS_HIDE: hide the subprocess console window that would normally be created. This option is only meaningful on Windows systems. On unix it is silently ignored.
- UV_PROCESS_WINDOWS_VERBATIM_ARGUMENTS: pass arguments verbatim, that is, not enclosed in double quotes (Windows)
- UV_PROCESS_DETACHED: detach child process from parent

- **stdio** (*list*) – Sequence containing StdIO containers which will be used to pass stdio handles to the child process. See the StdIO class documentation for for information.

Spawn the specified child process.

Exit callback signature: `callback(process_handle, exit_status, term_signal)`.

pyuv.**kill**(*signal*)

Parameters **signal** (*int*) – Signal to be sent to the process.

Send the specified signal to the child process.

pyuv.**pid**
Read only

PID of the spawned process.

class pyuv.**StdIO**([[[*stream*], *fd*], *flags*])

Parameters

- **stream** (*object*) – Stream object.
- **fd** (*int*) – File descriptor.
- **flags** (*int*) – Flags.

Create a new container for passing stdio to a child process. Stream can be any stream object, that is TCP, Pipe or TTY. An arbitrary file descriptor can be passed by setting the fd parameter.

The operation mode is selected by setting the flags parameter:

•UV_IGNORE: this container should be ignored.

•UV_CREATE_PIPE: indicates a pipe should be created. UV_READABLE_PIPE and UV_WRITABLE_PIPE determine the direction of flow, from the child process' perspective. Both flags may be specified to create a duplex data stream.

•UV_INHERIT_FD: inherit the given file descriptor in the child.

•UV_INHERIT_STREAM: inherit the file descriptor of the given stream in the child.

Async — Async handle

class pyuv.**Async**(*loop*, *callback*)

Parameters

- **loop** (*Loop*) – loop object where this handle runs (accessible through Async.loop).

- **callback** (*callable*) – Function that will be called after the `Async` handle fires. It will be called in the event loop.

Calling event loop related functions from an outside thread is not safe in general. This is actually the only handle which is thread safe. The `Async` handle may be used to pass control from an outside thread to the event loop, as it will allow the calling thread to schedule a callback which will be called in the event loop thread.

send()
Start the `Async` handle. The callback will be called *at least* once.

Callback signature: `callback(async_handle)`

`Prepare` — Prepare handle

class `pyuv.`**Prepare**(*loop*)

Parameters **loop** (*Loop*) – loop object where this handle runs (accessible through `Prepare.loop`).

`Prepare` handles are usually used together with *Check* handles. They run just before the event loop ia about to block for I/O. The callback will be called *once* each loop iteration, before I/O.

start(*callback*)

Parameters **callback** (*callable*) – Function that will be called when the `Prepare` handle is run by the event loop.

Start the `Prepare` handle.

Callback signature: `callback(prepare_handle)`.

stop()
Stop the `Prepare` handle.

`Idle` — Idle handle

class `pyuv.`**Idle**(*loop*)

Parameters **loop** (*Loop*) – loop object where this handle runs (accessible through `Idle.loop`).

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

start(*callback*)

Parameters **callback** (*callable*) – Function that will be called when the `Idle` handle is run by the event loop.

Start the `Idle` handle.

Callback signature: `callback(idle_handle)`.

stop()
Stop the Idle handle.

Check — Check handle

class pyuv.**Check**(*loop*)

Parameters **loop** (*Loop*) – loop object where this handle runs (accessible through Check.loop).

Check handles are usually used together with *Prepare* handles. They run just after the event loop comes back after being blocked for I/O. The callback will be called *once* each loop iteration, after I/O.

start(*callback*)

Parameters **callback** (*callable*) – Function that will be called when the Check handle is run by the event loop.

Start the Check handle.

Callback signature: callback(check_handle).

stop()
Stop the Check handle.

Signal — Signal handle

class pyuv.**Signal**(*loop*)

Parameters **loop** (*Loop*) – loop object where this handle runs (accessible through Signal.loop).

Signal handles register for the specified signal and notify the use about the signal's occurrence through the specified callback.

start(*callback*, *signum*)

Parameters

- **callback** (*callable*) – Function that will be called when the specified signal is received.
- **signum** (*int*) – Specific signal that this handle listens to.

Start the Signal handle.

Callback signature: callback(signal_handle, signal_num).

stop()
Stop the Signal handle.

pyuv.dns — Asynchronous getaddrinfo and getnameinfo

pyuv.dns.**getaddrinfo**(*loop*, ..., *callback=None*)
Equivalent of *socket.getaddrinfo*. When *callback* is not None, this function returns a *GAIRequest* object which has a *cancel()* method that can be called in order to cancel the request.

Callback signature: callback(result, errorno).

When *callback* is None, this function is synchronous.

pyuv.dns.**getnameinfo**(*loop*, ..., *callback=None*)
Equivalent of *socket.getnameinfo*. When *callback* is not None, this function returns a *GNIRequest* object which has a *cancel()* method that can be called in order to cancel the request.

Callback signature: `callback(result, errorno)`.

When *callback* is None, this function is synchronous.

Note: libuv used to bundle c-ares in the past, so the c-ares bindings are now also a separated project. The examples directory contains an example on how to build a full DNS resolver using the `Channel` class provided by pycares together with pyuv.

---

pyuv.fs — Asynchronous filesystem operations

This module provides asynchronous file system operations. All functions return an instance of *FSRequest*, which has 3 public members:

- path: the path affecting the operation
- error: the error code if the operation failed, 0 if it succeeded
- result: for those operations returning results, it will be stored on this member.

These members will be populated before calling the callback, which has the following signature: `callback(loop, req)`

Note: All functions in the fs module except for the *FSEvent* and *FSPoll* classes support both synchronous and asynchronous modes. If you want to run it synchronous don't pass any callable as the *callback* argument, else it will run asynchronously. If the async form is used, then a *FSRequest* is returned when calling the functions, which has a *cancel()* method that can be called in order to cancel the request, in case it hasn't run yet.

Note: All functions that take a file descriptor argument must get the file descriptor resulting of a pyuv.fs.open call on Windows, else the operation will fail. This limitation doesn't apply to Unix systems.

---

pyuv.fs.**stat**(*loop*, *path*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – File to stat.
- **callback** (*callable*) – Function that will be called with the result of the function.

stat syscall.

pyuv.fs.**lstat**(*loop*, *path*[, *callback*])
Same as *pyuv.fs.stat()* but it also follows symlinks.

pyuv.fs.**fstat**(*loop*, *fd*[, *callback*])
Same as *pyuv.fs.stat()* but using a file-descriptor instead of the path.

pyuv.fs.**unlink**(*loop*, *path*[, *callback*])

Parameters

- **loop** – loop object where this function runs.

- **path** (*string*) – File to unlink.
- **callback** (*callable*) – Function that will be called with the result of the function.

Remove the specified file.

pyuv.fs.**mkdir**(*loop*, *path*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – Directory to create.
- **callback** (*callable*) – Function that will be called with the result of the function.

Create the specified directory.

pyuv.fs.**rmdir**(*loop*, *path*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – Directory to remove.
- **callback** (*callable*) – Function that will be called with the result of the function.

Remove the specified directory.

pyuv.fs.**rename**(*loop*, *path*, *new_path*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – Original file.
- **new_path** (*string*) – New name for the file.
- **callback** (*callable*) – Function that will be called with the result of the function.

Rename file.

pyuv.fs.**chmod**(*loop*, *path*, *mode*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – File which permissions will be changed.
- **mode** (*int*) – File permissions (ex. 0755)
- **callback** (*callable*) – Function that will be called with the result of the function.

Remove the specified directory.

pyuv.fs.**fchmod**(*loop*, *fd*, *mode*[, *callback*])

Same as *pyuv.fs.chmod()* but using a file-descriptor instead of the path.

pyuv.fs.**link**(*loop*, *path*, *new_path*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – Original file.
- **new_path** (*string*) – Name for the hard-link.

- **callback** (*callable*) – Function that will be called with the result of the function.

Create a hard-link.

pyuv.fs.**symlink**(*loop*, *path*, *new_path*, *flags*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – Original file.
- **new_path** (*string*) – Name for the symlink.
- **flags** (*int*) – flags to be used on Windows platform. If UV_FS_SYMLINK_DIR is specified the symlink will be created to a directory. If UV_FS_SYMLINK_JUNCTION a junction point will be created instead of a symlink.
- **callback** (*callable*) – Function that will be called with the result of the function.

Create a symlink.

pyuv.fs.**readlink**(*loop*, *path*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – Link file to process.
- **callback** (*callable*) – Function that will be called with the result of the function.

Read link file and return the original file path.

pyuv.fs.**chown**(*loop*, *path*, *uid*, *gid*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – File which persmissions will be changed.
- **uid** (*int*) – User ID.
- **gid** (*int*) – Group ID.
- **callback** (*callable*) – Function that will be called with the result of the function.

Changes ownership of a file.

pyuv.fs.**fchown**(*loop*, *fd*, *mode*[, *callback*])
Same as *pyuv.fs.chown()* but using a file-descriptor instead of the path.

pyuv.fs.**open**(*loop*, *path*, *flags*, *mode*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – File to open.
- **flags** (*int*) – Flags for opening the file. Check *os.O_* constants.
- **mode** (*int*) – Mode for opening the file. Check *stat.S_* constants.
- **callback** (*callable*) – Function that will be called with the result of the function.

Open file.

pyuv.fs.**close**(*loop*, *fd*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **fd** (*int*) – File-descriptor to close.
- **callback** (*callable*) – Function that will be called with the result of the function.

Close file.

pyuv.fs.**read**(*loop*, *fd*, *length*, *offset*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **fd** (*int*) – File-descriptor to read from.
- **length** (*int*) – Amount of bytes to be read.
- **offset** (*int*) – File offset.
- **callback** (*callable*) – Function that will be called with the result of the function.

Read from file.

pyuv.fs.**write**(*loop*, *fd*, *write_data*, *offset*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **fd** (*int*) – File-descriptor to read from.
- **write_data** (*string*) – Data to be written.
- **offset** (*int*) – File offset.
- **callback** (*callable*) – Function that will be called with the result of the function.

Write to file.

pyuv.fs.**fsync**(*loop*, *fd*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **fd** (*int*) – File-descriptor to sync.
- **callback** (*callable*) – Function that will be called with the result of the function.

Sync all changes made to file.

pyuv.fs.**fdatasync**(*loop*, *fd*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **fd** (*int*) – File-descriptor to sync.
- **callback** (*callable*) – Function that will be called with the result of the function.

Sync data changes made to file.

pyuv.fs.**ftruncate**(*loop*, *fd*, *offset*[, *callback*])

Parameters

- **loop** – loop object where this function runs.

- **fd** (*int*) – File-descriptor to truncate.
- **offset** (*int*) – File offset.
- **callback** (*callable*) – Function that will be called with the result of the function.

Truncate the contents of a file to the specified offset.

pyuv.fs.**scandir**(*loop*, *path*, *flags*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – Directory to list.
- **callback** (*callable*) – Function that will be called with the result of the function.

List files from a directory. The return value is a list of `DirEnt` object, which has 2 fields: *name* and *type*.

pyuv.fs.**sendfile**(*loop*, *out_fd*, *in_fd*, *in_offset*, *length*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **in_fd** (*int*) – File-descriptor to read from.
- **in_fd** – File-descriptor to write to.
- **length** (*int*) – Amount of bytes to be read.
- **offset** (*int*) – File offset.
- **callback** (*callable*) – Function that will be called with the result of the function.

Send a regular file to a stream socket.

pyuv.fs.**utime**(*loop*, *path*, *atime*, *mtime*[, *callback*])

Parameters

- **loop** – loop object where this function runs.
- **path** (*string*) – Directory to list.
- **atime** (*double*) – New accessed time.
- **mtime** (*double*) – New modified time.
- **callback** (*callable*) – Function that will be called with the result of the function.

Update file times.

pyuv.fs.**futime**(*loop*, *fd*, *atime*, *mtime*[, *callback*])
Same as *pyuv.fs.utime()* but using a file-descriptor instead of the path.

class pyuv.fs.**FSEvent**(*loop*)

Parameters **loop** (*Loop*) – loop object where this handle runs (accessible through `FSEvent.loop`).

FSEvent handles monitor a given path for changes.

start(*path*, *flags*, *callback*)

Parameters

- **path** (*string*) – Path to monitor for changes.
- **flags** (*int*) – Flags which control what events are watched for. Not used at the moment.

- **callback** (*callable*) – Function that will be called when the given path changes any of its attributes.

Start the FSEvent handle.

Callback signature: callback(fsevent_handle, filename, events, error).

stop()
Stop the FSEvent handle.

filename
Read only

Filename being monitored.

class pyuv.fs.**FSPoll**(*loop*)

Parameters **loop** (*Loop*) – loop object where this handle runs (accessible through FSPoll.loop).

FSPoll handles monitor a given path for changes by using stat syscalls.

start(*path*, *interval*, *callback*)

Parameters

- **path** (*string*) – Path to monitor for changes.
- **interval** (*float*) – How often to poll for events (in seconds).
- **callback** (*callable*) – Function that will be called when the given path changes any of its attributes.

Start the FSPoll handle.

Callback signature: callback(fspoll_handle, prev_stat, curr_stat, error).

stop()
Stop the FSPoll handle.

Module constants

pyuv.fs.**UV_FS_SYMLINK_DIR**

pyuv.fs.**UV_FS_SYMLINK_JUNCTION**

pyuv.fs.**UV_RENAME**

pyuv.fs.**UV_CHANGE**

pyuv.fs.**UV_FS_EVENT_WATCH_ENTRY**

pyuv.fs.**UV_FS_EVENT_STAT**

pyuv.fs.**UV_DIRENT_UNKNOWN**

pyuv.fs.**UV_DIRENT_FILE**

pyuv.fs.**UV_DIRENT_DIR**

pyuv.fs.**UV_DIRENT_LINK**

pyuv.fs.**UV_DIRENT_FIFO**

pyuv.fs.**UV_DIRENT_SOCKET**

pyuv.fs.**UV_DIRENT_CHAR**

pyuv.fs.**UV_DIRENT_BLOCK**

pyuv.error — Exception definitions

This module contains the definition of the different exceptions that are used throughout *puyv*.

exception pyuv.**UVError**
Base exception class. Parent of all other exception classes.

exception pyuv.**ThreadError**
Exception raised by thread module operations.

exception pyuv.**HandleError**
Base exception class. Parent of all other handle exception classes.

exception pyuv.**HandleClosedError**
Exception raised if a handle is already closed and some function is called on it.

exception pyuv.**StreamError**
Base exception class for stream errors.

exception pyuv.**AsyncError**
Exception raised if an error is found when calling `Async` handle functions.

exception pyuv.**CheckError**
Exception raised if an error is found when calling `Check` handle functions.

exception pyuv.**DNSError**
Exception raised if an error is found when calling `DNSResolver` functions.

exception pyuv.**FSError**
Exception raised if an error is found when calling funcions from the `fs` module.

exception pyuv.**FSEventError**
Exception raised if an error is found when calling `FSEvent` handle functions.

exception pyuv.**IdleError**
Exception raised if an error is found when calling `Idle` handle functions.

exception pyuv.**PipeError**
Exception raised if an error is found when calling `Pipe` handle functions.

exception pyuv.**PrepareError**
Exception raised if an error is found when calling `Prepare` handle functions.

exception pyuv.**PollError**
Exception raised if an error is found when calling `Poll` handle functions.

exception pyuv.**SignalError**
Exception raised if an error is found when calling `Signal` handle functions.

exception pyuv.**TCPError**
Exception raised if an error is found when calling `TCP` handle functions.

exception pyuv.**TimerError**
Exception raised if an error is found when calling `Timer` handle functions.

exception pyuv.**UDPError**
Exception raised if an error is found when calling `UDP` handle functions.

exception pyuv.**TTYError**
Exception raised if an error is found when calling `TTY` handle functions.

exception pyuv.**ProcessError**
Exception raised if an error is found when calling `Process` handle functions.

pyuv.errno — Error constant definitions

This module contains the defined error constants from libuv and c-ares.

IMPORTANT: The errno codes in pyuv don't necessarily match those in the Python *errno* module.

pyuv.errno.**errorcode**
Mapping (code, string) with libuv error codes.

pyuv.errno.**strerror**(*errorno*)

Parameters **errorno** (*int*) – Error number.

Get the string representation of the given error number.

pyuv.thread — Thread synchronization primitives

class pyuv.thread.**Barrier**(*count*)

Parameters **count** (*int*) – Initialize the barrier for the given amount of threads

wait()
Synchronize all the participating threads at the barrier.

class pyuv.thread.**Mutex**

lock()
Lock this mutex.

unlock()
Unlock this mutex.

trylock()
Try to lock the mutex. If the lock could be acquired True is returned, False otherwise.

class pyuv.thread.**RWLock**

rdlock()
Lock this rwlock for reading.

rdunlock()
Unlock this rwlock for reading.

tryrdlock()
Try to lock the rwlock for reading. If the lock could be acquired True is returned, False otherwise.

wrlock()
Lock this rwlock for writing.

wrunlock()
Unlock this rwlock for writing.

trywrlock()
Try to lock the rwlock for writing. If the lock could be acquired True is returned, False otherwise.

class pyuv.thread.**Condition**(*lock*)

Parameters **lock** (Mutex) – Lock to be used by this condition.

signal()
Unblock at least one of the threads that are blocked on this condition.

broadcast()
Unblock all threads blocked on this condition.

wait()
Block on this condition variable, the mutex lock must be held.

timedwait(*timeout*)

Parameters **timeout** (*double*) – Time to wait until condition is met before giving up.

Wait for the condition to be met, give up after the specified timeout.

class pyuv.thread.**Semaphore**(*count=1*)

Parameters **count** (*int*) – Initialize the semaphore with the given counter value.

post()
Increment (unlock) the semaphore.

wait()
Decrement (lock) the semaphore.

trywait()
Try to decrement (lock) the semaphore. If the counter could be decremented True is returned, False otherwise.

pyuv.util — Miscelaneous utilities

pyuv.util.**hrtime**()
Get the high-resolution system time. It's given in nanoseconds, even if the system doesn't support nanosecond precision.

pyuv.util.**get_free_memory**()
Get the system free memory (in bytes).

pyuv.util.**get_total_memory**()
Get the system total memory (in bytes).

pyuv.util.**loadavg**()
Get the system load average.

pyuv.util.**uptime**()
Get the current uptime.

pyuv.util.**resident_set_size**()
Get the current resident memory size.

pyuv.util.**interface_addresses**()
Get interface addresses information.

pyuv.util.**cpu_info**()
Get CPUs information.

pyuv.util.**getrusage**()
Get information about resource utilization. getrusage(2) implementation which always uses RUSAGE_SELF. Limited support on Windows.

pyuv.util.**guess_handle_type**()
Given a file descriptor, returns the handle type in the form of a constant (integer). The user can compare it with constants exposed in pyuv.*, such as UV_TTY, UV_TCP, and so on.

class pyuv.util.**SignalChecker**(*loop*, *fd*)

Parameters

- **loop** (*Loop*) – loop object where this handle runs (accessible through `SignalChecker.loop`).
- **fd** (*int*) – File descriptor to be monitored for readability.

SignalChecker is a handle which can be used to interact with signals set up by the standard *signal* module.

Here is how it works: the user is required to get a pair of file descriptors and put them in nonblocking mode. These descriptors can be either a *os.pipe()*, a *socket.socketpair()* or a manually made pair of connected sockets. One of these descriptors will be used just for writing, and the other end for reading. The user must use *signal.set_wakeup_fd* and register the write descriptor. The read descriptor must be passed to this handle. When the process receives a signal Python will write on the write end of the socket pair and it will cause the SignalChecker to wakeup the loop and call the Python C API function to process signals: *PyErr_CheckSignals*.

It's better to use the *Signal* handle also provided by this library, because it doesn't need any work from the user and it works on any thread. This handle merely exists for some cases which you don't probably care about, or if you want to use the *signal* module directly.

start()
Start the signal checking handle.

stop()
Stop the signal checking handle.

Reference counting scheme

(This section is about the reference counting scheme used by libuv, it's not related in any way to the reference counting model used by CPython)

The event loop runs (when *Loop.run* is called) until there are no more active handles. What does it mean for a handle to be 'active'? Depends on the handle type:

- Timers: active when ticking
- Sockets (TCP, UDP, Pipe, TTY): active when reading, writing or listening
- Process: active until the child process dies
- Idle, Prepare, Check, Poll, FSEvent, FSPoll: active once they are started
- Async: always active, until closed

All handles have the *ref* read-write property available in order to modify the default behavior. These functions operate at the handle level (that is, the *handle* is referenced, not the loop) so if a handle is ref'd it will maintain the loop alive even if not active.

CHAPTER 3

Examples

UDP Echo server

```
from __future__ import print_function

import signal
import pyuv


def on_read(handle, ip_port, flags, data, error):
    if data is not None:
        handle.send(ip_port, data)

def signal_cb(handle, signum):
    signal_h.close()
    server.close()


print("PyUV version %s" % pyuv.__version__)

loop = pyuv.Loop.default_loop()

server = pyuv.UDP(loop)
server.bind(("0.0.0.0", 1234))
server.start_recv(on_read)

signal_h = pyuv.Signal(loop)
signal_h.start(signal_cb, signal.SIGINT)

loop.run()

print("Stopped!")
```

TCP Echo server

```
from __future__ import print_function

import signal
import pyuv


def on_read(client, data, error):
    if data is None:
        client.close()
        clients.remove(client)
        return
    client.write(data)

def on_connection(server, error):
    client = pyuv.TCP(server.loop)
    server.accept(client)
    clients.append(client)
    client.start_read(on_read)

def signal_cb(handle, signum):
    [c.close() for c in clients]
    signal_h.close()
    server.close()


print("PyUV version %s" % pyuv.__version__)

loop = pyuv.Loop.default_loop()
clients = []

server = pyuv.TCP(loop)
server.bind(("0.0.0.0", 1234))
server.listen(on_connection)

signal_h = pyuv.Signal(loop)
signal_h.start(signal_cb, signal.SIGINT)

loop.run()
print("Stopped!")
```

TCP Echo server using Poll handles

```
import sys
import socket
import signal
import weakref
import errno
import logging
import pyuv
```

```
logging.basicConfig(level=logging.DEBUG)

STOPSIGNALS = (signal.SIGINT, signal.SIGTERM)
NONBLOCKING = (errno.EAGAIN, errno.EWOULDBLOCK)
if sys.platform == "win32":
    NONBLOCKING = NONBLOCKING + (errno.WSAEWOULDBLOCK,)


class Connection(object):

    def __init__(self, sock, address, loop):
        self.sock = sock
        self.address = address
        self.sock.setblocking(0)
        self.buf = ""
        self.watcher = pyuv.Poll(loop, self.sock.fileno())
        self.watcher.start(pyuv.UV_READABLE, self.io_cb)
        logging.debug("{0}: ready".format(self))

    def reset(self, events):
        self.watcher.start(events, self.io_cb)

    def handle_error(self, msg, level=logging.ERROR, exc_info=True):
        logging.log(level, "{0}: {1} --> closing".format(self, msg), exc_info=exc_
↪info)
        self.close()

    def handle_read(self):
        try:
            buf = self.sock.recv(1024)
        except socket.error as err:
            if err.args[0] not in NONBLOCKING:
                self.handle_error("error reading from {0}".format(self.sock))
        if buf:
            self.buf += buf
            self.reset(pyuv.UV_READABLE | pyuv.UV_WRITABLE)
        else:
            self.handle_error("connection closed by peer", logging.DEBUG, False)

    def handle_write(self):
        try:
            sent = self.sock.send(self.buf)
        except socket.error as err:
            if err.args[0] not in NONBLOCKING:
                self.handle_error("error writing to {0}".format(self.sock))
        else :
            self.buf = self.buf[sent:]
            if not self.buf:
                self.reset(pyuv.UV_READABLE)

    def io_cb(self, watcher, revents, error):
        if error is not None:
            logging.error("Error in connection: %d: %s" % (error, pyuv.errno.
↪strerror(error)))
            return
        if revents & pyuv.UV_READABLE:
            self.handle_read()
        elif revents & pyuv.UV_WRITABLE:
```

```
            self.handle_write()

    def close(self):
        self.watcher.stop()
        self.watcher = None
        self.sock.close()
        logging.debug("{0}: closed".format(self))


class Server(object):

    def __init__(self, address):
        self.sock = socket.socket()
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.sock.bind(address)
        self.sock.setblocking(0)
        self.address = self.sock.getsockname()
        self.loop = pyuv.Loop.default_loop()
        self.poll_watcher = pyuv.Poll(self.loop, self.sock.fileno())
        self.async = pyuv.Async(self.loop, self.async_cb)
        self.conns = weakref.WeakValueDictionary()
        self.signal_watchers = set()

    def handle_error(self, msg, level=logging.ERROR, exc_info=True):
        logging.log(level, "{0}: {1} --> stopping".format(self, msg), exc_info=exc_
↪info)
        self.stop()

    def signal_cb(self, handle, signum):
        self.async.send()

    def async_cb(self, handle):
        handle.close()
        self.stop()

    def io_cb(self, watcher, revents, error):
        try:
            while True:
                try:
                    sock, address = self.sock.accept()
                except socket.error as err:
                    if err.args[0] in NONBLOCKING:
                        break
                    else:
                        raise
                else:
                    self.conns[address] = Connection(sock, address, self.loop)
        except Exception:
            self.handle_error("error accepting a connection")

    def start(self):
        self.sock.listen(socket.SOMAXCONN)
        self.poll_watcher.start(pyuv.UV_READABLE, self.io_cb)
        for sig in STOPSIGNALS:
            handle = pyuv.Signal(self.loop)
            handle.start(self.signal_cb, sig)
            self.signal_watchers.add(handle)
        logging.debug("{0}: started on {0.address}".format(self))
```

```
        self.loop.run()
        logging.debug("{0}: stopped".format(self))

    def stop(self):
        self.poll_watcher.stop()
        for watcher in self.signal_watchers:
            watcher.stop()
        self.signal_watchers.clear()
        self.sock.close()
        for conn in self.conns.values():
            conn.close()
        logging.debug("{0}: stopping".format(self))


if __name__ == "__main__":
    server = Server(("127.0.0.1", 9876))
    server.start()
```

Standard IO Echo server using Pipe handles

```
import signal
import sys
import pyuv


def on_pipe_read(handle, data, error):
    if data is None or data == b"exit":
        pipe_stdin.close()
        pipe_stdout.close()
    else:
        pipe_stdout.write(data)

def signal_cb(handle, signum):
    if not pipe_stdin.closed:
        pipe_stdin.close()
    if not pipe_stdin.closed:
        pipe_stdout.close()
    signal_h.close()


loop = pyuv.Loop.default_loop()

pipe_stdin = pyuv.Pipe(loop)
pipe_stdin.open(sys.stdin.fileno())
pipe_stdin.start_read(on_pipe_read)

pipe_stdout = pyuv.Pipe(loop)
pipe_stdout.open(sys.stdout.fileno())

signal_h = pyuv.Signal(loop)
signal_h.start(signal_cb, signal.SIGINT)

loop.run()
```

Standard IO Echo server using TTY handles

```
from __future__ import print_function

import signal
import sys
import pyuv


def on_tty_read(handle, data, error):
    if data is None or data == b"exit":
        tty_stdin.close()
        tty_stdout.close()
    else:
        tty_stdout.write(data)

def signal_cb(handle, signum):
    tty_stdin.close()
    tty_stdout.close()
    signal_h.close()


loop = pyuv.Loop.default_loop()

tty_stdin = pyuv.TTY(loop, sys.stdin.fileno(), True)
tty_stdin.start_read(on_tty_read)
tty_stdout = pyuv.TTY(loop, sys.stdout.fileno(), False)

if sys.platform != "win32":
    print("Window size: (%d, %d)" % tty_stdin.get_winsize())

signal_h = pyuv.Signal(loop)
signal_h.start(signal_cb, signal.SIGINT)

loop.run()

pyuv.TTY.reset_mode()
```

CHAPTER 4

ToDo

Things yet to be done

```
- Better docstrings!

Issue tracker: https://github.com/saghul/pyuv/issues
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

O

P

Q

R

S

T

U

W