User Guide to lunix, Comprehensive Unix API Module for Lua

William Ahern

August 1, 2014

Contents

1 About

`lunix` is a bindings library module to common Unix system APIs. The module is regularly tested with Linux/glibc, OS X, FreeBSD, NetBSD, OpenBSD, Solaris, and AIX. The best way to describe it is in contradistinction to `luaposix`, the most popular bindings module for Unix APIs in Lua.

Thread-safety Unlike `luaposix`, it strives to be as thread-safe as possible on the host platform. Interfaces like `strerror_r` and `O_CLOEXEC` are used throughout. The module even includes a novel solution for the inherently non-thread-safe `umask` system call, where calling `umask` from one thread might result in another thread creating a file with unsafe or unexpected permissions.

POSIX Extensions Unlike `luaposix`, the library does not restrict itself to POSIX, and emulates an interface when not available natively on a supported platform. For example, the library provides `arc4random` (absent on Linux and Solaris), `clock_gettime` (absent on OS X), and a thread-safe `timegm` (absent on Solaris).

Leak-safety Unlike `luaposix`, the library prefers dealing with `FILE` handles rather than raw integer descriptors. This helps to mitigate and prevent leaks or double-close bugs—a common source of problems in, e.g., asynchronous applications. Routines like `chdir` or `opendir` transparently accept string paths, FILE handles, DIR handles, or even a raw integer descriptors.

2 Dependencies

2.1 Operating Systems

`lunix` targets modern POSIX-conformant and POSIX-aspiring systems. But unlike `luaposix` it branches out to implement common GNU and BSD extensions. All interfaces are available on all supported platforms, regardless of whether the platform provides a native interface.

I try to regularly compile and test the module against recent versions of OS X, Linux/glibc, FreeBSD, NetBSD, OpenBSD, Solaris, and AIX.

2.2 Libraries

2.2.1 Lua 5.1, 5.2, 5.3

`lunix` targets Lua 5.1 and above.

2.3 GNU Make

The Makefile requires GNU Make, usually installed as gmake on platforms other than Linux or OS X. The actual `Makefile` proxies to `GNUmakefile`. As long as `gmake` is installed on non-GNU systems you can invoke your system's `make`.

3 Installation

The module is composed of a single C source file to simplify compilation across environments. Because there several extant versions of Lua often used in parallel on the same system, there are individual targets to build and install the module for each supported Lua version. The targets `all` and `install` will attempt to build and install both Lua 5.1 and 5.2 modules.

Note that building and installation and can accomplished in a single step by simply invoking one of the install targets with all the necessary variables defined.

3.1 Building

There is no separate `./configure` step. System introspection occurs during compile-time. However, the "`configure`" make target can be used to cache the build environment so one needn't continually use a long command-line invocation.

All the common GNU-style compiler variables are supported, including `CC`, `CPPFLAGS`, `CFLAGS`, `LDFLAGS`, and `SOFLAGS`. Note that you can specify the path to Lua 5.1, Lua 5.2, and Lua 5.3 include headers at the same time in CPPFLAGS; the build system will work things out to ensure the correct headers are loaded when compiling each version of the module.

3.1.1 Targets

`all`
: Build modules for Lua 5.1 and 5.2.

`all5.1`
: Build Lua 5.1 module.

`all5.2`
: Build Lua 5.2 module.

`all5.3`
: Build Lua 5.3 module.

3.2 Installing

All the common GNU-style installation path variables are supported, including `prefix`, `bindir`, `libdir`, `datadir`, `includedir`, and `DESTDIR`. These additional path variables are also allowed:

`lua51path`
: Install path for Lua 5.1 modules, e.g. `$(prefix)/share/lua/5.1`

`lua51cpath`
: Install path for Lua 5.1 C modules, e.g. `$(prefix)/lib/lua/5.1`

`lua52path`
: Install path for Lua 5.2 modules, e.g. `$(prefix)/share/lua/5.2`

`lua52cpath`
: Install path for Lua 5.2 C modules, e.g. `$(prefix)/lib/lua/5.2`

`lua53path`
: Install path for Lua 5.3 modules, e.g. `$(prefix)/share/lua/5.3`

`lua53cpath`
: Install path for Lua 5.3 C modules, e.g. `$(prefix)/lib/lua/5.3`

3.2.1 Targets

`install`
: Install modules for Lua 5.1 and 5.2.

`install5.1`
: Install Lua 5.1 module.

`install5.2`
: Install Lua 5.2 module.

`install5.3`
: Install Lua 5.3 module.

4 Usage

4.1 Modules

4.1.1 `unix`

At present `lunix` provides a single module of routines.

`environ[]`

Binding to the process-global `environ` array using metamethods.

__index
: Utilizes the internal `getenv` binding.

__newindex
: Utilizes the internal `setenv` binding.

__pairs
: Takes a snapshot of the environ table to be used by the returned iterator for key–value loops. *Other than Solaris*[1]*, no system supports thread-safe access of the environ global.*

__ipairs
: Similar to __pairs, but the iterator returns an index integer as the key followed by the environment variable as a single string—"FOO=BAR".

__call
: Identical to the __pairs metamethod, to be used to create an iterator directly as Lua 5.1 doesn't support `__pairs`.

`arc4random()`

Returns a cryptographically strong uniformly random 32-bit integer as a Lua number. On Linux the `RANDOM_UUID` `sysctl` feature is used to seed the generator. This avoids fiddling with file descriptors, and also works in a chroot jail. On other platforms without a native `arc4random` interface, such as Solaris, the implementation must resort to /dev/urandom for seeding.

Note that unlike the original implementation on OpenBSD, `arc4random` on OS X and FreeBSD (prior to 10.0) seeds itself from /dev/urandom. This could cause problems in chroot jails.

`arc4random_buf(`n`)`

Returns a string of length n containing cryptographically strong random octets using the same CSPRNG underlying `arc4random`.

[1] See https://blogs.oracle.com/pgdh/entry/caring_for_the_environment_making

`arc4random_stir()`

Stir the arc4random entropy pool using the best available resources. This normally should be unnecessary.

`arc4random_uniform([`*n*`])`

Returns a cryptographically strong uniform random integer in the interval $[0, n-1]$ where $n \leq 2^{32}$. If n is omitted the interval is $[0, 2^{32}-1]$ and effectively behaves like `arc4random`.

`chdir(`*dir*`)`

If *dir* is a string, attempts to change the current working directory using `chdir`. Otherwise, if *dir* is a FILE handle referencing a directory, or an integer file descriptor referencing a directory, attempts to change the current working directory using `fchdir`.

Returns `true` on success, otherwise returns `false`, an error string, and an integer system error.

`chown(`*file*`[, `*uid*`][, `*gid*`])`

file may either be a string path for use with `chown`, or a FILE handle or integer file descriptor for use with `fchown`. *uid* and *gid* may be integer values or symbolic string names.

Returns `true` on success, otherwise returns `false`, an error string, and an integer system error.

`chroot(`*path*`)`

Attempt to `chroot` to the specified string *path*.

Returns `true` on success, otherwise returns `false`, an error string, and an integer system error.

`clock_gettime(`*id*`)`

id should be the string "realtime" or "monotonic", or the integer constant CLOCK_REALTIME or CLOCK_MONOTONIC.

Returns a time value as a Lua floating point number, otherwise returns `nil`, an error string, and an integer system error.

`closedir(`*dir*`)`

Closes the DIR handle, releasing the underlying file descriptor.

`execve(`*path*`[, `*argv*`][, `*env*`])`

Executes *path*, replacing the existing process image. *path* should be an absolute pathname as the $PATH environment variable is not used. *argv* is a table or ipairs–iterable object specifying the argument vector to pass to the new process image. Traditionally the first such argument should be the basename of *path*, but this is not enforced. If absent or empty the new process image will be passed an empty argument vector. *env* is a table or ipairs–iterable object specifying the new environment. If absent or empty the new process image will contain an empty environment.

On success never returns. On failure returns `false`, an error string, and an integer system error.

`execl(`*path*`, ...)`

Executes *path*, replacing the existing process image. The $PATH environment variable is not used. Any subsequent arguments are passed to the new process image. The new process image inherits the current environment table.

On success never returns. On failure returns `false`, an error string, and an integer system error.

`execlp(`*file*`, ...)`

Executes *file*, replacing the existing process image. The $PATH environment variable is used to search for *file*. Any subsequent arguments are passed to the new process image. The new process image inherits the current environment table.

On success never returns. On failure returns `false`, an error string, and an integer system error.

`execvp(`*file*`[, `*argv*`])`

Executes *file*, replacing the existing process image. The $PATH environment variable is used to search for *file*. Any subsequent arguments are passed to the new process image. The new process image inherits the current environment table.

On success never returns. On failure returns `false`, an error string, and an integer system error.

`_exit([`*status*`])`

Exits the process immediately without first flushing and closing open streams, or calling `atexit` handlers. If *status* is boolean `true` or `false`, exits with EXIT_SUCCESS or EXIT_FAILURE, respectively. Otherwise, *status* is an optional integer status value which defaults to 0 (EXIT_SUCCESS).

`exit([`*status*`])`

Like `_exit`, but first flushes and closes open streams, and calls `atexit` handlers.

`fork()`

Forks a new process. On success returns the PID of the new process in the parent and the integer 0 in the child. Otherwise returns `false`, an error string, and an integer system error.

`getegid()`

Returns the effective process GID as a Lua number.

`getenv(`*name*`)`

Returns the value of the environment variable *name* as a string, or `nil` if it does not exist.

Not thread-safe on any system other than Solaris[2] *and NetBSD*[3]*. On Linux* `getenv` *is thread-tolerant as pointers returned from* `getenv` *will remain valid throughout the lifetime of the process, but Linux will write over existing values on update so concurrent use with* `setenv` *could lead to inconsistent views.*

`geteuid()`

Returns the effective process UID as a Lua number.

`getmode(`*mode*`[, `*omode*`])`

The `getmode` interface derives from the routine so-named in almost every `chmod(1)` utility implementation and which exposes the parser for symbolic file permissions.

mode should be a symbolic mode value with a valid syntax as described by POSIX within the `chmod(1)` utility man page. If specified, *omode* should be an integer or a string in decimal, hexidecimal, or octal notation, and represents the original mode value used by the symbolic syntax for inheritance.

`getgid()`

Returns the real process GID as a Lua number.

`getgrnam(`*grp*`[, `...`])`

grp is an integer GID or string symbolic group name suitable for use by either `getgrgid(3)` or `getgrnam(3)`, respectively.

If no other arguments are specified, on success returns a table with the following fields

.name
: Symbolic group name as a string, or `nil` if absent.

.passwd
: Password information as a string, or `nil` if absent.

.gid
: GID as integer.

.mem
: Array of supplementary group names, or `nil` if absent.

If additional arguments are given, on success each field specified (as named above) is returned as part of the return value list. "members" may be used as an alternative to "mem". Note that the return value may be `nil` if the field was absent.

If no group was found, returns `nil` followed by the error string "no such group".

If a system error occurred, returns `nil`, an error string, and an integer system error.

[2] See https://blogs.oracle.com/pgdh/entry/caring_for_the_environment_making

[3] NetBSD provides `getenv_r(3)`

`getifaddrs([...])`

Returns an iterator over the current system network interfaces on success. If a system error occurred, returns `nil`, an error string, and an integer system error.

If no arguments are specified, each invocation of the iterator returns a table with the following fields

.name
: Interface symbolic name as a string.

.flags
: Interface flags as an integer bit field.

.family
: Interface address family as an integer.

.addr
: Interface address as a string, or `nil` if of an unknown address family.

.netmask
: Interface address netmask as a string, or `nil` if absent or of an unknown address family.

.prefixlen
: Interface address prefixlen as an integer, or `nil` if absent or of an unknown address family.

.dstaddr
: Interface destination address if point-to-point, or `nil` if absent or of an unknown address family.

.broadaddr
: Interface broadcast address, or `nil` if absent or of an unknown address family.

If arguments are given, each field specified (as named above) is returned as part of the return value list on every invocation of the iterator.

`getpid()`

Returns the process ID as a Lua number.

`getpwnam(`*usr*`[, ...])`

usr is an integer UID or string symbolic user name suitable for use by either `getpwuid(3)` or `getpwnam(3)`, respectively.

If no other arguments are specified, on success returns a table with the following fields

.name
: Symbolic user name as a string, or `nil` if absent.

.passwd
: Password information as a string, or `nil` if absent.

.uid
: UID as integer.

.gid
: Primary GID as integer.

.dir
: Home directory path, or `nil` if absent.

.shell
: Login shell path, or `nil` if absent.

.gecos
: Additional user information, or `nil` if absent.

If additional arguments are given, on success each field specified (as named above) is returned as part of the return value list. Note that the return value may be `nil` if the value was empty in the database.

If no user was found, returns `nil` followed by the error string "no such user".

If a system error occurred, returns `nil`, an error string, and an integer system error.

`gettimeofday([`*ints*`])`

Returns the current time as a Lua floating point number or, if *ints* is `true`, as two integers representing seconds and microseconds.

On failure returns `nil`, an error string, and an integer system error.

`getuid()`

Returns the real process UID as a Lua number.

`issetugid()`

Returns `true` if the process environment is considered unsafe because of setuid, setgid, or similar operations, otherwise `false`.

`kill(`*pid*`, `*signo*`)`

Sends signal *signo* to process or process group *pid*. Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`link(`*path*1`, `*path*2`)`

Creates a new directory entry at *path*2 as a hard link to *path*1.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`mkdir(`*path*`[, `*mode*`])`

Create a new directory at *path*. *mode*, if specified, should be a symbolic mode string following the POSIX syntax as described by the `chmod(1)` utility man page. Otherwise, *mode* defaults to 0777. In either case, *mode* is masked by the process umask.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`mkpath(`*path*`[, `*mode*`][, `*imode*`])`

Like `mkdir`, but also creates intermediate directories if missing. *imode* is the mode for intermediate directories. Like *mode* it is restricted by the process umask, but unlike *mode* the user write bit is unconditionally set to ensure the full path can be created.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`opendir(`*path*`|`*file*`|`*dir*`|`*fd*`)`

Creates a DIR handle for reading directory entries. Caller may specify a path string, a Lua FILE handle, another DIR handle, or an integer descriptor. In the latter three cases, the underlying descriptor is duplicated using `dup3` (if available) or `dup2` because there's no safe way to steal the descriptor from existing FILE or DIR handles. But it's not a good idea to mix reads between the two original and duplicated descriptors as they will normally share the same open file entry in the kernel, including the same position cursor.[4]

Returns a DIR handle on success, otherwise nil, an error string, and an integer system error.

`raise(`*signo*`)`

Sends signal *signo* to calling thread. Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`readdir(`*dir*`[, `*field* ...`])`

Reads the next directory entry. If no field arguments are specified, on success returns a table with the following fields

.name
: Name of file.

.ino
: Inode of file.

.type
: A numeric value describing the file type, similar to the "mode" field returned by `stat`, except without any permission bits present. You can pass this value to `S_ISREG`, `S_ISDIR`, `S_ISFIFO`, etc.

  Available on Linux and BSD derivatives, but, e.g., will be `nil` on Solaris.

[4] In the future may add ability to open /proc/self/fd or /dev/fd entries, which should create a new open file entry.

If additional arguments are given, on success each field specified (as named above) is returned as part of the return value list. Note that the return value may be `nil` if the value was unavailable.

If the end of directory entries has been reached, returns `nil`.

If a system error occurred, returns `nil`, an error string, and an integer system error.

`rename(`*path*1`,` *path*2`)`

Renames the file *path*1 to *path*2. The paths must reside on the same device.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`rewinddir(`*dir*`)`

Rewinds the DIR handle so the directory entries may be read again.

`rmdir(`*path*`)`

Remove the directory at *path*.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`S_ISBLK(`*mode*`)`

Tests whether the specified *mode* value—as returned by, e.g., `stat` or `readdir`—represents a block device.

Returns `true` or `false`.

`S_ISCHR(`*mode*`)`

Tests whether the specified *mode* value—as returned by, e.g., `stat` or `readdir`—represents a character device.

Returns `true` or `false`.

`S_ISDIR(`*mode*`)`

Tests whether the specified *mode* value—as returned by, e.g., `stat` or `readdir`—represents a directory.

Returns `true` or `false`.

`S_ISFIFO(`*mode*`)`

Tests whether the specified *mode* value—as returned by, e.g., `stat` or `readdir`—represents a FIFO or pipe.

Returns `true` or `false`.

`S_ISREG(`*mode*`)`

Tests whether the specified *mode* value—as returned by, e.g., `stat` or `readdir`—represents a regular file.

Returns `true` or `false`.

`S_ISLNK(`*mode*`)`

Tests whether the specified *mode* value—as returned by, e.g., `stat` or `readdir`—represents a symbolic link.

Returns `true` or `false`.

`S_ISSOCK(`*mode*`)`

Tests whether the specified *mode* value—as returned by, e.g., `stat` or `readdir`—represents a socket.

Returns `true` or `false`.

`setegid(`*gid*`)`

Set the effective process GID to *gid*. *gid* must be an integer or symbolic group name.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`seteuid(`*uid*`)`

Set the effective process UID to *uid*. *uid* must be an integer or symbolic user name.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`setenv(`*name*`, `*value*`[, `*overwrite*`])`

Sets the environment variable *name* to *value*. If the variable already exists then it is not changed unless *overwrite* is `true`. *overwrite* defaults to *true*.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

This function is thread-safe on Solaris, NetBSD, and Linux. But see note at `getenv`. FreeBSD and OpenBSD are confirmed to be not thread-safe. The status of AIX and OS X is unknown.

`setgid(`*gid*`)`

Set the real process GID to *gid*. *gid* must be an integer or symbolic group name.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`setsid()`

Create a new session and process group.

Returns the new process group ID on success, otherwise `nil`, an error string, and an integer system error.

`setuid(`*uid*`)`

Set the real process UID to *uid*. *uid* must be an integer or symbolic user name.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`sigfillset([`*set*`])`

Returns a sigset_t userdata object with all bits filled. If *set* is specified should be an existing sigset_t userdata object to reuse.

`sigemptyset([`*set*`])`

Returns a sigset_t userdata object with all bits cleared. If *set* is specified should be an existing sigset_t userdata object to reuse.

`sigaddset(`*set*`[, `*signo* ...`])`

Returns a sigset_t userdata object with the specified signals set. If *set* is not a sigset_t object, a new, empty sigset_t is instantiated and initialized according to whether *set* is `nil`, an integer signal number, an array of integer signal numbers, or the string "*" (filled) or "" (empty). If specified, *signo* and additional arguments should be integer signal numbers to be added to the sigset_t object.

`sigdelset(`*set*`[, `*signo* ...`])`

Like `sigaddset`, but *signo* and subsequent integer signal numbers are cleared from the sigset_t object.

`sigismember(`*set*`, `*signo*`)`

Returns `true` if *signo* is a member of sigset_t *set*, otherwise false.

`sigprocmask([`*how*`, `*set*`[, `*oset*`]])`

If *how* and *set* are defined, sets the signal mask of the current process or thread. *how* should be one of `SIG_BLOCK`, `SIG_UNBLOCK`, or `SIG_SETMASK`. *set* should be a sigset_t userdata object, or a number, string, or array suitable for initializing a sigset_t object as discussed in `sigaddset`.

Returns the old mask as a sigset_t userdata object on success, otherwise `nil`, an error string, and an integer system error. *oset* is an optional sigset_t userdata object to be reused as the return value, and is first cleared before passing to the system call.

Whether the process or thread mask is set is implementation defined, and varies across platforms. Threaded applications should use `pthread_sigmask`*, which is guaranteed to set the mask of the current thread.*[5] *Unfortunately, there is no interface which is guaranteed to only set the process mask. New threads inherit the mask of the creating thread, so standard practice is typically to block everything in the main thread while creating new threads.*

[5] Use of `pthread_sigmask` requires linking with –lpthread on some platforms and for this reason is presently not supported by `lunix`.

`sigtimedwait(`*set*`[, `*timeout*`])`

Atomically clears any pending signal specified in *set* from the pending set of the process *and* thread. If none are pending, waits for *timeout* seconds, or indefinitely if *timeout* is not specified. Fractional seconds are supported.

On success returns an integer signal number cleared from the pending set and an array representing the members of the siginfo_t structure (without the "si_" prefix).[6] On error returns `nil`, an error string, and an integer system error. If *timeout* is specified and no signal was cleared before the timeout, the system error will be `ETIMEDOUT`.

OS X and OpenBSD lack a native `sigtimedwait` implementation. On OS X `lunix` uses `sigpending` and `sigwait` to emulate the behavior. However, in a multi–threaded application if another thread clears a signal between `sigpending` and `sigwait` then `sigwait` could block indefinitely. There's no way to solve this race condition.[7] On OpenBSD `sigwait` is only available through libpthread, but on OpenBSD libpthread must be loaded at process load–time and cannot be brought in as a `dlopen` run–time dependency. Therefore an alternative emulation is used which clears the pending signal by installing a noop signal handler. This is not thread-safe if another thread is also installing a signal handler simultaneously. Threaded applications on these platforms should be mindful of these limitations. The `cqueues` project supports thread-safe signal listening with `kqueue` on both OpenBSD and Mac OS X.

`symlink(`*path*`1, `*path*`2)`

Creates a new directory entry at *path*2 as a symbolic link to *path*1.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`timegm(`*tm*`)`

tm is a table of the form returned by the Lua routine `os.date("*t")`. This allows converting a datetime in GMT directly to a POSIX timestamp without having to change the process timezone, which is inherently non-thread-safe.

Returns a POSIX timestamp as a Lua number.

`truncate(`*file*`[, `*size*`])`

Truncate *file* to *size* bytes (defaults to 0). *file* should be a string path, or `FILE` handle or integer file descriptor.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`tzset()`

Initializes datetime conversion information according to the TZ environment variable, if available.

Return `true`.

[6] Currently only the .si_signo member is copied from siginfo_t.

[7] One possible solution is to explicitly `raise` the signal before calling `sigpending`, but this solutions relies on untested assumptions about signal handling on these platforms.

`umask([`*cmask*`])`

If *cmask* is specified, sets the process file creation mask and returns the previous mask as a Lua number.

If *cmask* is not specified, queries the process umask in a thread-safe manner and returns the mask as a Lua number.

`uname([...])`

If no arguments are given, on success returns a table with the following fields

.sysname
: Name of the current system as a string.

.nodename
: Name of this node within an implementation-defined communications network as a string.

.release
: Release name of the operating system as a string.

.version
: Version of the operating system as a string.

.machine
: Hardware description of the system as a string.

If additional arguments are given, on success each field specified (as named above) is returned as part of the return value list.

On failure returns `nil`, an error string, and an integer system error.

`unlink(`*path*`)`

Deletes the file entry at *path*.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

`unsetenv(`*name*`)`

Deletes the environment variable *name* from the environment table.

Returns `true` on success, otherwise `false`, an error string, and an integer system error.

This function is thread-safe on Solaris, NetBSD, and Linux. But see note at `getenv`*. Also see note at* `setenv`*.*

4.1.2 `unix.dir`

The `unix.dir` module implements the prototype for DIR handles, as returned by `unix.opendir`.

`dir:files([`*field* ...`])`

Returns an iterator over `unix.readdir(`...`)`.

`dir:read([`*field* ...`])`

Identical to `unix.readdir`.

`dir:rewind()`

Identical to `unix.rewinddir`.

`dir:close()`

Identical to `unix.closedir`.