Towards a Lua scripted operating system

Justin Cormack @justincormack

https://github.com/justincormack

Whats this all about?

- We all use operating systems all the time
- They have become big and hard to understand
- Everything written in C, you just see lots of config files

When we see a large amount of C with config files gone mad, we replace lots of it with Lua!

That sounds a bit hard?

What does the interface to the (Linux, Unix) operating system look like?

- System calls: open(), read(), socket(), mmap() and hundreds more!
- File systems: /proc, /sys, /dev
- Miscellaneous interfaces: ioctl(), fcntl(). There are a few thousand of these...
- Kernel sockets: netlink

API, ABI, architecture

Linux (and NetBSD) has a fixed ABI guarantee, unlike POSIX that defines a C API (headers etc).

So target kernel ABI directly, everything self contained in Lua, not compiled C. Partly as started with LuaJIT ffi interface. Some complexity with eg the details of 64 bit files on 32 bit machines, and so on. Means it works on Android even though Android C library does not provide bindings to much kernel functionality. Very fast on LuaJIT.

Lua API design

- Tried to make it easy to use and uniform
- There are vast numbers of constants, namespace them into tables, access as strings
- There are large numbers of types, C structs which are like tables.
- Error handling is uniform (success, error return)

Lets look at some examples...

Lua API example

```
S = require "syscall"

fd, err = S.open("filename", "rdwr,creat", "rwxu")
```

In C terms that is `O_RDWR|O_CREAT` and `S_IRWXU`, in Lua terms looks up in a table `S.c.O["rdwr,creat"]` that has a metatable that understands strings and bitwise OR, plus other things eg `"0777"` for octal file permissions.

Lua API choices

Discussion on list this week about format

```
S.setrlimit(S.c.RLIMIT.CPU, S.t.rlimit{rlim_cur = 4, rlim_max = 4}) -- valid, direct translation
S.setrlimit("cpu", {cur = 4, max = 4}) -- simplest option close to C
process.limits.soft.cpu = 4 -- not valid yet, could be layered on top
```

More API examples

```
s = S.socket("unix", "seqpacket, nonblock")

str = s:read(20) -- methods of fd

assert(S.mount("none", tmpfile, "tmpfs", "rdonly, noatime"))

addr = S.t.in_addr("127.0.0.1") -- types

assert(S.mknod("/dev/null", "fchr,0666", S.t.device(1, 3))) -- make
character device
```

More complex API examples

```
i = S.nl.interfaces()
print(i.wlan0.inet6[1].addr)
2001:370:1f0A:1862:4913:23f2:6532:124a
print(i.eth0.mtu)
1500
```

Can create interfaces, routes and so on too (if root), can also listen for changes to network state.

Iterators

Provide iterators where this matches underlying interfaces

```
fd = S.open("/dev", "directory, rdonly")
for d in fd:getdents() do
  print(d.name)
end
```

This means can iterate over very large directories without creating a

Types

- Types have metatables to make them easier to use
- The underlying struct can be initaliased and treated as a table

```
sa = S.t.sockaddr_in{port = 80, addr = "any"}
print(sa.sin_addr) -- the actual C field
0.0.0.0
print(sa.sin_port, sa.port)
20480       80
```

Layered API

- Only part that interfaces with C is raw syscall layer, which just deals with numbers, pointers, and hidden, modified parameters eg 32 vs 64 bit files
- Then (Lua) syscall layer converts eg tables to structs, deals with constants, may return multiple values, deal with non standard interfaces such as ioctl and so on
- Then an additional layer may add more complex functionality eg netlink, utilities, even more abstract interfaces. Only partially done.

What is provided

- Provides what the OS provides directly: sockets, files, memory.
- Does not provide DNS lookup or anything built on sockets (no http, SSL etc). This is is inconvenient for many applications, so aim to provide (largely 100% Lua) versions by using the Openresty Lua libraries.
- However does provide lots of really useful stuff, like file change notifications (inotify), network configuration, security features that other libraries usually do not provide.

Current major issues

- Documentation; saying its like C not sufficient
- Making interfaces even easier to use
- Portability. Used to require LuaJIT ffi, now working with luaffi, working towards full portability soon.
- Other OS support: Linux and NetBSD good now, OSX poor, ... FreeBSD, Cygwin, OSv, ... Linux MIPS, ...
- More commonality with luasocket, LuaPosix and other overlapping libraries. As a community not very good at coordinating.

Testing

We like tests!

- Comprehensive set of tests for pretty much all systems calls
- Very useful when refactoring code
- Have found one bug in NetBSD so far, possibly one in Linux
- Tests error conditions as well as normal returns
- Also test expected size and offsets of structures against C. Some difficulties finding out what the kernel uses.

Fun stuff

- NetBSD rump kernel support
- This is an build of NetBSD for userspace, can run under Linux, or in other environments eg native under Xen
- Userspace filesystem, socket and device drivers. Everything except memory allocation, processes which the host deals with.
- Boot a Lua program direct on Xen with no operating system but with the same syscall interface.
- Also useful for porting Posix code to embedded systems.

Fun stuff 2

- NetBSD Lua in kernel and core
- Lots of opportunity to use Lua in the operating system
- See Marc Balmers' talk after this.
- Lots of opportunity to use Lua, everywhere from installer to commands.

Come and join in!

Fun stuff 3

You could boot your computer just using Lua

- Almost all there now.
- Need a DHCP client.
- Have booted a minimal system in a Linux container with static IP.
- Quite possible to use on an embedded system instead of busybox.
- An application could configure its own environment, rather than relying on having a whole distribution, eg in Linux containers.

Why?

- Most of the OS code people write in C could perfectly well be written in Lua, it is not performance critical.
- The C libraries tend to make things much less easy to understand, lots of macros
- Things like netlink are very poorly documented, only one library that talks to it. Had to work out from scratch. Actually easier in Lua than in C.
- Surprisingly, few languages have a good interface to the parts outside Posix. Most people shell out, which has all sorts of issues and is messy

Code on GitHub