function presentation()
    print("Lua in the NetBSD Kernel")
end
Ideas for Users

Modifying software written in C is hard\textsuperscript{^w}impossible for users

Give the power to modify and extend the system to the user

Let the user explore the system
Ideas for Developers

Rapid Application Development approach to driver development
Modifying the system behaviour
Configuration of kernel subsystems
This was *NOT* my Goal:

Provide a language to write system software in
Considering Some Alternatives

Python
Java
But not Perl, Tcl, Javascript
Python

Not to difficult to integrate in C
Huge library
Memory consumption
Difficult object mapping
Java

Easy to integrate

Difficult object mapping

Memory considerations

Has been used for driver development
This caught my eye:

Builds in all platforms with an ANSI/ISO C compiler
Fits into 128K ROM, 64K RAM per interpreter state
Fastest in the realm of interpreted languages
Well-documented C/C++ API to extend applications
One of the fastest mechanisms for call-out to C
Incremental low-latency garbage collector
Sandboxing for restricted access to resources
Meta-mechanisms for language extensions,
  e.g. class-based object orientation and inheritance
Natural datatype can be integer, float or double
Supports closures and cooperative threads
Open source under the OSI-certified MIT license

1 Complete Lua SOC, practical applications in 256K ROM / 64K RAM

www.lua.org
Lua in NetBSD Userland

Library (liblua.so) and binaries (lua, luac) committed to -current

Will be part of NetBSD 6

No back port to NetBSD 5 stable
Lua in the NetBSD Kernel

Linux project „Lunatic“

GSoC 2010 project „Lunatic“

Research type of project

WORK IN PROGRESS!
Every process has its own address space
Lua states in different processes are isolated
Kernel

One address space

Every thread that „is in the kernel“ uses the same memory

Lua states are not isolated

⇒ Locking is an issue
A first look

# modload lua
# luactl create test_1
# luactl load test_1 ./hello.lua
# luactl destroy test_1
Implementation
Components

The lua(4) device driver (as module)
Lua States
Lua Modules
Lua Users
The lua(4) Device

ioctl(2) interface to userland
create, manage, destroy states
'require‘ modules to states
maintain a list of loaded modules
load and execute code
**Lua States**

Are always created „empty“
Can be assigned to subsystems
Are under control of lua(4)
Lua Modules

Are regular kernel modules
Have its own class: MODULE_CLASS_LUA
Register with lua(4) when loading
Can only be unloaded if not used
Lua Users

Kernel subsystems that use Lua
Create Lua states
Register themselves with lua(4)
The `luactl(8)` Userland Command

Used to control the `lua(4)` device via `ioctl(2)` calls

Create, destroy states

Load Lua code into states
'require' in the Kernel

require can be disabled
Check if a module already registered
If not, do a module autoload, if not prohibited
'require' Implementation

Check if a module already registered
If not, do a module autoload, if not prohibited, and try again

Naming scheme:
require 'xyz' → luaxyz.kmod
sysctl(8) Variables

kern.lua.require=1
kern.lua.autoload=1
kern.lua.maxcount=0
kern.lua.bytecode=0
Loading Lua Code

LUALOAD ioctl(2)
Path must contain ,/`
call lua_load()
Checks kern.lua.maxount
calls lua_pcall()
Kernel lua_Reader

uses the vn_open(9) functions:
vn_rdwr(UIO_READ, ...)

Security

No automatic code loading
module autoload in 'require' can be
turned off, as can 'require' itself

Execution count can be limited

Bytecode loading turned off by
default
Todos

MP-safeness
More bindings to standard kernel services
Implement pwdog(4) in Lua
Conclusions so far...

It works

C bindings can be substantial overhead

MP-safeness must be guaranteed

Still no real driver written in Lua
Lua in FreeBSD (not yet...)

Userland parts can be considered done

Interest from the team
Future Work

split compiler/interpreter?
gpio, watchdog, PCI
tty line disciplines
In god we trust, in C we code!

Marc Balmer
marc@msys.ch, m@x.org,
mbalmer@NetBSD.org