

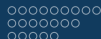


Lua in the NetBSD Kernel

Marc Balmer <mbalmer@NetBSD.org>

www.Mcours.com

Site N°1 des Cours et Exercices Email: contact@mcours.com



Topics

① The Programming Language Lua

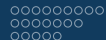
- The Lua Interpreter
- Syntax and such
- Modules

② Embedding Lua in C Programs

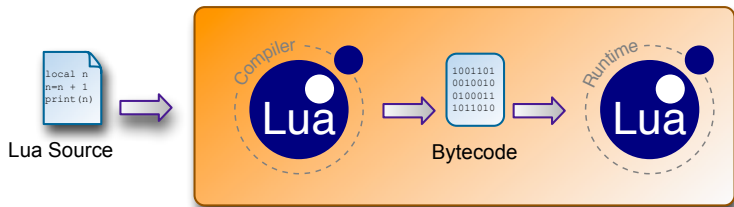
- State Manipulation
- Calling C from Lua
- Calling Lua from C

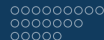
③ Lua in the NetBSD Kernel

- Use Cases
- Implementation Overview
- Implementation Details

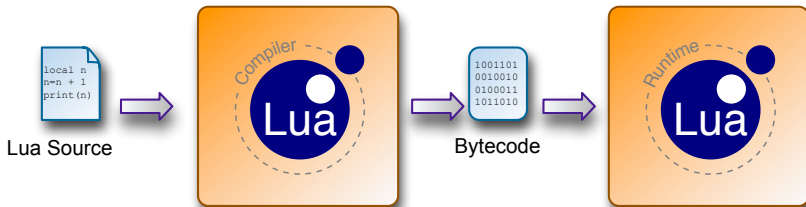


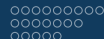
Running Lua Sourcecode





Compiling / Running Lua Bytecode





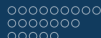
Running from C

- `int luaL_dofile(lua_State *L, const char *filename)`
- `int luaL_dostring(lua_State *L, const char *str)`



Values, Variables, and, Data Types

- Variables have no type
- Values do have a type
- Functions are first-class values



Tables

- Tables are THE data structure in Lua
- Nice constructor syntax
- Tables make Lua a good DDL
- Metatables can be associated with every object



Lua Table Constructor

Create and initialize a table, access a field:

```
mytable = {  
    name = 'Marc',  
    surname = 'Balmer',  
    email = 'm@x.org'  
}  
  
print(mytable.email)
```

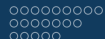
Extending Lua Programs

Access GPIO pins from Lua:

```
require 'gpio'  
  
g = gpio.open('/dev/gpio0')  
g:write(4, gpio.PIN_HIGH)  
g:close()
```

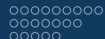
www.Mcours.com

Site N°1 des Cours et Exercices Email: contact@mcours.com



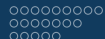
Creating and Destroying a State

- `lua_State *L = lua_newstate()`
- `luaopen_module(L)`
- `lua_close(L)`



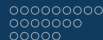
Calling a C Function

- Function has been registered in `luaopen_module()`
- `int function(lua_State *L)`
- Parameters popped from the stack
- Return values pushed on the stack
- Return Nr. of return values



Calling a Lua Function

- Find the function and make sure is **is** a function
- Push parameters on the stack
- Use `lua_call(lua_State *L, int index)`
- or `lua_pcall(lua_State *L, int index)`
- Pop return values from the stack



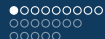
Calling a Lua Function

The Lua function

```
function hello()  
    print('Hello, world!')  
end
```

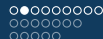
Call hello from C:

```
lua_getglobal(L, "hallo");  
lua_pcall(L, 0, 0, 0);
```



Ideas for Users

- Modifying software written in C is hard for users
- Give users the power to modify and extend the system
- Let users explore the system in an easy way



Ideas for Developers

- „Rapid Application Development” approach to driver development
- Modifying the system behaviour
- Configuration of kernel subsystems



Alternatives

- Python
- Java



Python

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



Java

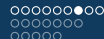
- Easy to integrate
- Difficult object mapping
- Memory considerations
- Has been used for driver development



Lua in NetBSD Userland

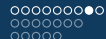
- Library (liblua.so) and binaries (lua, luac) committed to -current
- Will be part of NetBSD 6
- No back port to NetBSD 5 stable planned





Lua in the NetBSD Kernel

- Started as GSoC project, porting Lunatik from Linux to NetBSD
- Proof that the Lua VM can run in the kernel, lack of infrastructure
- Infrastructure beeing added now



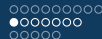
Running in Userland

- Every process has its own address space
- Lua states in different processes are isolated

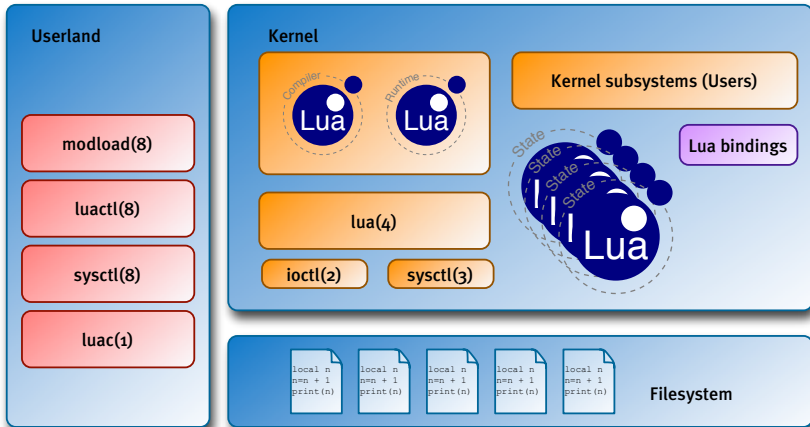


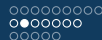
Running in the Kernel

- One address space
- Every thread that „is in the kernel“ uses the same memory
- Locking is an issue

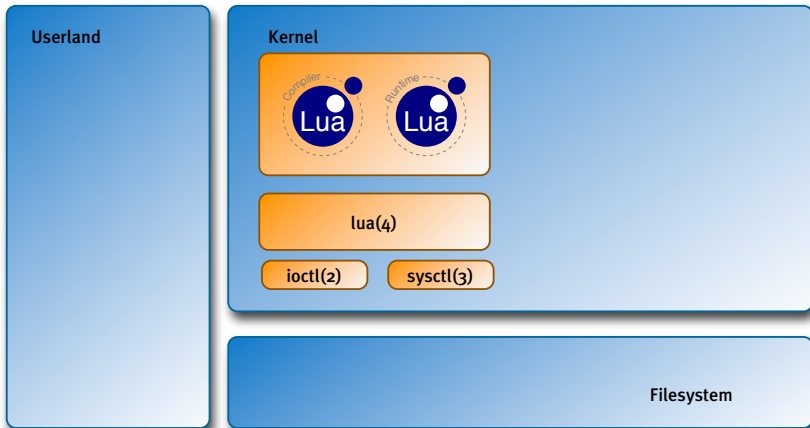


The Big Picture



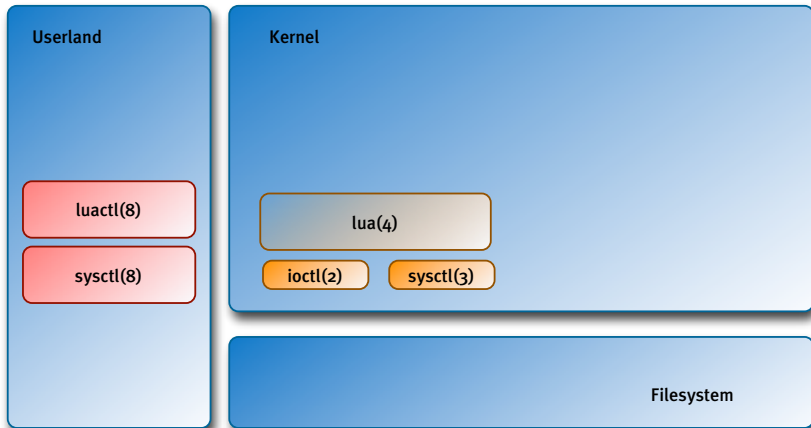


The lua(4) Device Driver





lua(4) ioctl(2) / sysctl(3) Interface

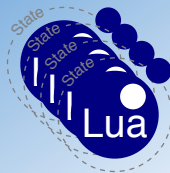




Lua States

Userland

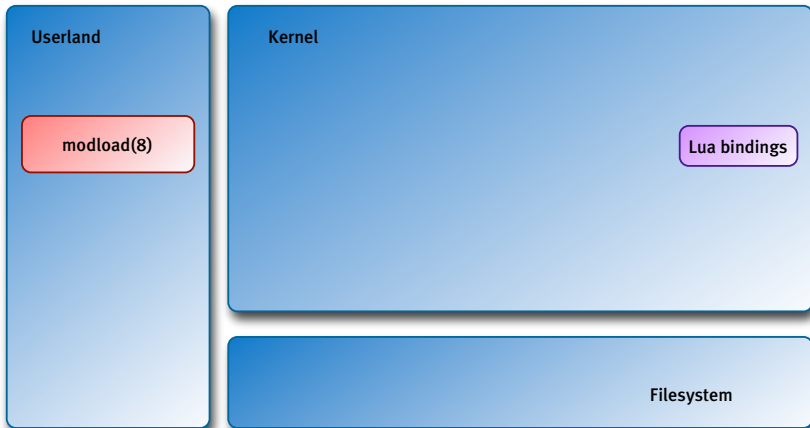
Kernel



Filesystem

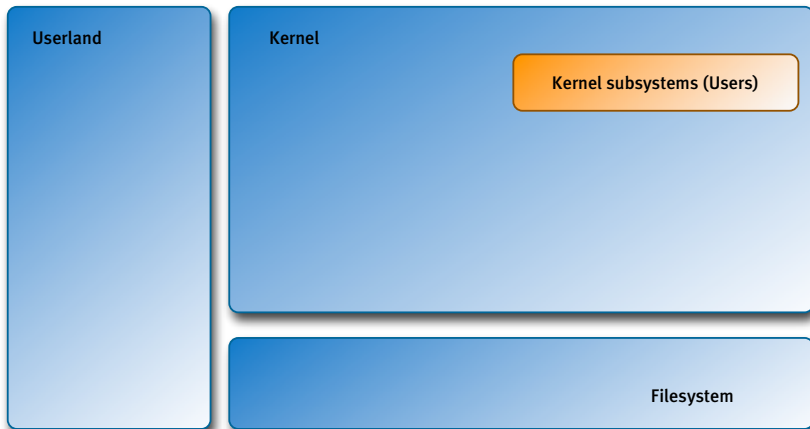


Lua Modules



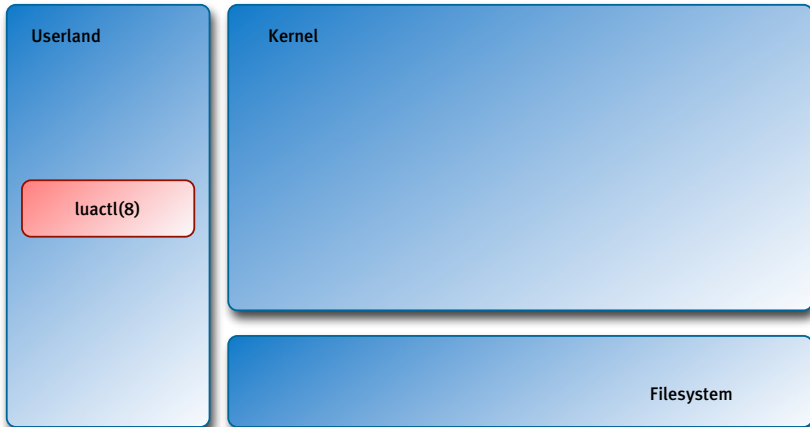


Lua Users





The luactl(8) Command





„require”

- Modules are kernel modules
- 'require' can be turned off
- Modules must be assigned to Lua states by hand then
- By default 'require' is on and modules are even autoloaded
- Module autoloading can be turned off



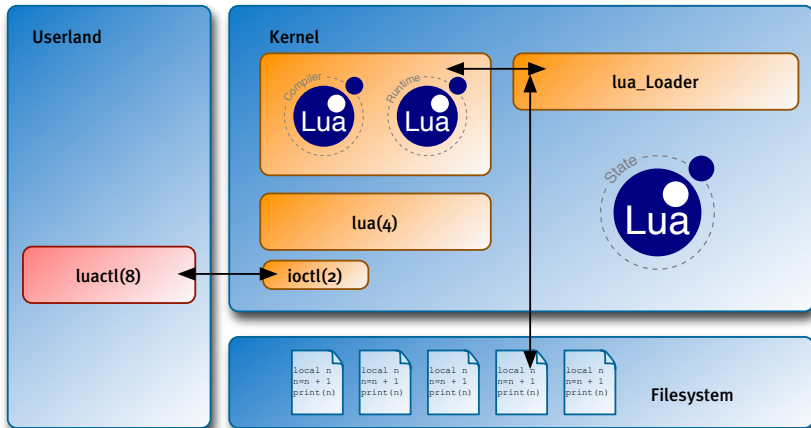


sysctl(8) Variables

- kern.lua.require=1
- kern.lua.autoload=1
- kern.lua.bytecode=0
- kern.lua.maxcount=0



Loading Lua Code





Security

- New Lua states are created empty
- Full control over the loading of code
- No access to kernel memory, -functions but through predefined bindings

www.Mcours.com

Site N°1 des Cours et Exercices Email: contact@mcours.com

Time for Questions

