

Porting LUA for Nautilus

Goutham Kannan
Illinois Institute Technology
gkannan1@hawk.iit.com

Imran Ali Usmani
Illinois Institute Technology
iusmani@hawk.iit.com

Sunny Changediya
Illinois Institute Technology
schangedi@hawk.iit.com

ABSTRACT

Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description. In this paper, we describe in detail the process involved in porting it for Nautilus. Nautilus being an AeroKernel OS has a very thin kernel layer and contains no user space. The lack of user space makes running user space application directly on nautilus impossible at run-time. This need can be satisfied by allowing the users to run application as interpreted scripts such as Lua scripts.

Keywords

Lua, Nautilus, Porting.

1. INTRODUCTION

The ability to run applications in user space is one of the basic functionality of a general Operating System. We use Nautilus as our base operating system. Since Nautilus does not have any user space, the executable binaries cannot be executed by loading them dynamically during run-time. Currently the user-level applications are compiled along with the Operating System. The main idea is to avoid the applications being compiled every time along with OS. We have come up with an idea of providing support for interpreted language, using which the user applications or commands can be executed like scripts. This allows to run easy-to-write user friendly lua scripts and commands at run-time rather than writing complex C applications at compile time.

We chose to port Lua among all interpreted languages due to its smaller code base of about 2000 lines of code. The lua footprint is also small of about 100kB compared to 824kB for heavy scripting languages like python. It can be build on any platform having a C compiler. The lua provides straight forward interface to languages like C and C++, i.e. you can embed C/C++ code in lua scripts or vice-versa. It has a

faster interpreter and use less memory. To further improve the performance, critical portions of the code can be written in C. It has a nice, simple and powerful syntax, which is user-friendly. The lua community is friendly and enthusiastic community and a good amount of documentation is available. The code base is simple and stable, which can be easily tweaked or modified if needed.

After embedding lua into the nautilus kernel, we expect to invoke nautilus functions at run-time using lua interpreter commands. We may be able to invoke lua scripts, whenever nautilus has a file-system in it. But currently we can still send lua commands line-by-line if we access nautilus command prompt from a remote system using a RS-232 cable through UART port.

2. THE PROPOSED SOLUTION

The nautilus source code can be found at <https://bitbucket.org/kchale/nautilus>. In-order to run nautilus we require QEMU Emulator. We source code for lua 5.2.3 can be downloaded from <https://www.lua.org/ftp/>. The lua source code has been added to a nautilus source tree. The lua headers are also placed in the includes folder of nautilus. Existing Makefile is extended to compile lua source code. The existing libc dependency headers are removed from lua source. Next, upon compiling of nautilus, undefined references to symbols and functions show up. The declarations of all the undefined function references are added to .h files (E.g. libccompat.h) and their definition are added to corresponding .c files (i.e. libccompat.c). This process is repeated until all the compilation errors are exhausted.

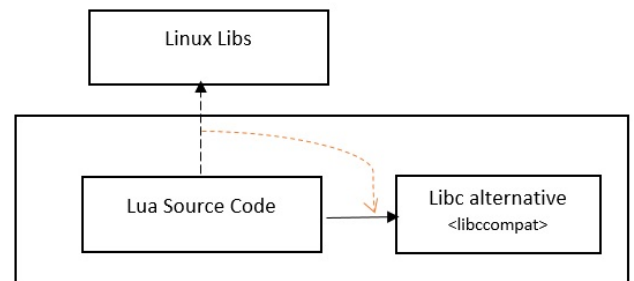


Figure 1: Porting Lua

Lua's entry function(i.e. main()) is called from the main method of arch/x86/init.c. The first issue that blocked us

from reaching the Lua interpreter prompt was the absence of `realloc()` function definition. We were able to trace the code flow and find cause of the PANIC by instrumenting the code(using `printk`),this PANIC was resolved by adding a new definition for `realloc()`^[1]. Lua stack objects were created using `realloc()` memory allocation.

The next main hurdle was making jumps to interpreter entry point. The function pointer to the desired function was called in a protected mode using the function `Lua _rawrun-protected()`. This function internally was using the jumps viz. `setjmp` and `longjmp`. The absence of assembly implementation of jumps made the system to panic. The jumps were implemented for nautilus as an assembly code.

We faced complex problems handling macro implementation of Lua. In Lua macros were implemented to perform operations such as storing state in Table, pushing values to stack, and calling function `inturn`. Tracing back complex macros and their implementation was toughest thing to work on.

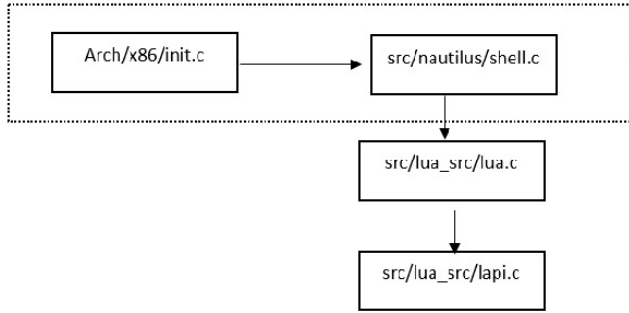


Figure 2: Control Flow

The actual Lua implementation of the prompt was handled by `fwrite()` to display the prompt banner as well as the results and by `fgets()` for receiving the input from the user. These functionalities were achieved using the existing nautilus's `nk_vc_print()` and `nk_vc_gets()`.

The interpreter was unable to understand integer, float, double inputs. We traced the cause of this issue to `strtod()` function from `libc`. We added implementation of `strtod`^[2] to make lua interpreter understand numbers. After this lua was able to perform operations on numbers but to print it over the screen it wasn't able to convert those numbers to string format. The lua internally was using `sprintf` from `libc` but since we removed all `libc` dependencies, it was redirected to nautilus defined `sprintf` function. this function was not efficient enough to convert float and doubles to strings and it was returning garbage values. so we used `snprintf` function of nautilus. this had option of mentioning of string bounds to prevent buffer overflow. But the current implementation of `snprintf` in nautilus does not support float and double conversion, so we limit format specifier to integers only.

The other issue was executing subroutines in multiple lines. the issue was because of the value returned by `nk_vc_gets` (Error) on accepting newline character. We were able to resolve this and now able to define subroutines using multiple lines.

3. CURRENT STATUS

When nautilus boots up, it loads lua interpreter by de-

fault. The user can execute lua specific commands as well as subroutines. We have tested operators, iterators, loops, decision making, functions, strings, integers, arrays, and built-in routine such as reverse, substring, upper, lower etc functionalities. The interpreter cannot run lua scripts since nautilus does not have a file system. As mentioned before the current lua interpreter also does not supports float and double operations. After testing, we found that modulo operator is not working.

4. EXPERIMENTS

Type	Operations	result
String	<code>string.reverse</code>	pass
	<code>string.sub</code>	pass
	<code>string.format</code>	pass
	<code>string.len</code>	pass
	<code>print</code>	pass
	<code>string.upper</code>	pass
	<code>string.lower</code>	pass
	<code>string.char</code> <code>string.find</code>	pass Fail
Operators	<code>+</code>	pass
	<code>-</code>	pass
	<code>*</code>	pass
	<code>/</code>	pass
	<code>=</code>	pass
	<code>%</code> <code>(<, >, ==, >=, <=)</code>	fail pass
Subroutine (user created)	<code>max(a,b)</code>	pass
	<code>palindrome(string)</code>	pass
	<code>factorial(b)</code>	pass
	<code>pow(a,b)</code>	pass
Conditional	<code>if-else</code>	pass
	<code>while</code>	pass
Data structure	Array	pass
	Iterators	pass
Data Types	Integer	pass
	Float	Fail
	Double	Fail

Figure 3: Experiments performed

We have performed Lua supported string operations such as (1) `string.reverse`, (2) `string.sub`, (3) `string.format`, (4) `string.len`, (5) `print`, (6) `string.upper`, (7) `string.lower`, (8) `string.char`.

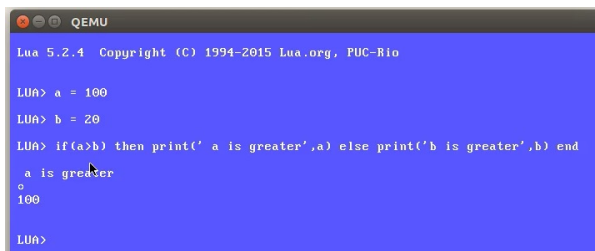
We have performed integer operations such as (1) addition, (2) subtraction, (3) multiplication, (4) division, (5) operators (`>`, `<`, `>=`, `<=`, `==`, `=`, `%`).

We created sample lua subroutines and performed operations such as (1) `max(a,b)`, (2) `palindrome(string)`, (3) `factorial(a)`, (4) `pow(a,b)`

We have also performed operations using `while`, `if-else`, arrays, and iterators.

5. IMPROVEMENTS

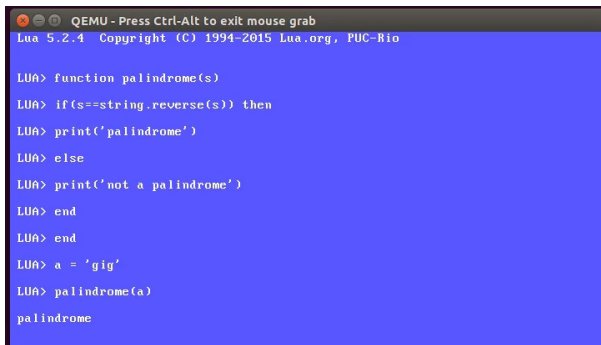
We can add support for float and double arithmetic operations. In this implementation of lua, we are unable to perform modulo operations. We can improve it to support modulo operator using C's `modulo`.



```
QEMU
Lua 5.2.4 Copyright (C) 1994-2015 Lua.org, PUC-Rio

Lua> a = 100
Lua> b = 20
Lua> if(a>b) then print('a is greater',a) else print('b is greater',b) end
a is greater
100
Lua>
```

Figure 4: Integer Operation



```
QEMU - Press Ctrl-Alt to exit mouse grab
Lua 5.2.4 Copyright (C) 1994-2015 Lua.org, PUC-Rio

Lua> function palindrome(s)
Lua> if(s==string.reverse(s)) then
Lua> print('palindrome')
Lua> else
Lua> print('not a palindrome')
Lua> end
Lua> end
Lua> end
Lua> a = 'gig'
Lua> palindrome(a)
palindrome
```

Figure 5: String Operations

6. SCOPE

1. The importance of interpreter based language is that you can give commands on the fly, and you don't need to compile them with the kernel code before boot up, even if there is no user space available. If we are able to hook up nautilus functions to lua interpreter commands, we can invoke nautilus functions at run time from lua command prompt.

2. We can access the lua command prompt from a remote system through UART port using RS232 cable. We can run lua scripts over the nautilus-lua by sending script commands line by line from a remote system using UART port. So we can write a perl or python script which can read a lua script line-by-line and send it over to a particular UART port which sends commands to Nautilus lua prompt.

3. if there is a file system on nautilus, lua scripts can also be invoked from the command prompt of nautilus. Then we can compare linux vs nautilus throughput for executing lua code.

7. ACKNOWLEDGEMENT

We thank Prof.Kyle Hale for contributing to the libc equivalent implementations of realloc(), setjmp(), and longjmp() and his guidance throughout the project. This work is a contribution towards course work CS595-03 Runtime design of Operating systems for supercomputing. This work is carried out under the guidance of Prof.Kyle Hale.

8. CONCLUSION

We were able to compile Lua in Nautilus. We removed libc dependencies, but still we were able to port Lua successfully. We have performed string, number arithmetic, and conditional constructs operations and most of them were



```
QEMU - Press Ctrl-Alt to exit mouse grab
Lua> function max(n,m)
Lua> if(n>m) then
Lua> print(n)
Lua> else
Lua> print(m)
Lua> end
Lua> end
Lua> max(4,2)
4
Lua> max('c','m')
m
```

Figure 6: Subroutine Operations

successful. We still need to add support for double, float, and modulo operators.

9. REFERENCES

- [1]Nautilus Patch1,Patch for Setjmp and Longjmp, Available: <https://bitbucket.org/kchale/nautilus>.
- [2] Opensource.apple.com,'Implementation of strtod', Available: <https://opensource.apple.com/source/Libc/Libc-167/string.subproj/strtod.c>, Accessed:15-Nov-2016.