Running Application Specific Kernel Code by a Just-In-Time Compiler

Ake Koomsin       Yasushi Shinjo
Department of Computer Science
University of Tsukuba
Agenda

• Motivation & Objective
• Approach
• Evaluation
• Related work
Motivation and Objective
Standard services provided by a typical kernel may not be enough for some applications
Scriptable Operating Systems with Lua

- Lourival V. Neto el at. (DLS ’14)
- Allow privileged users to extend an OS with the Lua script
- However
  - System-wide + Privileged users
  - Performance

This figure is from the paper
Objective

Make the existing OS provides specialized kernel services in an application scope
Approach
Lua & LuaJIT

- Lua: A scripting language
  - Portable & Embeddable
  - Wireshark, Redis, MySQL Proxy
  - Game engines

- LuaJIT: Interpreter and Just-In-Time (JIT) compiler
  - Better performance
  - Tracing JIT compiler
General idea

- Port LuaJIT as the scripting engine into a kernel module

  - FreeBSD 10.1 x86

  - x86 only for now due to LuaJIT’s internals

- What can we extend with application specific kernel scripting?
System call scripting

- Create specialized system calls on the basis of the existing ones
- Propose two system calls
  - `int register_lua_syscall(char *script, size_t sz)`
  - `int lua_syscall(int lua_fd, unsigned int nargs, int arg1, ..., int arg8)`
function run(sfd, buf_addr, ...) 

local nrecv_array = {} 
for nreq = 1, max_batch do 
-- 1024 is the size of buffer 
-- 16 is the size of struct sockaddr 
-- 4 is the size of socklen_t 
local offset = nreq - 1 
local buf_offset = offset * 1024 
local sockaddr_offset = offset * 28 
local sockaddr_len_offset = offset * 4 
local byte_recv = recvfrom( 
    sfd, 
    buf_addr + buf_offset, 
    1024, 0, 
    sockaddr_addr + sockaddr_offset, 
    sockaddr_len_addr + sockaddr_len_offset) 
-- condition check, omitted 

end 

return #nrecv_array 

end
System call scripting

- Potential combinations
  - Multiple `accept()`
  - Multiple `recv*(())` and `send*(())`
  - `select() / kevent()` follow by `recv*(())`
Safe execution of Lua scripts

• Lua scripts are not able to
  • Access the kernel memory directly
  • Call any kernel function directly
• Bindings are required
Safe execution of Lua scripts
Safe execution of Lua scripts

• Resource exhaustion
  
  • Memory
    
    • Modify the memory allocation to check for a certain limit
  
  • Infinite loop
    
    • ???
Safe execution of Lua scripts

How to stop an infinite loop?

while true do
  end

function f()
  f()
end
Safe execution of Lua scripts

• Potential solutions to recover an infinite loop after a timeout
  • Jump guard
  • Preempt the running kernel thread and change its saved instruction pointer register
  • Patch all assembly code with nop and a jump to an exit point
Safe execution of Lua scripts

1. for \( i = 1, n \) do end
   always terminate

2. while (cond) do end
   depending on cond

repeat

3. ... until (cond)
   depending on cond
Safe execution of Lua scripts

1. \[\text{for } i = 1, n \text{ do end} \quad \text{always terminate}\]

2. \[\text{while (cond) do end} \quad \text{depending on cond}\]

3. \[\text{repeat} \quad \text{depending on cond}\]

   \[\ldots\]

4. \[\text{until (cond)} \]
Safe execution of Lua scripts

• Resource exhaustion

• Memory
  • Modify the memory allocation to check for a certain limit

• Infinite loop
  • Only for loop is allowed + callout(9)
Evaluation
Experimental setup

- Target: Memcached
- Scenario [Facebook 2013]
  - GET requests over UDP
  - SET requests over TCP
- We modified Memcached to support multiple GET request processing over UDP
Experimental setup

Client: Memaslap

- Intel i7 3850, 12 GB
- Ubuntu 14.04 x86-64
- 4 worker threads
- 128 concurrencies
- 9:1 GET:SET ratio

Server: Memcached

- Intel i7 950, 32 GB
- FreeBSD 10.1 x86
- 1 worker thread
- 2 GB max object size
- (Ours) 16 max batch

Link: 1 Gbps
- Average response time
- Transactions per second
- Only GET requests
Average response time
(Server: 1 thread, Client: 4 threads 128 concurrencies)

Up to 30% with interpreter, 33% with JIT
Average transactions per second  
(Server: 1 thread, Client: 4 threads 128 concurrencies)

Up to 37% with interpreter, 44% with JIT
Is it difficult to use?

• If you are familiar with system programming, it should not be a problem

• Pointer arithmetics⚠️

```lua
function run(sfd, buf_addr, ...) 
  ... 
  local buf_offset = offset * 1024 
  local sockaddr_offset = offset * 28 
  local sockaddr_len_offset = offset * 4 
  local byte_recv = recvfrom(...) 
  ... 
end
```
Is it difficult to use?

- If you are familiar with system programming, it should not be a problem
  - Pointer arithmetics⚠️
- It will affect the way an application is written
Related work
Related work

- System Call Clustering [Rajagopalan 2003]
- Linux’s Syslets [Molnar 2007]
- FlexSC [Soares 2010]
- SPIN Operating System [Bershad 1995]
- Scriptable Operating Systems with Lua [Neto 2014]
Conclusion

• Application specific kernel code using the Lua language with the LuaJIT compiler
  • System call scripting

• Evaluation on Memcached
  • 33% reduction in average response time
  • 44% increase in transactions per second
  • For small size response over UDP, one CPU core
Future work

• Application specific network stack

• Incorporating with FreeBSD Jails virtual network stack
Q & A