lume

Lua Mutitasking Environment

jvisca@fing.edu.uy - grupo mina / fing / udelar - 2015

Motivation

Writing Lua applications...

- Used for fast prototyping of networked applications...
  - ...extended to control / sensor interaction
- Must be highly portable
- Running on embedded / low power hardware

Applications naturally grow up to need concurrence

- Performance: blocking on I/O is bad
- Some tasks are naturally concurrent: serializing them is a loss/loss

Concurrence

Preemptive

- Controlled by OS: Transparent for the applications
- Implemented at the hardware level
- Example: UNIX, *

Cooperative

- Controlled by apps: Explicit in the source code
- Implemented in user space
- Example: RISC OS, Windows 3.1

Concurrence

Message Passing

- “Processes”
- Examples: UNIX pipes, web services
- API model:
  serialization (e.g. JSON) +
  messaging (e.g. Sockets)
- Pros: robust, easily distributable
- Cons: serialization&messaging overhead, hard to design

Shared State

- “Threads”
- Examples: pthreads, [*obligatorios del curso de redes*]
- API model:
  access control (e.g. semaphors)
- Pros: no data copying
- Cons: high contact surface, error prone, hard to track bugs (when preemptive)

Sierra Wireless luasched

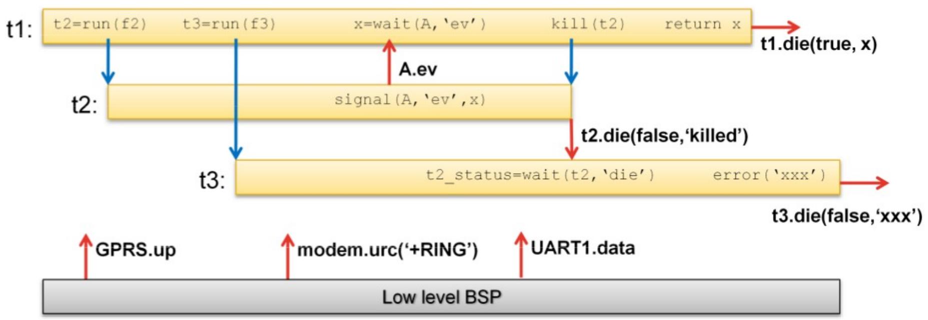
Scheduling

- Create or kill a task: `t=sched.run(function()…end)`, `sched.kill(t)`
- Send a signal: `sched.signal(emitter, event, [args…])`
  - Used for low level I/O
  - For inter-tasks synchronization
  - For extensibility
- Wait for a signal: `sched.wait(emitter, [timeout], events…)`
- React to signals: `sched.sighook()`, `sched.sigrun()`, `sched.sigrunonce()`

Written in pure Lua, with current state accessible as proc.tasks (never needed to convert to C).

Sierra Wireless Proprietary and Confidential January 21, 2010 Page 20

Scheduling



Sierra Wireless Proprietary and Confidential January 21, 2010 Page 21

SIERRA WIRELESS

http://www.lua.org/wshop11/m2m-embedded-development-with-lua.pdf

Luasched vs Lumen

- Sierra Wireless
- Enterprise grade M2M development and tooling
- Modified Lua + modified luasocket
- Tooling modules in C (shell, LTN12)

- jvisca@fing.edu.uy
- “API changes and other random breakages occur”
- Plain lua + luasocket & nixio backends
- All tooling in pure Lua+lumen sched.

Lumen scheduler

- Inspired in Sierra Wireless luasched.
- Cooperative
- Message passing and shared memory supported
    - Messaging trough signals and pipes/streams
    - Access control trough critical section marking
- Pure Lua, no external dependencies: runs on Lua 5.1, 5.2, 5.3 LuaJIT
- Implemented using coroutines
- Runs in a single VM instance (state), available as a Lua module
  ```
  local sched = require 'sched'
  ```
- MIT licence

Lumen vs Others

- Pros
  - Lightweight
  - Pure Lua: Hardware & OS-independent
  - Simple usage model
  - Well suited for I/O bound logic
  - Tools available: logging, remote shell, webserver...
- Cons
  - Single Lua state – single core.
    (Multiple States can interchange messages, tough)
  - Slow context switching (compared to preemptive), not well suited to CPU-bound logic

Lumen scheduler

```lua
local sched=require 'sched'

-- task emits signals
sched.run(function()
    for i = 1, 10 do
        sched.signal('an_event', i)
        sched.sleep(1)
    end
end)

-- task receives signals
sched.run(function()
    local waitd = {'an_event'}
    while true do
        local ev, data = sched.wait(waitd)
        print (data)
    end
end)

sched.go()
```

Lumen tasks

- Create and run a task

```lua
local f = function(...)
   -- blabla
end
local taskd = sched.run( f, ... )
```

- Create a task, run later

```lua
local taskd = sched.new_task( f )
taskd:run(...) --start
```

- Current task

```lua
local my_taskd = sched.running_task
```

Tasks control

- Pause tasks

```
taskd:set_pause(true)  --pause
taskd:set_pause(false) --unpause
```

- Yielding and waiting

```
taskd:wait()
taskd:sleep(1.5)
```

- Finishing a task == finishing the function

```
local f = function(...)
   error ...  --here
   return ... --here
end --here
```

- Killing a task

```
taskd:kill()
```

Lumen signals - emitting

- Tasks emit signals
- A signals is a *event*, plus *parameters*
  - The event can be of any type
  - Parameters in any number, of any type
- Tasks should emit signals for everything that could be interesting to other tasks
- Examples:

```
sched.signal( 'new client', skt )
local EVENT_DIE = {}
sched.signal(EVENT_DIE, true, 10, 10)
sched.signal(EVENT_DIE, false, 'div by 0')
```

Lumen signals - receiving

- Tasks can wait for signals. Waiting for a signal is the main method for synchronization.
- Wait descriptors define a list of event(s) (can be '*')
- The wait call returns the signal's parameters
- Examples:

```
local waitd = {
    EVENT_DIE,
    EVENT_FINISH,
    timeout = 10 --optional
}
local event, p1, p2, p3 = sched.wait(waitd)
```

Lumen signals - receiving

- There are some useful shortcuts for handling signals in a event-like manner

```
sched.sigrun(waitd, f)
sched.new_sigrun_task(waitd, f)
sched.sigrunonce(waitd, f)
sched.new_sigrunonce_task(waitd, f)
```

- They are implemented in plain wait calls

```
sched.new_sigrun_task = function ( waitd, f )
   return M.new_task (function()
      while true do
         f(M.wait(waitd))
      end
   end)
end
```

Lumen signals - buffering

- Signals are like UDP: if nobody is waiting for them, they get lost.

```
while true do
    sched.wait(waitd1)
    --do something
    sched.wait(waitd2) --events for waitd1 can get missed
    --do something
end
```

- Sometimes this is the right thing. When not:

```
local waitd = sched.new_waitd({
    '*',
    buff_mode='drop_last', --or 'drop_first'
})
```

- Still, this is not enough sometimes (faster emitter than receiver).

Signal alternatives: pipes & streams

Signals never block the emitter. Not useful on a producer – consumer scheme.

Pipes

```lua
local pipes=require 'pipes'
local p=pipes.new(10)

--one task
p:write(a, b)

--other task
local ok,a,b=p:read()
```

Streams

```lua
local stream=require 'stream'
local s=stream.new(1000)

--one task
s:write('####')

--other task
local text=s:read()
```

- Implemented using plain signals
- Pipes and streams allow timeouts

Resource sharing and access

- A catalog allows to publish objects under a well known name, and wait for them to appear.

```
local catalogs = require 'catalog'
local tasks = catalogs.get_catalog('tasks')

--in one task
tasks:register('consumer', taskd)

--in another task
local consumer = tasks:waitfor('consumer', 60)
```

- Some ideas for catalogs are 'events', 'pipes', 'devices'

Mutexes

Needed only seldom, because scheduling is non-preemptive: only needed if task yields control explicitly inside critical section.

```
local mutex = require 'mutex'

local n = 0

local critical = function ()
    print( 'before', n )
    n = n + 1
    sched.wait()
    n = n - 1
    print( 'after', n )
end

local mx = mutex.new()
local synched = mx:synchronized(critical)
```

Other method:

```
local function critical()
  mx:acquire()
  -- do stuff
  mx:release()
end
```

I/O: selector module

- Basic POSIX support
  - Sockets (TCP&UDP)
  - Files
  - Piping-in output from programs
- Data can be obtained in several ways, depending on its nature
  - Signals on data arrival
  - Callback to a provided function on data arrival
  - Stream
- Sending can be synchronous or asynchronous.
- Nixio and luasocket as backends

```
local selector = require 'tasks/selector'
selector.init({service='nixio'})
```

I/O: selector module

- Send a UDP Packet

```
local udpsend = selector.new_udp("127.0.0.1", 8888)
udpsend:send('data!')
```

- Start a TCP Server, a handler read lines

```
local tcp_server = selector.new_tcp_server("127.0.0.1", 8888, 'line',,
function(sktd, data, err)
    print ('arrived:', sktd, data)
    return true
end)
```

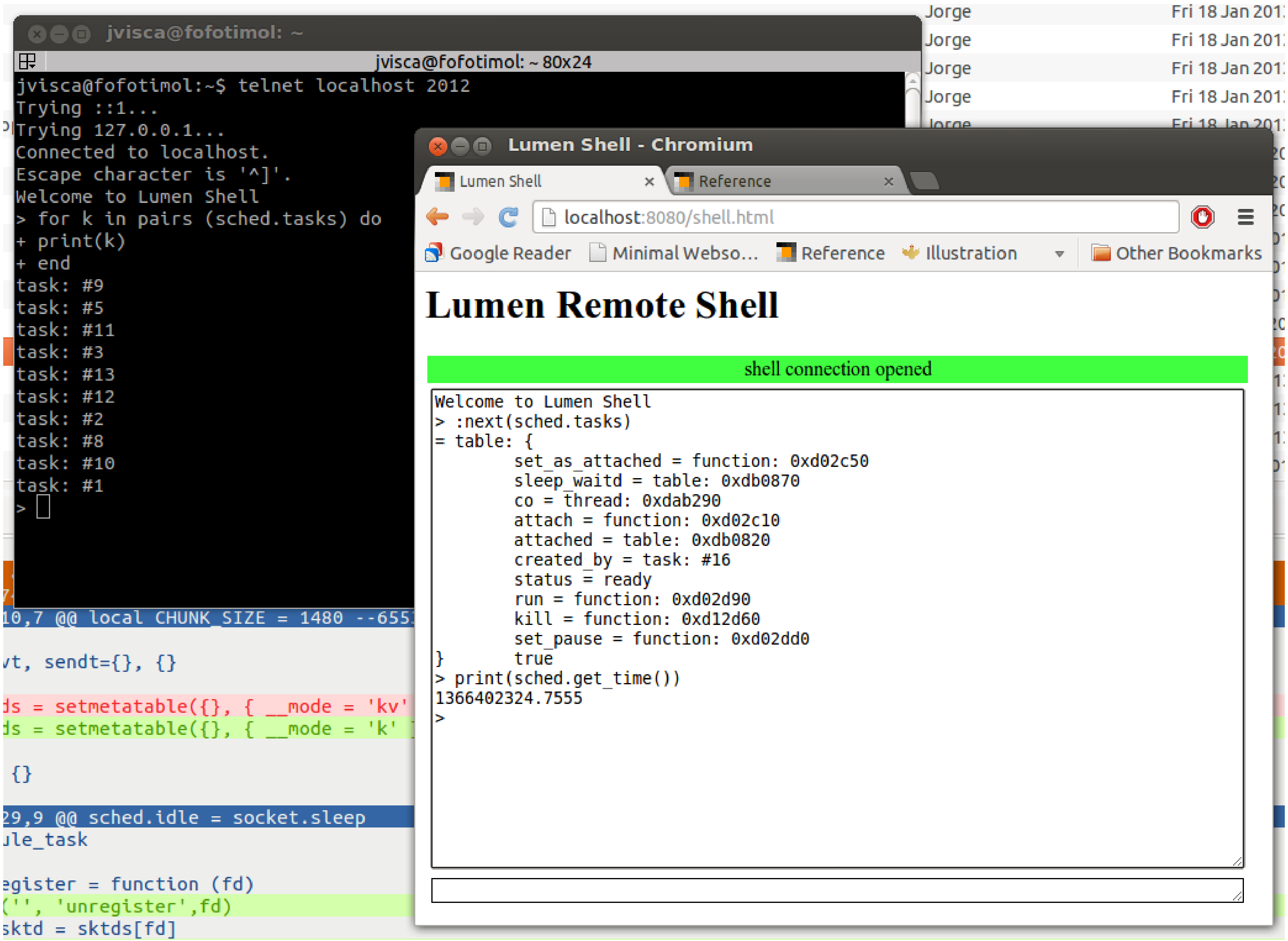
- Receive UDP trough signals

```
local udprecv = selector.new_udp(nil, nil, "127.0.0.1", 8888)
sched.sigrun(
    {udprecv.events.data, timeout=60},
    function(_, _, ...) print("arrived:", ...) end
)
```

- Read from file trough a stream

```
local fs = stream.new(buffer_size)
local fd, err = selector.new_fd ('/dev/tty0', {"rdonly"}, nil, fs)
while true do print( fs:read() ) end
```

Tooling



jvisca@fofotimol: ~
jvisca@fofotimol: ~ 80x24
jvisca@fofotimol:~$ telnet localhost 2012
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Welcome to Lumen Shell
> for k in pairs (sched.tasks) do
+ print(k)
+ end
task: #9
task: #5
task: #11
task: #3
task: #13
task: #12
task: #2
task: #8
task: #10
task: #1
>
Lumen Shell - Chromium
Lumen Shell
Reference
localhost:8080/shell.html
Google Reader
Minimal Webso...
Reference
Illustration
Other Bookmarks
Lumen Remote Shell
shell connection opened
Welcome to Lumen Shell
> :next(sched.tasks)
= table: {
set_as_attached = function: 0xd02c50
sleep_waitd = table: 0xdb0870
co = thread: 0xdab290
attach = function: 0xd02c10
attached = table: 0xdb0820
created_by = task: #16
status = ready
run = function: 0xd02d90
kill = function: 0xd12d60
set_pause = function: 0xd02dd0
}   true
> print(sched.get_time())
1366402324.7555
>
Jorge
Fri 18 Jan 201

References

- Lumen sources:
  https://github.com/xopxe/Lumen
- Lumen API:
  http://xopxe.github.com/Lumen/

End