LibOS as a regression test framework for Linux networking

Hajime Tazaki

2016/02/12

netdev 1.2
Outline

- libOS introduction
- testing framework introduction
- case studies
- QA
what is LibOS?

- Library version of Linux kernel
- presented at netdev0.1, proposed to LKML (2015)

http://www.slideshare.net/hajimetazaki/library-operating-system-for-linux-netdev01
media

- LWN
  - https://lwn.net/Articles/637658/
- Phoronix
- Linux Magazine
- Hacker News
  - https://news.ycombinator.com/item?id=9259292
how to use it?

- Network Stack in Userspace (NUSE)
  - LD_PRELOADed application
  - Network stack personality
- Direct Code Execution (DCE, ns-3 network simulator)
  - Network simulation integration (running Linux network stack on ns-3)
what is NOT LibOS?

- *not only* a userspace operating system
- *not only* a debugging tool

- but LibOS *is*
  - a library which can link with *any programs*
  - a library to form any purpose of program
anykernel

- introduced by a NetBSD hacker (rump kernel)

- Definition:

  We define an anykernel to be an organization of kernel code which allows the kernel's unmodified drivers to be run in various configurations such as application libraries and microkernel style servers, and also as part of a monolithic kernel. -- Kantee 2012.

- can form various kernel for various platforms

- userspace (POSIXy), bare-metal, qemu/kvm, Xen
  - Unikernel?
single purpose operating system

- Strip downed software stack
- single purpose
- resource efficient with speed
- boot within TCP 3-way handshake

[1]: Madhavapeddy et al., Jitsu: Just-In-Time Summoning of Unikernels, USENIX NSDI 2015
Demos with Linux kernel library

Unikernel on Linux (ping6 command embedded kernel library)

Unikernel on qemu-arm (hello world)
what's different?

- User Mode Linux
  - generate executable of Linux kernel in userspace
  - no shared library
- Containers
  - no foreign OS (shared kernel with host)
- nfsim
  - broader coverage of kernel code
recent news

- Linux kernel library (LKL) is coming
  - by Octavian Purdila (Intel)
  - since 2007, reborn in late 2015

*LibOS project is going to migrate to LKL project*

- port NUSE code to LKL already
- DCE (ns-3 integration) not yet
- unikernel in progress
testing network stack
motivation

- testing networking code is hard
  - complex cabling
  - inefficiency with massive VM instances
- You may do
  - in your own large testbed
  - with your test programs
are we enough?

- frequently changing codebase
  - many commits (30~40 commits/day)
  - out of 982K LoC (`cloc net/`)
  - may have increased num of regression bugs

- the number of commit per day
your test

- easy to create in your laptop with VM (UML/Docker/Xen/KVM)
- only IF the test is enough to describe
your test (cont'd)

- huge resources to conduct a test
- not likely to **reproduce**
- tons of **configuration scripts**
- running on different machines/OSes
  - controlling is troublesome
  - distributed debugger...
many terminal windows with gdb
other projects

- Test suites/projects
  - LTP (Linux test project, https://linux-test-project.github.io/)
  - kselftest (https://kselftest.wiki.kernel.org/)
  - autotest (http://autotest.github.io/)
  - kernelci (https://kernelci.org/)
  - NetDEF CI (quagga)

- those are great but networking is always hard
  - controlling remote hosts is (sometimes) painful
  - combination of userspace programs are unlimited
  - timing is not deterministic, across distributed networks
why LibOS ?

• **single process model** with multiple nodes
  ▪ ease of debug/test/development
• **deterministic behavior** (by ns-3 network simulator)
• rich network configuration by **ns-3 network simulator**
• ease of testing by automation (on public CI server)
public CI server (circleci.com)

- test per commit (push)
- test *before* commit
- easily detect regressions
architecture

1. Virtualization Core Layer
   - deterministic clock of simulator
   - stack/heap management
   - isolation via dlmopen(3)
   - single process model

2. Kernel layer
   - reimplemention of API
   - glue/stub codes for kernel code
   - use as-is

3. POSIX glue layer
   - reimplemention of POSIX API
   - hijack host system calls
How?

- a single scenario script (C++, sorry) to describe all
  - application, network stack (kernel as a lib), traffic, link, topology, randomness, timing, etc

1. Recompile your code
   - Userspace as Position Independent Executable (PIE)
   - Kernel space code as shared library (libsim-linux.so)

2. Run with ns-3
   - Load the executables (binary, library) in an isolated environment among nodes
   - synchronize simulation clocks with apps/kernels clock
features

- app supports
  - routing protocols (Quagga)
  - configuration utilities (iproute2)
  - traffic generator (iperf/ping/ping6)
  - others (bind9, unbound, dig)

- protocol supports
  - IPv4/ARP/IPv6/ND
  - TCP/UDP/DCCP/SCTP/(mptcp)
  - L2TP/GRE/IP6IP6/FOU
what's *not* useful

- performance study of the computation
  - deterministic clock assumes *unlimited* computation/storage resources
  - e.g., you can define 100Tbps link without any packet loss
test suite list

- verify results
  - socket (raw\{6\}, tcp\{6\}, udp\{6\}, dccp\{6\}, sctp\{6\})
  - encapsulation (lt2p, ip6ip6, ip6gre, fou)
  - quagga (rip, ripng, ospfv\{2,3\}, bgp4, radvd)
  - mptcp
  - netlink
  - mip6 (cmip6, nemo)

- simple execution
  - iperf
  - thttpd
  - mptcp+iperf handoff
  - tcp cc algo. comparison
  - ccnd
bugs detected by DCE (so far)

- having nightly tested with the latest net-next (since Apr. 2013~4yrs)

- [net-next,v2] ipv6: Do not iterate over all interfaces when finding source address on specific interface. (v4.2-rc0, during VRF)

- [v3] ipv6: Fix protocol resubmission (v4.1-rc7, expanded from v4 stack)

- [net-next] ipv6: Check RTF_LOCAL on rt->rt6i_flags instead of rt->dst.flags (v4.1-rc1, during v6 improvement)

- [net-next] xfrm6: Fix a offset value for network header in _decode_session6 (v3.19-rc7?, regression only in mip6)
Use Case
network simulator in a nutshell

- (mainly research purpose)
- flexible parameter configurations
- usually in a single process
  - can be extended distributed/parallel processes for speedup
- usually with abstracted protocol implementation
  - but no abstraction this time (thanks to LibOS)
- always produce same results (**deterministic**)
  - can inject pseudo-randomness
  - not realistic sometimes
  - but useful for the test (always reproducible)
workflow

1. (installation of DCE)

   ```make testbin -C tools/testing/libos```

2. develop a model (of interests)
   
   - (you already have: the Linux network stack)

3. write a simulation scenario
   
   - write a network topology
   - parameters configuration (randomization seed, link, traffic, applications)

4. test it
   
   - one-shot (locally)
   - nightly, per-commit, per-push, etc
simulation scenario

```c
int main(int argc, char **argv)
{
    // create nodes
    NodeContainer nodes;
    nodes.Create (100);

    // configure DCE with Linux network stack
    DceManagerHelper dce;
    dce.SetNetworkStack ("ns3::LinuxSocketFdFactory",
                        "Library", StringValue ("libsim-linux-4.4.0.so"));
    dce.Install (nodes);

    // run an executable at 1.0 second on node 0
    DceApplicationHelper process;
    ApplicationContainer apps;
    process.SetBinary ("your-great-server");
    apps = process.Install (nodes.Get (0));
    apps.Start (Seconds (1.0));

    Simulator.Stop (Seconds(1000.0))
    Simulator.Run ();
}
```
API (of DCE helpers)

- userspace app
  - ns3::DceApplicationHelper class
- kernel configuration
  - sysctl with LinuxStackHelper::SysctlSet() method
- printk/log
  - generated into files-X directory (where X stands for the node number)
  - syslog/stdout/stderr tracked per process (files-X/var/log/{PID}/)
- an instant command (ip)
  - LinuxStackHelper::RunIp()
- manual
test it!

- use `waf` for a build the script

```
cd tools/testing/libos/buildtop/source/ns-3-dce/
./waf
```

- run the script with `test.py` to generate XUnit test results

```
./test.py -s exapmle -r
```

- run the script with valgrind

```
./test.py -s exapmle -g
```

- a wrapper in Makefile

```
make test ARCH=lib ADD_PARAM=" -s example"
```

*(the directories may be changed during upstream (etc), sorry 'bout that)*
case study: encapsulation test

ns-3-dce/test/addons/dce-linux-ip6-test.cc

- unit tests for encapsulation protocols
  - ip6gre, ip6-in-ip6, l2tp, fou
  - with iproute2, ping6, libsim-linux.so (libos)

- full script
encap protocols tests

1) tunnel configurations

```
LinuxStackHelper::RunIp (nodes.Get (0), Seconds (0.5),
    "-6 tunnel add tun1 remote 2001:db8:0:1::2 
     "local 2001:db8:0:1::1 dev sim0");
LinuxStackHelper::RunIp (nodes.Get (1), Seconds (0.5),
    "-6 tunnel add tun1 remote 2001:db8:0:1::1 
     "local 2001:db8:0:1::2 dev sim0");
```

2) set up ping6 command to generate probe packet

```
dce.SetBinary ("ping6");
dce.AddArgument ("2001:db8:0:5::1");
apps = dce.Install (nodes.Get (1));
apps.Start (Seconds (10.0));
```

3) verify if the encap/decap work fine or not

```
if (found && icmp6hdr.Get_Type () == Icmpv6Header::ICMPV6_ECHO_REPLY) {
    m_pingStatus = true;
}
```
That's it. Test Test Test !
XUnit test result generation

- `make test ARCH=lib ADD_PARAM=" -s linux-ip6-test -r"` gives you a test result retained

```xml
% head testpy-output/2016-02-08-09-49-32-CUT/dce-linux-ip6.xml
<Test>
  <Name>dce-linux-ip6</Name>
  <Result>PASS</Result>
  <Time real="3.050" user="2.030" system="0.770"/>
</Test>

<Test>
  <Name>Check that process &plain; completes correctly.</Name>
  <Result>PASS</Result>
  <Time real="0.800" user="0.370" system="0.310"/>
</Test>

<Test>
  <Name>Check that process &ip6gre; completes correctly.</Name>
  <Result>PASS</Result>
  <Time real="0.600" user="0.460" system="0.100"/>
</Test>
```
**git bisect**

you can now *bisect* a bug with a single program!

- prepare a bisect.sh

```bash
#!/bin/sh

# Merge and compile:
git merge origin/nuse --no-commit
make clean ARCH=lib
make library ARCH=lib OPT=no
make test ARCH=lib ADD_PARAM=" -s dce-umip"

RET= $? 
git reset --hard
exit $RET
```

- run it!

```bash
$ git bisect run ./bisect.sh
```
gcov (coverage measurement)

- coverage measurement across multiple nodes

```make
make library ARCH=lib COV=yes
make test ARCH=lib
```

(the COV=yes option does the job for you)
<table>
<thead>
<tr>
<th>Library</th>
<th>Coverage</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
<th>Functions</th>
<th>Max Complexity</th>
<th>Overall Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>process</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>main</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>net</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>net-bridge</td>
<td>0%</td>
<td>0%</td>
<td>N/A</td>
<td>0%</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>net-ipv4</td>
<td>96%</td>
<td>98%</td>
<td>43%</td>
<td>30%</td>
<td>28%</td>
<td>25%</td>
</tr>
<tr>
<td>net-ipv4-addr</td>
<td>95%</td>
<td>95%</td>
<td>45%</td>
<td>37%</td>
<td>24%</td>
<td>24%</td>
</tr>
<tr>
<td>net-key</td>
<td>100%</td>
<td>100%</td>
<td>3%</td>
<td>1%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>net-gtp</td>
<td>100%</td>
<td>100%</td>
<td>35%</td>
<td>36%</td>
<td>20%</td>
<td>20%</td>
</tr>
<tr>
<td>net-ftp</td>
<td>88%</td>
<td>68%</td>
<td>23%</td>
<td>17%</td>
<td>10%</td>
<td>10%</td>
</tr>
<tr>
<td>net-filter</td>
<td>98%</td>
<td>98%</td>
<td>82%</td>
<td>81%</td>
<td>78%</td>
<td>78%</td>
</tr>
<tr>
<td>net-filter-lpdp</td>
<td>33%</td>
<td>33%</td>
<td>4%</td>
<td>4%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>net-ftp</td>
<td>100%</td>
<td>100%</td>
<td>92%</td>
<td>90%</td>
<td>86%</td>
<td>86%</td>
</tr>
<tr>
<td>net-ftp</td>
<td>100%</td>
<td>100%</td>
<td>92%</td>
<td>90%</td>
<td>86%</td>
<td>86%</td>
</tr>
</tbody>
</table>
gdb (debugger)

- Inspect codes during experiments
  - among distributed nodes
  - in a single process
- perform a simulation to reproduce a bug
- see how badly handling a packets in Linux kernel

http://yans.pl.sophia.inria.fr/trac/DCE/wiki/GdbDce
Memory error detection among distributed nodes in a single process

Use Valgrind

http://yans.pl.sophia.inria.fr/trac/DCE/wiki/Valgrind
Summary

- walk through review of testing framework with LibOS + DCE
- uniqueness of experiment with the library (LibOS)
  - multiple (host) instances in a single process
  - flexible network configurations
  - deterministic scheduler (i.e., bugs are always reproducible)
future directions

- merging to LKL (Linux Kernel Library)
  - part of LibOS has done
- continuous testing to net-next branch
  - I'm watching at you (don't get me wrong.. :))
resources

- Web
  - [https://www.nsnam.org/overview/projects/direct-code-execution/](https://www.nsnam.org/overview/projects/direct-code-execution/) (DCE specific)
- Github
  - [https://github.com/libos-nuse/net-next-nuse](https://github.com/libos-nuse/net-next-nuse)
- LKL (Linux Kernel Library)
  - [https://github.com/lkl/linux](https://github.com/lkl/linux)