Ktap

A New Scripting Dynamic Tracing Tool For Linux

zhangwei(Jovi) jovi.zhangwei@huawei.com

LinuxCon Japan in May 2013

Self Introduction: Jovi Zhang

- Senior Software Engineer in Huawei company
- Start hacking on Linux kernel since 2010
- Interest in system tracing and high level language design
- Long term experience in embedded Linux development

Contents

- Introduction
- Architecture
- Basic language elements
- Examples
- Project status and future work

ktap

Quote from README:

KTAP is a new scripting dynamic tracing tool for Linux,
it is designed to give operational insights that allow users
to tune, troubleshoot and extend kernel and application.

KTAP also is designed for enabling great interoperability
with Linux kernel, it gives user the power to modify and
extend the system, and let users explore the system in an
easy way.

Current dynamic tracing language

- Dtrace in Solaris
  - Innovation tool in Solaris, widely used
  - Compile to bytecode, restricted functionality
  - Heavily based on SDT(statically defined tracing)
  - Ported to Linux: Dtrace in oracle Linux, dtrace4linux
  - GPL-incompatible CDDL license

- Systemtap
  - Borrows ideas from Dtrace
  - Translate to C, GCC compile it into kernel module
  - GPL license

More related work

- Java in kernel mode
  - Safe device driver model based on kernel-mode JVM
  - Short practical usage due to its limitations
- Lunatik/Luak
  - Ported to NetBSD kernel, no longer maintained
  - Not purpose for dynamic tracing
  - Mainly contains a diff-file, to patching the lua sources
  - Hard to maintain
- PacketScript
  - A Lua scripting engine for in-kernel packet processing

Why we create ktap?

There have so many projects aimed to enable dynamic scripting into Linux kernel,

But we still not found a practical tool/language to meet our requirements.

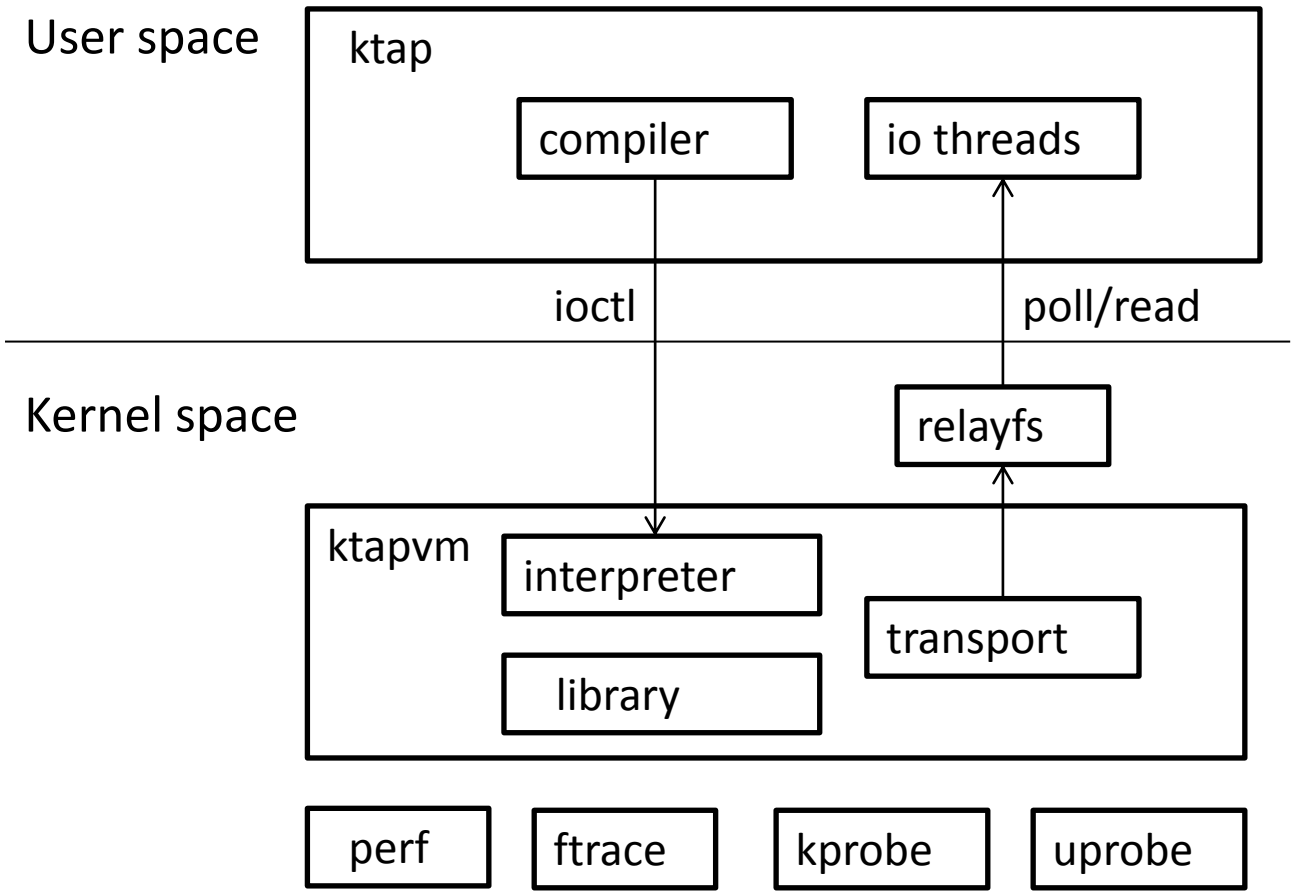
Motivation of ktap

- Lightweight Linux dynamic tracing
  - Kernel header file dependence issue
  - Big compiler tool chain(cross compile) issue
- Kernel script counterpart with great interoperability
  - Make kernel debugging more easy and safe
  - Give users the power to modify and extend the system
  - Let users explore the system in an easy way
- Leveraging current tracing infrastructure
  - statically tracepoints, kprobe and uprobe, perf events, ftrace

Design guideline

- bytecode based vm
- embedding and interoperability
- lightweight
- performance
- extendibility
- safety in sandbox

Architecture



Language: syntax

- Simple C like syntax
- Dynamic typed currently
- Shell style comments: “#”
- Structured control statements
  - if/while/for control flow block

See more in Ktap/doc/syntax.txt

Language: table

- Powerful data structure: associative array

  tbl = {}

  tbl[0] = 1

  tbl[-1] = “-1”

  tbl[“key”] = “value”

- Really useful for dynamic tracing

  tbl[execname()] = tbl[execname()] + 1

  count(tbl, execname())

  histogram(tbl)

Language: runtime library

- Built-in function
  - printf(…), cpu(), pid(), execname() …
- Kdebug
  - kdebug.probe(event_desc, func)
  - kdebug.probe_end(func)
- Timer
  - timer.s (N, func)
  - timer.ms (N, func), timer.us(N, func)

See more in Ktap/doc/library.txt

Language: event object mode

Event is abstracted as object

- e.name
- e.tostring()
- e.field(N)
- e.sc_nr, e.sc_is_enter
- e.sc_arg1, e.sc_arg2, … e.sc_arg6
- e.retval, e.set_retval(N)

See more in ktap/doc/ktap-tracing.txt

Example: enable all tracepoints

```
function eventfun (e) {
	printf("%d %d\t%s\t%s", cpu(), pid(), execname(), e.tostring())
}

kdebug.probe("tp:", eventfun)

kdebug.probe_end(function () {
	printf("probe end\n")
})
```

Example: enable all tracepoints

```
root# ktap scripts/tracepoints.kp
Press Control-C to stop.
1 285   kworker/1:2    softirq_entry: vec=1 [action=TIMER]
1 285   kworker/1:2    timer_cancel: timer=f4b521a4
1 285   kworker/1:2    timer_expire_entry: timer=f4b521a4 function=wakeup_readers now=962534
1 285   kworker/1:2    timer_expire_exit: timer=f4b521a4
1 285   kworker/1:2    softirq_exit: vec=1 [action=TIMER]
0 776   sshd    sys_rt_sigprocmask(how: 2, nset: bfdf915c, oset: 0, sigsetsize: 8)
0 776   sshd    sys_enter: NR 175 (2, bfdf915c, 0, 8, b7264ff4, bfdf92a8)
0 776   sshd    sys_rt_sigprocmask -> 0x0
0 776   sshd    sys_exit: NR 175 = 0
0 776   sshd    sys_enter: NR 3 (a, bfdf516c, 4000, b857e620, b6df268c, bfdf516c)
0 776   sshd    sys_exit: NR 3 = 4095
0 776   sshd    sys_enter: NR 4 (3, b858c348, 1030, 1030, bfdf91d8, b858c348)
0 776   sshd    irq_handler_entry: irq=19 name=eth0
…
```

Example: histogram style

```
# showing all tracepoints in histogram style
hist = {}
function eventfun (e) {
        count(hist, e.name)
}

kdebug.probe("tp:", eventfun)

kdebug.probe_end(function () {
       histogram(hist)
})
```

Example: histogram style

root# ktap scripts/tracepoints.kp

total enabled 956 events

Press Control-C to stop.

```
                value ------------- Distribution ------------- count
       rcu_utilization |@@@@@@                                 54960
      hrtimer_cancel |@@@                                      32739
           cpu_idle |@@                                       29519
       hrtimer_start |@@                                       28788
       sched_switch |@                                        16365
        softirq_raise |@                                       14003
         softirq_exit |@                                       13994
        softirq_entry |@                                       13983
   hrtimer_expire_exit |@                                      13729
  hrtimer_expire_entry |@                                      13725
             sys_exit |@                                       13030
            sys_enter |@                                       13030
                    …
```

Use Cases

Ktap can do many things, what you do with it is up to you.

- Dynamic tracing
  - Performance tuning, Like Systemtap, Dtrace does
- Packet processing
  - Inspired by PacketScript project
- Kernel subsystem extending
  - Use ktap to implement some logic
- Kernel testing
  - Unit testing: test drivers or BSP code more easily
- Others: Dynamic function patching; Easy kernel learning

Project Status

- Ktap released 0.1 in May 2013
- Support x86-32 and x86-64 (other arch is not tested yet)
- Support kernel 3.1 or later version
- Lines of codes: Ktapvm ~5K, userspace ktap ~4K
- Samples ready: ktap/scripts/
- Documentation ready: ktap/doc/
- Language basic testsuite ready: ktap/test/

Next Work

- Implement more tracing scripts
- FFI(foreign function interface)
  - Embedding and Interoperability
  - Operate on kernel data structure directly
  - Calling kernel function directly
- Performance optimization
  - Reducing overhead of ktap interpreter

The end

- ktapvm is a super hackable kernel module, come help
  - Under active development
  - Use ktap, give feedback

- Further information
  - Website: Not available yet
  - Code locations: https://github.com/ktap/ktap.git
  - Mailing list: ktap@freelists.org
  - Documentation: ktap/doc
  - Sample scripts: ktap/scripts