Debugging Kernel without Debugger

Masami Hiramatsu
<masami.hiramatsu.pt@hitachi.com>
Software Platform Research Dept.
Yokohama Research Lab.
Hitachi Ltd.
Who am I?

**Masami Hiramatsu**

- Researcher in Hitachi Yokohama Research Lab.
  - Responsible for Linux kernel development
  - Working on tracing/debugging/virtualization/embedded etc...

- Linux kernel developer
  - Kprobes maintainer
  - Perftools developer
Table of Contents

- Debugging Kernel
  - Printk
  - KDB
  - SystemTap
  - Preferred Features

- Perf Probe

- Ftrace with Dynamic Events
Debugging the Kernel

- What we want to know
  - Which points were passed?
  - What value the variables have?

- Kernel Features/tools
  - Printk
    - Easy to use. Everyone knows!
  - KDB
    - Local debugging with KDB
    - Remote debugging with KGDB via GDB
  - SystemTap
    - Scriptable tracing tool
  - Ftrace/Perftools
    - In-kernel tools
Easy to use, but...

- Requires rebuilding kernel
  - Hard to use on production system
  - Takes a time, especially on early stage of trouble shooting.
- Sometimes timing bug vanishes...
  - Especially on multi-thread racing bug

This should be the last resort

- If there is no other ways...
KDB

- **KDB is a great feature**
  - No need to use any tool :)
  - But, runtime kernel debugging is sometimes hard
    - Bugs depends on timing (e.g. device initialization)
  - KDB itself can’t use debuginfo
    - It’s KGDB – which is a debugger!
  - No online disassembler
    - It is hard to put breakpoint via local KDB
SystemTap

- An infrastructure to simplify the gathering of information about the running Linux system.

Pros

- Programmable dynamic event tracing
  - Dynamic == you can define event on every line of kernel source without rebuilding kernel :)

Cons

- Requires a kernel build environment :-(
- Depends on stable kernel module APIs
  - Sometimes it breaks on newer/developing kernel tree
- Script Language
  - Always we need to write a script for tracing even one event.
- Out of tree...
Preferred Features

- What features should debugging tools have?
- Analyze accurate behavior
  - Without rebuilding kernel
  - Without remote machine (use KVM!)
  - No additional language
  - In the mainline
  - (hopefully) no additional tools, just using kernel
Perf Probe

- **Perf-tools has a tracing feature**
  - Perf Probe provides dynamic printk like feature
    - Perf probe doesn’t do anything except what you want!

```
perf probe + perftools ~= SystemTap – (script language + compiler)
```

- And more feature

```
perf probe + perf script ~= SystemTap – (original script language + compiler) + perl or python
```
Easy Usage of Perf Probe

**Typical Usecase**

- 1) Investigate target code
  - List target functions with `-F`
  - Find the target line number with `-L`
  - Check which local variables can be accessed with `–V`
- 2) Define events
  - Add an event with `-a` (or just pass event definition)
- 3) Trace it
  - With `perf record`
- 4) Analyze it
  - With `perf report` or `perf script`
Perf Probe: Example

**Probe setting and tracing on vfs_read**

1) **Investigate target code**
   # perf probe -F
   # perf probe -L vfs_read
   # perf probe -V vfs_read:4

2) **Define events**
   # perf probe -a 'vfs_read:4 file file->f_mode'

3) **Trace it**
   # perf record -e probe:vfs_read -aRf ls -l

4) **Analyze it**
   # perf script

See tools/perf/Documentation/perf-probe.txt, for details.

Useful Links:
Recent Upstream Works

- Hardware branch tracer
- Offline Module Tracing
- Uprobes
Perf Probe can puts events on modules (drivers)
- It is also possible to put events on off-line modules
- Motivation: trace driver initialization
  - Most of driver troubles happen on initializing device

Usage

Online module:

```bash
# perf probe -m <MODULE_NAME> XXX_func
# perf record -e probe:XXX_func <do-something>
# perf report
```

Offline module:

```bash
# perf probe -m <PATH_TO_MODULE> XXX_initfunc
# perf record -e probe:XXX_initfunc modprobe XXX
# perf report
```
Module Tracing Example

**Tracing OpenvSwitch init function**

```bash
# perf probe -m /lib/modules/3.3.0-rc7+/kernel/net/openvswitch/openvswitch.ko -a "ovs_vport_init:7 dev_table"
Added new events:
  probe:ovs_vport_init (on ovs_vport_init:7 with dev_table)
  probe:ovs_vport_init_1 (on ovs_vport_init:7 with dev_table)
  probe:ovs_vport_init_2 (on ovs_vport_init:7 with dev_table)

You can now use it in all perf tools, such as:

```bash
  perf record -e probe:ovs_vport_init_2 -aR sleep 1
```

```bash
# perf record -e probe:ovs_¥* modprobe openvswitch
# perf script
...
  modprobe 5796 [000] 1863.444330: ovs_vport_init:
     (fffffffffa00d7734) dev_table=0
  modprobe 5796 [000] 1863.444335: ovs_vport_init_1:
     (fffffffffa00d773f) dev_table=ffff8801adf14000
```
Uprobes implementation is under going on tip-tree (perf/uprobes)

- Add new events on user program
- “-x” option for specifying program binary

```
# perf probe -x /lib64/libc.so.6 malloc
```

- Currently, you can only use symbols
- Debuginfo analyzer will be able to handle user programs too.
Debugging without Tools

- **Debugging methods: what do we need to know?**
  - 1) Which path did kernel execute?
  - 2) What value the variables had?
  - 3) From where the kernel reached?

- **How does the ftrace help us?**
  - 1) function-call graph and traceevent (static and dynamic)
  - 2) traceevent (static and dynamic)
  - 3) stacktrace
Easy Usage of Ftrace

- **Debugging a kernel on small system (e.g. lkmv, embedded)**
  - Your tools: kallsyms, disassembler, ftrace
    - Don’t be so sad, still you have great power tools in kernel!
  - 1) Mount debugfs
  - 2) Find related functions in /proc/kallsyms
  - 3) Disassemble the functions
  - 4) Put events on the functions
  - 5) Trace it.
Ftrace: Example

Probe setting and tracing on vfs_read

1) Mount debugfs
   # mount -t debugfs none /sys/kernel/debug

2) Find related functions
   # grep read /proc/kallsyms

3) Analyze Binary - without any tool!
   # cat vfs_read > /sys/kernel/debug/x86/disassembler
   # cat /sys/kernel/debug/x86/disassembler

4) Add event
   # echo 'p vfs_read+.. %di +0x3c(%di):u32' >> kprobe_events

5) Trace it!
   # echo 1 > events/kprobes/p_vfs_read../enable
   # cat trace
Features

- Support only most popular instructions
- Support rawcode/address dump feature
- Symbols are automatically solved (by kallsyms)
- No-MMX, No-SSE, No-AVX, etc.
  - Because these instructions rarely (0.01%?) used in kernel
  - It is easy to enhance it to support those, but who cares?
- AT&T-like syntax (not fully compatible with objdump)
  - Operand-size suffixes (d,q,dq etc.) are ignored.
  - Some instructions’ name isn’t correct. (e.g. cbw)
  - If you revert the AT&T-rize patch, it shows in Intel syntax.
In-kernel disassembler can show assembly code

This is only for x86 and fun!
You don’t need outer-tools except shell and kernel!
  • E.g. Debugging kernel on busybox is now possible :)

Interfaces

via debugfs: /sys/kernel/debug/x86/disassembler
  • Write a target kernel function name or address and read it.
via KDB: ‘dis’ command is now available
  • This helps you to put breakpoint on-line, a lot!
via panic message: kernel dumps assembly code
  • Instead of useless byte code, it shows assembly code.
via userspace tool: tools/bogodis
  • Just for testing.
Disassembler relays on x86 instruction decoder

- Based on same opcode map data
  - Use similar AWK script to generate instruction format table at build time

---

**Diagram:**

- **Git repository**
  - **X86-opcode-map.txt**
    - **Awk script**
      - **Instruction attribute table**
        - **Generating at build time**
          - **Instruction format table**
            - **Awk script**
              - **Instruction decoder**
                - **disassembler**
                  - **include**
                    - **Use**
Another function: Stacktrace

- Ftrace with dynamic event allows you to get stacktrace everywhere.

  - Just write “1” into tracing/options/stacktrace

```bash
# echo 1 > /sys/kernel/debug/tracing/options/stacktrace
# echo 1 > /sys/kernel/debug/tracing/events/probe/enable
# modprobe openvswitch
# cat trace
...
modprobe-6935 [000] d... 41899.099670: ovs_vport_init:
(ovs_vport_init+0x24/0x50 [openvswitch]) dev_table=0
  modprobe-6935 [000] d... 41899.099675: <stack trace>
=>
=> do_one_initcall
=> sys_init_module
=> system_call_fastpath
  modprobe-6935 [000] d... 41899.099678: ovs_vport_init_1:
(ovs_vport_init+0x2f/0x50 [openvswitch]) dev_table=ffff8801adf16000
  modprobe-6935 [000] d... 41899.099680: <stack trace>
=>
=> do_one_initcall
=> sys_init_module
```
Summary

- **Perftool is a powerful tool for debugging kernel**
  - Without any out-of-tree tools.
    - Perftools is shipped with kernel :)

- **If you don’t have perftools, you can use ftrace**
  - Ftrace is a native debugging tool in kernel
  - Recent x86 disassembler helps it a lot!

- **We don’t need external Debugger**
Future Work

- **What features do we need for debugging?**
  - Linux already have debugger, tracer, etc.

- **Refactoring tracing/debugging subsystems**
  - Refactoring among similar features
    - Ex) SW breakpoint is used by *probes/kgdb
    - Ingo Molnar is working for consolidation
  - Many ring-buffers in kernel
    - Ex) Perf, ftrace, relayfs etc.
  - Virtualization isn’t matter?
    - Guest tracing is similar to application tracing (LTTng helps?)
Questions?

Masami Hiramatsu
<masami.hiramatsu.pt@hitachi.com>
Debugging Kernel without Debugger

Linux Collaboration Summit 2012
Masami Hiramatsu
<masami.hiramatsu.pt@hitachi.com>
Yokohama Research Lab.
Hitachi Ltd.,
Trademarks

- Linux is a trademark of Linus Torvalds in the United States, other countries, or both.
- Intel and Core are trademarks or registered trademarks of Intel Corporation in the United States, other countries, or both.
- Other company, product, or service names may be trademarks or service marks of others.
Example 1

Add new event on socket and record it

e.g.
# perf probe -a "sock_sendmsg sock size"
# perf probe -a "sock_recvmsg sock size"
# perf record -e "probe:*" -afR top
(exit)

# perf trace
  rdesktop-15710 [001] 96311.971853: sock_recvmsg:
  (ffffffff814486c0) sock=fff880161c94780 size=4
  rdesktop-15710 [001] 96311.971868: sock_recvmsg:
  (ffffffff814486c0) sock=fff880161c94780 size=d9
  rdesktop-15710 [001] 96311.998124: sock_sendmsg:
  (ffffffff81448850) sock=fff880161c94780 size=3d
  cupsd-980   [001] 96312.414984: sock_recvmsg:
  (ffffffff814486c0) sock=fff8801abecf480 size=604
  rdesktop-15710 [001] 96312.511370: sock_recvmsg:
  (ffffffff814486c0) sock=fff880161c94780 size=4
Perf probe scripting is also available

e.g.)

```
# perf trace -g perl
generated Perl script: perf-trace.pl
# vi perf-trace.pl
sub trace_end
{
    print("Sent total size:

    while (($sock, $size) = each (%sockstat_send)) {
        printf("socket:%x %d bytes

    }

    ...
}

sub probe::sock_sendmsg
{
    ...
    $sockstat_send{$sock} += $size;
}

# perf trace -s perl:perf-trace.pl
Sent total size:
socket:ffff880161c94780 488 bytes
Received total size:
socket ffff8801abecf480 1540 bytes
socket ffff880161c94780 24258 bytes
```
Simple API

- disassemble(char *buf, size_t len, struct insn *insn)
  - Print assembly code at address into buf.
    - If next is given, put the next instruction address in that.
  - Buf : a buffer to where disassembled code is stored
  - Len: Length of buf
  - Insn: Disassemble target instruction

E.g. disassemble 1 instruction at given address

```
kernl_insn_init(&insn, address);
if (disassemble(buf, len, &insn) >= 0)
    printk("%p: %s", address, buf);
```