Kernel Probes for MIPS, ARM and PPC32

Sony India Software Centre
Sony Corporation

Madhvesh Sulibhavi (madhvesh.s@ap.sony.com)
Lubna Badkar (lubna@ap.sony.com)
Tim Bird (tim.bird@am.sony.com) Presenting
Contents

• Kernel Probes (Kprobes) Overview
• Kprobes Support for MIPS
• Kprobes Support for ARM
• Kprobes Support for PPC32 (BookE)
• Overhead Measurement Details
• Source Patches
• References
Kprobes Overview

- Kprobes is simple and lightweight mechanism to collect debugging information dynamically.

- Linux main line kernel 2.6.16 (and later) includes support for X86, PPC64, Sparc, IA64, x86_64

- Kprobes support for MIPS, ARM and PPC32 arch is implemented by Sony for vanilla kernel 2.6.16-39.
Kprobes Overview (Contd)

- Kprobe mainly contains following three handlers
  - Pre-Handler
    - Executed before the probed instruction
    - It can be used to dump the register contents before executing the probed instruction
  - Post-Handler
    - Executed after the probed instruction
    - It can be used to dump the register contents after executing the probed instructions
  - Fault-Handler
    - Executed when some fault occurs in the pre handler or post handler or in the instruction being debugged
Kprobes Overview (Contd)

Includes:
- Initializing the kprobes
- Register kprobes
- Deregister kprobes

Kprobes Management (Arch Independent)

One Kprobe At probed Address Addr1

Two Kprobes At The probed Address Addr3

Multiple Kprobes at probed Address AddrN

Kprobe Aggregate List

hlist

kprobe_table

Addr1

Addr2

Addr3

AddrN
Kprobes Overview (Contd)

Code Flow for register_kprobe()

- (kernel_module)
  - Kprobe module (Init)
  - (kernel/kprobe.c)
    - register_kprobe()
    - __register_kprobe()
    - get_kprobe()
  - (If single probe or multiple probes at different probe point)
    - arch_prepare_kprobe()
    - arch_arm_kprobe()
  - ( arch/$arch/kernel/kprobe.c )
  - (If multiple probes at the same probe point)
    - register_aggr_kprobe()
    - get_insn_slot()
    - add_aggr_kprobe()
    - copy_kprobe()
    - add_new_kprobe()
Kprobes Overview (Contd)

Code Flow for deregister_kprobe()

kprobe module (exit)

unregister_kprobe()

get_kprobe()

(If single probe or multiple probes at different probe point)

arch_disarm_kprobe()

cleanup_p = 1

arch_remove_kprobe()

free_insn_slot()

ARCH Dependent

(kernel/kprobe.c)

(If multiple probes at the same probe point)

cleanup_p = 0
MIPS Arch Kprobes Contents

• Target Environment Details
• Kprobes Design
• Sample Example Output
• Sample Example Disassembly
• Limitations
MIPS Target Environment Details

• **Target Board**
  - Toshiba RBHMA4400 (TX4937)

• **OS**
  - Linux 2.6.16.39

• **Configuration**
  - CONFIG_PREEMPT=y
  - CONFIG_KALLSYMS=y
  - CONFIG_KALLSYMSALL=y
  - CONFIG_DEBUG_KERNEL=y

**Hardware Features:**
- TX4937 @ 300Mhz.
- System Clock @ 66Mhz
- USB 1.1 Host Controller
MIPS Kprobes Design

• MIPS contains “break” instruction using which break point can be achieved.
• MIPS does not have hardware single step support and hence required alternate mechanism.
• Single step feature is implemented in software using “break” instruction with different code value
• MIPS Linux Kernel supports the following different break codes and two of these are used to implement Kprobes
### MIPS Kprobes Design (Contd)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRK_USERBP</td>
<td>User bp (used by debuggers)</td>
</tr>
<tr>
<td>BRK KERNELBP</td>
<td>Break in the kernel</td>
</tr>
<tr>
<td>BRK_ABORT</td>
<td>Sometimes used by abort(3) to SIGIOT</td>
</tr>
<tr>
<td>BRK BD_TAKEN</td>
<td>For bd slot emulation - not implemented</td>
</tr>
<tr>
<td>BRK BD_NOTTAKEN</td>
<td>For bd slot emulation - not implemented</td>
</tr>
<tr>
<td>BRK_SSTEPBP</td>
<td>User bp (used by debuggers)</td>
</tr>
<tr>
<td>BRK_OVERFLOW</td>
<td>Overflow check</td>
</tr>
<tr>
<td>BRK_DIVZERO</td>
<td>Divide by zero check</td>
</tr>
<tr>
<td>BRK RANGE</td>
<td>Range error check</td>
</tr>
<tr>
<td>BRK STACKOVERFLOW</td>
<td>For Ada stack checking</td>
</tr>
<tr>
<td>BRK_NORLD</td>
<td>No rld found - not used by Linux/MIPS</td>
</tr>
<tr>
<td>BRK_THREADBP</td>
<td>For threads, user bp (used by debuggers)</td>
</tr>
<tr>
<td>BRK BUG</td>
<td>Used by BUG()</td>
</tr>
<tr>
<td>BRK KDB</td>
<td>Used in KDB_ENTER()</td>
</tr>
<tr>
<td>BRK_MULOVF</td>
<td>Multiply overflow</td>
</tr>
</tbody>
</table>

*Linux Kernel Supported Break Codes in asm-mips/break.h*

- This is Used for Kprobe to Implement BreakPoint
- This is Used for Kprobe to Implement Single Step Operation
MIPS Kprobes Design (Contd)

• Break instruction format is

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>6 5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>CODE</td>
<td>BREAK</td>
<td></td>
</tr>
</tbody>
</table>

• When exception is generated, Exception Program Counter (EPC) register holds the instruction which raised the exception. Using the EPC value, the “code” field is extracted to check whether the exception is of BREAK0 or BREAK5 type

• By using different values for CODE field, the break point and single step operations are achieved


MIPS Kprobes Design (Contd)

Kprobes Control Flow

- Original Instruction
  - insn1
  - insn2

- Original Instruction Replaced by Break
  - Break 0
  - insn1
  - insn2

- Pre Handler
- Post Handler

New Executable Page
(Stores Original Instruction for Single Step)

- Copied Instruction
  - insn1
  - Break 5

Two Instructions are copied
insn1 = Original Instruction
Break5 = This opcode causes single step exception
MIPS Kprobes Design (Contd)

Kprobes Execution Flow

- Break Point Hit
  - Break 0 (0x0000000d)
  - Opcode=0x0000000d
  - Single Step Exception with 0x0000014d opcode

- do_break
  - Opcode=0x0000014d
  - Break 5 (0x0000014d)
  - Opcode 0x0000014d

- kprobe_handler
  - Pre Handler
  - Post_kprobe_handler
  - Post Handler

- Interrupts are Disabled
  - Break 0

- EPC will be set to “BREAK 5” in the new Executable Page

- Resume Execution
  - Restore Original Interrupts Status
  - regs->cp0_epc = orig_epc + 4
  - cp0_status &= ~ST0_IE
• Kprobes Control Flow Details
  – When a particular instruction “insn1” is probed, the original instruction will be copied on the executable page and replaces the probed instruction by ‘break 0’ i.e 0x0000000d
  – ‘break 5’ (0x0000014d) will be copied on the executable page after the original probed instruction
  – A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler
  – When break 0 is hit, kprobes exception notifier will be called by the exception handler do_break() with the die value DIE_BREAK. This will in turn call kprobe_handler() which performs executing pre_handler() associated with kprobes
MIPS Kprobes Design (Contd)

- Kprobes Control Flow Details (Contd)
  - `kprobe_handler()` takes care of moving program counter to the copied instruction by calling `prepare_single_step()`.
  - After executing the probed instruction, the next instruction on the copied page ‘break 5’ will be executed.
  - When break 5 is hit, again kprobes exception notifier will be called by the exception handler `do_break()` with the die value `DIE_SSTEPBP`. This will in turn call user defined `post_handler()`.
  - After the instruction debugging, the execution resumes to the next instruction “insn2”
When kprobe is registered, original instruction is replaced with 'break' instruction '0000000d'.

When probe point is hit, pre handler gets executed and 'pc' will be made to point to original instruction copied on executable page.

When kprobe is registered, original instruction is replaced with 'break' instruction '0000000d'.

After original instruction execution, again 'break' instruction '00000014d' is executed. This causes single-step exception and post-handler gets executed. After this, 'pc' will be set to next instruction after the probed instruction.

New Executable Page
MIPS Kprobes Limitations

- Only minimal testing is performed and needs to be tested using system tap test suites
- Not tested under SMP configuration
- Following instruction types are not supported
  - Jump register or jump and link register (ex: jr, jalr)
  - Jump And Link and Jump (ex: jal, j)
  - Branch instructions (ex: bltz, bgez, beq, bne, blez, bgtz)
  - coprocessor instructions (ex: cop1)
ARM Arch Kprobes Contents

• Target Environment Details
• Kprobes Design
• Sample Example Output
• Sample Example Disassembly
• Limitations
ARM Target Environment Details

• Target Board
  – OMAP Starter Kit (OSK 5912)

• OS
  – Linux 2.6.16.39

• Configuration
  – CONFIG_PREEMPT=y
  – CONFIG_KALLSYMS=y
  – CONFIG_KALLSYMSALL=y
  – CONFIG_DEBUG_KERNEL=y

Hardware Features:
OSK5912 @ 192 Mhz.
32 M Byte Flash ROM
USB Host Port
ARM Kprobes Design

- ARM arch does not have break or single step instructions
- Kprobes is implemented using one of the exception handling mechanism

ARM Supports Below Exception Types

<table>
<thead>
<tr>
<th>Exception</th>
<th>Mode</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast Interrupt Request</td>
<td>FIQ</td>
<td>Fast interrupt handling</td>
</tr>
<tr>
<td>Interrupt Request</td>
<td>IRQ</td>
<td>Normal interrupt handling</td>
</tr>
<tr>
<td>SWI and RESET</td>
<td>SVC</td>
<td>Protected mode for OS</td>
</tr>
<tr>
<td>Pre-fetch or data abort</td>
<td>ABT</td>
<td>Memory protection handling</td>
</tr>
<tr>
<td>Undefined Instruction</td>
<td>UND</td>
<td>SW emulation of HW coprocessors</td>
</tr>
</tbody>
</table>

Undefined Exception Type is used for Kprobes
ARM Kprobes Design (Contd)

- Undefined instructions are those which the processor cannot decode in the absence of coprocessor. So this is used for kprobes implementation.

- Undefined Instruction opcode is as below

- If the “cond” field opcode[31:28] value = 1110, then it indicates the unconditional type.
ARM Kprobes Design (Contd)

- When undefined exception is generated, Link Register (LR) will be updated with appropriate return address and this is used to implement single step operation and to resume the next instruction of the probe point
- In the exception handler, “die notifier” kernel feature is used to differentiate breakpoint and single step exceptions
ARM Kprobes Design (Contd)

Kprobes Control Flow

Original Instruction
insn1
insn2

Original Instruction Replaced by Undef
0xe7f001f8
insn2

Pre Handler

PC

Post Handler

PC

New Executable Page
(Stores Original Instruction for Single Step)

Copied Instruction

 insn1
0xe7f001fc

Two Instructions are copied
insn1 = Original Instruction
0xe7f001fc = Undefined Instruction
to Cause Single Step Exception.

Program Text

0xe7f001fc
insn1

Program Text

Program Text
ARM Kprobes Design (Contd)

Kprobes Execution Flow

- **Break Point Hit**
  - **Unref Insn** (0xe7f001f8)
  - **do_undefinstr**
  - **Opcode= 0xe7f001f8**
  - **Opcode= 0xe7f001fc**

- **Post_kprobe_handler**
  - **Resume Execution**
  - **Post Handler**
  - **PC will be set to undef insn in the new Executable Page**

- **Kprobe_handler**
  - **Undef Insn (0xe7f001fc)**
  - **Pre Handler**
  - **Op Code 0x0000014d**

- **PC will be set to undef insn, exception will be generated**

- **Single Step Exception with 0xe7f001fc opcode**

- **Kprobes Execution Flow**
  - Single Step Exception with 0xe7f001fc opcode
  - Pre Handler
  - Post Handler
  - Resume Execution
  - Original Interrupts Status Restored ARM_pc = p->addr+4
ARM Kprobes Design (Contd)

• Kprobes Control Flow Details
  – When a particular instruction “insn1” is probed, the original instruction will be copied on the executable page and replaces the probed instruction by undefined instruction ‘0xe7f001f8’ i.e BREAKPOINT_INSTRUCTION
  – Undefined Instruction with opcode 0xe7f001fc (BREAKPOINT_INSTRUCTION_2) will be copied on the executable page
  – When “BREAKPOINT_INSTRUCTION” is hit, kprobes exception notifier will be called by the exception handler do_undefinstr() with the die value DIE_UNDEF_1. This will in turn call kprobe_handler() which executes pre_handler() associated with kprobes
• Kprobes Control Flow Details (Contd)
  – kprobe_handler() takes care of moving program counter to the copied instruction by calling prepare_single_step().
  – After executing the probed instruction, the next instruction on the copied page ‘BREAKPOINT_INSTRUCTION_2’ will be executed
  – When BREAKPOINT_INSTRUCTION_2 is hit, again kprobes exception notifier will be called by the exception handler do_break() with the die value DIE_UNDEF_2. This will in turn calls post_kprobe_handler().
  – After the instruction debugging, the execution resumes to the next instruction “insn2”
ARM Sample Example Output

Kernel Module With do_fork as Probe Point

Kprobe Module
Kp.addr=do_fork
kp.pre_handler=before_hook
kp.post_handler=after_hook

before_hook() {
    Stack_dump()
    Show_allregs()
}

Stack dump

Registers

after_hook() {
    Stack_dump()
    Show_allregs()
}
ARM Example Disassembly

When kprobe is registered, the original instruction is replaced with 'undefined' instruction '0xe7f001f8'.

After original instruction execution, again 'undefined' instruction '0xe7f001fC' is executed. This causes single-step exception and post-handler will be executed. After this, pc will be moved to the next instruction after the probed instruction.
ARM Kprobes Limitations

- Only minimal testing is performed and needs to be tested using system tap test suites
- Not tested under SMP configuration
- Following instruction types are not supported
  - Branch/link with exchange instruction type (ex: bx or blx)
  - Data processing instructions whose destination register is a PC (ex: AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, MOV, BIC, MVN)
  - Load instructions which destination register is PC (ex: ldr)
  - Multiple load type instructions (ex: ldm)
  - Branch and branch with link type (Ex: b, bl)
PPC32 Arch Kprobes Contents

• Target Environment Details
• Kprobes Design
• Sample Example Output
• Sample Example Disassembly
• Limitations
PPC32 Target Environment Details

• Target Board
  – Ebony (PPC440gp)

• OS
  – Linux 2.6.16.39

• Configuration
  – CONFIG_PREEMPT=y
  – CONFIG_KALLSYMS=y
  – CONFIG_KALLSYMSALL=y
  – CONFIG_DEBUG_KERNEL=y

Hardware Features:
  440GP Processor @ 500 Mhz
  System Clock @ 66.66Mhz
PPC32 Kprobes Design

- Kprobe support for ppc64 is present in linux main line tree
- Kprobe support for PPC32 was posted recently
  http://patchwork.ozlabs.org/linuxppc/patch?id=9244
- But the above patch addresses only for non BookE versions.
- Sony has provided kprobes support for PPC32 BookE versions
- PPC32 with BookE versions involved technical challenges while
  addressing single step exception handling. These issues are
  highlighted and also the solution to overcome these is discussed
  in detail.
PPC32 Kprobes Design (Contd)

• PPC32 Arch does have trap instructions ‘tw’.
• Trap instruction format is

```
31  TO  RA  RB  4   
0   6   11  16  21  31
```

• There are different variations of trap instruction.
• This variation is mainly due to ‘TO’ field of the trap instruction
• In general trap instruction will be of the type 0x7c000008
• The trap used for kprobe is an unconditional trap “tw” i.e ‘0x7fe00008’ for which TO =31 and RA=RB=0
Different types of trap instruction are as follows

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>TO Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>twlt</td>
<td>Less than</td>
<td>16</td>
</tr>
<tr>
<td>Twle</td>
<td>Less than or equal</td>
<td>20</td>
</tr>
<tr>
<td>Tweq</td>
<td>Equal</td>
<td>4</td>
</tr>
<tr>
<td>Twge</td>
<td>Greater than or equal</td>
<td>12</td>
</tr>
<tr>
<td>Twgt</td>
<td>Greater than</td>
<td>8</td>
</tr>
<tr>
<td>Twnl</td>
<td>Not less than</td>
<td>12</td>
</tr>
<tr>
<td>Twne</td>
<td>Not equal</td>
<td>24</td>
</tr>
<tr>
<td>Twng</td>
<td>Not greater than</td>
<td>20</td>
</tr>
<tr>
<td>Twllt</td>
<td>Logically less than</td>
<td>2</td>
</tr>
<tr>
<td>Twlle</td>
<td>Logically less than or equal</td>
<td>6</td>
</tr>
<tr>
<td>Twlge</td>
<td>Logically greater than or equal</td>
<td>5</td>
</tr>
<tr>
<td>Twlgt</td>
<td>Logically greater than</td>
<td>1</td>
</tr>
<tr>
<td>Twlnl</td>
<td>Logically not less than</td>
<td>5</td>
</tr>
<tr>
<td>Twlng</td>
<td>Logically not greater than</td>
<td>6</td>
</tr>
<tr>
<td>**Tw</td>
<td>Unconditional</td>
<td><strong>31</strong></td>
</tr>
</tbody>
</table>

This is Used by Kprobes to Implement Break Point Exception
Kprobes Control Flow

- Original Instruction
- Original Instruction Replaced by Trap
- Pre Handler
- Post Handler

New Executable Page
(Stores Original Instruction for Single Step)

Copied Instruction

- NIP

Insns:
- insn1
- insn2

Program Text

- New Executable Page

- Only Original Instruction is stored. By enabling DE bit in MSR and IC, IDM bits in DBCR0, will generate single step debug exception

- Program Text

 insn1

 insn2

 insn1

 insn2

0x7fe00008
PPC32 Kprobes Design (Contd)

Kprobes Processing Flow For NON BookE Implementation

arch/powerpc/kernel/kprobes.c

Break Point Hit

preempt_count = X

program_check_exception

preempt_disable()

kprobe_handler

preempt_count = X+1

DEBUG_EXCEPTION

DebugException

preempt_enable_no_resched()

post_kprobe_handler

preempt_count = X+1

ret_from_crit_exc

preempt_count = X-1

Resume Execution

preempt_count = X-1

preempt_count = X-1
PPC32 Kprobes Design (Contd)

**Issue Noticed During Kprobes Implementation For PPC32 BookE Versions**

1. **Break Point Hit**
   - `kprobe_handler`
   - `preempt_count = X`
   - `program_check_exception`
   - `preempt_count = X`

2. **DEBUG_EXCEPTION**
   - `entry.S`
   - `traps.c`
   - `preempt_count = 0`
   - `Bug Wrapped in Exception Stack`
   - `preempt_count in Exception Stack is ZERO`
   - `Issue1`:
     - `preempt_count = -1`
     - `This Block Generates OOPS!!`
   - `Issue2`:
     - `preempt_count = X+1`
     - `Original Stack is Restored`
     - `Resume Execution`
     - `Original( Current ) Stack replaced by Exception Stack`
     - `preempt_disable()`
     - `preempt_enable_no_resched()`
     - `current stack frame`
     - `-----
        struct thread_info {
        preempt_count;
        ...
        }
        current_thread_info()
        -----`

3. **post_kprobe_handler**
   - `ret_from_crit_exc`
   - `preempt_count = X+1`
   - `entry.S traps.c`
PPC32 Kprobes Design (Contd)

Using KGDB Approach, Issue 2 Was Resolved

DebugException() {

----
Copy org stack thread_info to Exception Stack thread_info
----
kprobe_exceptions_notify()
post_kprobe_handler()
{
}

----
Copy back Exception Stack thread_info to Original stack thread_info

entry.S

---
This Small Window Caused System to Fail !! (As part of LTP stress test)

Original(Current) Stack replaced by Exception Stack

Preempt_count in Exception Stack is X+1

Program_check_exception

kprobe_handler

preempt_count = X

preempt_disable()

preempt_count = X+1

program_check_exception

kprobe_handler

preempt_count = X

entry.S

DEBUG_EXCEPTION

preempt_count = 0

DebugException

preempt_count = X+1

preempt_count in Exception Stack is ZERO

preempt_disable(

preempt_count = X+1

preempt_enable_no_resched()

preempt_count in Exception Stack is X+1

Resume Execution

entry.S

preempt_count = X+1

trasps.c

DebugException() {

----
Copy org stack thread_info to Exception Stack thread_info
----
kprobe_exceptions_notify()
post_kprobe_handler()
{
}

----
Copy back Exception Stack thread_info to Original stack thread_info

entry.S

---
This Small Window Caused System to Fail !! (As part of LTP stress test)
**PPC32 Kprobes Design (Contd)**

Solution Provided For PPC32 BookE Implementation

```c
DebugException() {
    ----
    Copy org stack thread_info to Exception Stack thread_info
    ----
    kprobe_exceptions_notify()
    post_kprobe_handler()
    {
        
    }
    ----
    Copy back Exception Stack thread_info to Original stack thread_info
}
```

```c
head_booke.h (Patch Details)
-------
rlwinm r6, r1, 0, 0, 18;
lwz r6, TI_PREEMPT(r6);
rlwinm r7, r11, 0, 0, 18;
stw r6, TI_PREEMPT(r7);
mr r1, r11;
-------
```

```c
DebugException()
```

Labels:
- **kprobe_handler**
- **preempt_count = X+1**
- **entry.S**
- **traps.c**
- **DebugException**
- **preempt_count = X+1**
- ** ISSUE2 Resolved**
- ** ISSUE1 Resolved**
- **Exception Stack Thread Info preempt_count is Copied with Org Stack Thread Info, Before SP Is Modified**
- **Org Stack Pointer Is Stored Here**
- **Exception Stack Frame Is Prepared Before this Point**
- **Here Exception SP is set**
- **Exception Stack Frame Contains Original Stack at Offset 16**

Registers:
- **R0**
- **R1**
- **R2**
- **R3**
- **R31**
- **SRR0**
- **SRR1**
- **CTR**

Contents:
- **0**
- **4**
- **8**
- **12**
- **16**
- **20**
- **24**
- **28**
- **140**
- **144**
- **148**
- **152**
- **156**
- **196**
PPC32 Kprobes Design (Contd)

Working Kprobes PPC32 BookE Execution Flow

Break Point Hit
Trap
(0x7fe00008)

DebugException

kprobe_handler

reg->msr |= MSR_DE
DBCR0 |= IC | IDM

Pre
Handler

post_kprobe_handler

Resume Execution

Post
Handler

DBCR0 set with IC and IDM bits

MSR set with DE bit

Causes Single Step Exception

Since DE bit set in MSR and IC, IDM bits are set in DBCR0,
Single Step Exception is generated

Interrupts are Disabled
reg->msr &= ~(MSR_EE)
reg->msr &= ~(MSR_CE)

Original Interrupts
Status Restored
reg->nip = p->addr+4

Opcode = 0x7fe00008

April-2007 ELC

Sony India Software Centre
PPC32 Kprobes Design (Contd)

• Kprobes Control Flow Details
  – When a particular instruction “insn1” is probed, the original instruction will be copied on the executable page and replaces the probed instruction by ‘trap’ i.e 0x7fe00008
  – A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler
  – When ‘tw’ is hit, kprobes exception notifier will be called by the exception handler program_check_exception () with the die value DIE_BPT. This will in turn call kprobe_handler() which performs executing pre_handler() associated with kprobes
PPC32 Kprobes Design (Contd)

- Kprobes Control Flow Details (Contd)
  - `kprobe_handler()` takes care of moving program counter to the copied instruction by calling `prepare_single_step()`. DE bit of MSR and IC, IDM bits of DBCR0 will also be set as part of `prepare_single_step()` and this will generate debug exception.
  - Since MSR[DE] bit is set, single step operation will be performed on the copied instruction in the executable page and kprobes exception notifier will be called by the exception handler `DebugException()` with the die value `DIE_SSTEP`. This will in turn call user defined `post_handler()`
  - After the instruction debugging, the execution resumes to the next instruction “Insn2”
PPC32 Sample Example Output

Kernel Module With do_fork as Probe Point

Kprobe Module
Kp.addr=do_fork
kp.pre_handler=before_hook
kp.post_handler=after_hook

before_hook()
{
  Stack_dump()
  Show_allregs()
}

after_hook()
{
  Stack_dump()
  Show_allregs()
}

Stack dump

Registers

Call Trace:
[C7CE7D90] [C00092F4] show_stack+
0x3c/0x194 (unreliable)
[C7CE7D90] [D10862C4] k_001_before_hook+
0x44/0x70 [k_001]
[C7CE7D90] [C02152C4] kprobe_exceptions_notify+
0x354/0x47c
[C7CE7E00] [C0215B74] notifier_call_chain+
0x50/0x6c
[C7CE7E00] [C0214C04] __kprobes_text_start+
0x234/0x4d4
[C7CE7E70] [C02021D4] ret_from_except_full+
0x0/0x4c
[C7CE7F30] [7FA0CB18] 0x7fa0cb18
[C7CE7F40] [C0001BE4] ret_from_syscall+0x0/0x3c

NIP: C017050 LR: C0095BC CTR: 30089150
REGS: c7ce7e80 TRAP: 0700 (2.6.16.39-alp-alp)
MSR: 00029000 <CE,EE,ME> CR: 22000022
XER: 20000000
TASK = e0476050[611] 'ash' THREAD: c7ce6000
GPR00: 000000C0 C7CE7F30 C0476050 00000011
7FA0CAE0 C7CE7F50 00000000 00000000
GPR08: 00000000 00000000 C0021E44 00000000
00000000 78000000 1001228 FFF9ED50
GPR16: 8FA72401 FFF837F1 03F940AA 00000000
00000000 10064728 10064738
GPR24: 00000000 00000000 10064718 10064718
00000000 10070000 10067460 C7CE7F50
NIP [C0017050] do_fork+0x0/0x238
LR [C00095BC] sys_fork+0x50/0x64

Since this is the Exception Stack, there is NO CALL TRACE !!!

The Registers are:
NIP: D1074004 LR: C0095BC CTR: 30089150
REGS: c02f150 TRAP: 2002 (2.6.16.39-alp-alp)
MSR: 00001000 <ME> CR: 22000022
XER: 20000000
TASK = e0476050[611] 'ash' THREAD: c7ce6000
GPR00: 000000C0 C7CE7ED0 C0476050 00000011
7FA0CAE0 C7CE7F50 00000000 00000000
GPR08: 00000000 00000000 C0001E44 00000000
00000000 78000000 1001228 FFF9ED50
GPR16: 8FA72401 FFF837F1 03F940AA 00000000
00000000 10064728 10064738
GPR24: 00000000 00000000 10064718 10064718
00000000 10070000 10067460 C7CE7F50
NIP [D107404] 0xd1074004
LR [C00095BC] sys_fork+0x50/0x64
### PPC32 Example Disassembly

When kprobe is registered, the original instruction is replaced with 'unconditional trap' instruction '0x7fe00008'.

<table>
<thead>
<tr>
<th>0000fa4 &lt;do_fork&gt;:</th>
<th>8001fa4 &lt;do_fork&gt;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1fa4: 94 21 ff c0</td>
<td>1fa4: 7f e0 00 08</td>
</tr>
<tr>
<td>stwu r1,-64(r1)</td>
<td>trap</td>
</tr>
<tr>
<td>1fa8: 7d 80 00 26</td>
<td>1fa8: 7d 80 00 26</td>
</tr>
<tr>
<td>mfc r12</td>
<td>mfc r12</td>
</tr>
<tr>
<td>1fac: 7c 08 02 a6</td>
<td>1fac: 7c 08 02 a6</td>
</tr>
<tr>
<td>mfr r0</td>
<td>mfr r0</td>
</tr>
<tr>
<td>1fb0: bf 01 00 20</td>
<td>1fb0: bf 01 00 20</td>
</tr>
<tr>
<td>stmw r24,32(r1)</td>
<td>stmw r24,32(r1)</td>
</tr>
<tr>
<td>1fb4: 7c 7e 1b 78</td>
<td>1fb4: 7c 7e 1b 78</td>
</tr>
<tr>
<td>mr r30,r3</td>
<td>mr r30,r3</td>
</tr>
<tr>
<td>1fb8: 90 01 00 44</td>
<td>1fb8: 90 01 00 44</td>
</tr>
<tr>
<td>stw r0,68(r1)</td>
<td>stw r0,68(r1)</td>
</tr>
<tr>
<td>1fbc: 7c 9f 23 78</td>
<td>1fbc: 7c 9f 23 78</td>
</tr>
<tr>
<td>mr r31,r4</td>
<td>mr r31,r4</td>
</tr>
<tr>
<td>1fc0: 91 81 00 1c</td>
<td>1fc0: 91 81 00 1c</td>
</tr>
<tr>
<td>stw r12,28(r1)</td>
<td>stw r12,28(r1)</td>
</tr>
<tr>
<td>1fc4: 7c bc 2b 78</td>
<td>1fc4: 7c bc 2b 78</td>
</tr>
<tr>
<td>mr r28,r5</td>
<td>mr r28,r5</td>
</tr>
<tr>
<td>1fc8: 7c db 33 78</td>
<td>1fc8: 7c db 33 78</td>
</tr>
<tr>
<td>mr r27,r6</td>
<td>mr r27,r6</td>
</tr>
<tr>
<td>1fca: 7c ff 3b 78</td>
<td>1fca: 7c ff 3b 78</td>
</tr>
<tr>
<td>mr r26,r7</td>
<td>mr r26,r7</td>
</tr>
<tr>
<td>1fd0: 7d 19 43 78</td>
<td>1fd0: 7d 19 43 78</td>
</tr>
<tr>
<td>mr r25,r8</td>
<td>mr r25,r8</td>
</tr>
<tr>
<td>1fd4: 48 00 00 01</td>
<td>1fd4: 48 00 00 01</td>
</tr>
<tr>
<td>bl 1fd4 &lt;do_fork+0x30&gt;</td>
<td>bl 1fd4 &lt;do_fork+0x30&gt;</td>
</tr>
</tbody>
</table>

When probe point is hit, kprobe executes pre-handler and DE bit in MSR register will be enabled and pc is made to point original instruction copied on executable page.

After original instruction execution on executable page, since DE bit is enabled in MSR register, it generates single step debug exception. During debug exception, the post-handler gets executed and pc will be moved to the next instruction after the probed instruction.

Executable Page

When kprobe is registered, the original instruction is replaced with 'unconditional trap' instruction '0x7fe00008'.

When probe point is hit, kprobe executes pre-handler and DE bit in MSR register will be enabled and pc is made to point original instruction copied on executable page.

After original instruction execution on executable page, since DE bit is enabled in MSR register, it generates single step debug exception. During debug exception, the post-handler gets executed and pc will be moved to the next instruction after the probed instruction.

Executable Page
PPC32 Kprobes Limitations

- If post handler tries to dump stack contents, then call trace cannot be displayed because during post handler execution, the stack corresponds to Exception Stack.
- Only minimal testing is performed and needs to be tested using system tap test suites
- Not tested under SMP configuration
# Overhead Measurement Details

Kprobes Overhead Measurement Data for MIPS, ARM and PPC32 Arch

Overhead Measurement Test Sample: [http://sourceware.org/ml/systemtap/2006-q1/msg00318.html](http://sourceware.org/ml/systemtap/2006-q1/msg00318.html)

<table>
<thead>
<tr>
<th>Overhead</th>
<th>MIPS (usec/call)</th>
<th>ARM (usec/call)</th>
<th>PPC32 (usec/call)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time taken to execute gettimeofday</td>
<td>1.16</td>
<td>5.22</td>
<td>0.99</td>
</tr>
<tr>
<td>Time taken to execute gettimeofday when a kprobe is inserted at do_gettimeofday with only Pre-handler</td>
<td>2.98</td>
<td>11.85</td>
<td>3.26</td>
</tr>
<tr>
<td><strong>pre_handler Overhead</strong></td>
<td>1.81</td>
<td>6.62</td>
<td>2.27</td>
</tr>
<tr>
<td>Time taken to execute gettimeofday when a kprobe is inserted at do_gettimeofday with only Post-handler</td>
<td>2.99</td>
<td>12.04</td>
<td>3.27</td>
</tr>
<tr>
<td><strong>post_handler Overhead</strong></td>
<td>1.82</td>
<td>6.81</td>
<td>2.28</td>
</tr>
<tr>
<td>Time taken to execute gettimeofday when a kprobe is inserted at do_gettimeofday with both Pre and post handler</td>
<td>3.01</td>
<td>12.10</td>
<td>3.30</td>
</tr>
<tr>
<td><strong>(pre+post)_handler Overhead</strong></td>
<td>1.84</td>
<td>6.87</td>
<td>2.31</td>
</tr>
</tbody>
</table>
Source Patches

- MIPS patches can be extracted from below link under patches for 2.6.16
  - [http://tree.celinuxforum.org/CelfPubWiki/PatchArchive](http://tree.celinuxforum.org/CelfPubWiki/PatchArchive)

- ARM patches can be extracted from below link under patches for 2.6.16
  - [http://tree.celinuxforum.org/CelfPubWiki/PatchArchive](http://tree.celinuxforum.org/CelfPubWiki/PatchArchive)

- PPC32 patches will be uploaded very soon in CELF Patch Archive page
References