

Oracle Open Source Software

Debugging Xen and Xen Guests

Mukesh Rathor
Oracle Corporation

June 2008

Xen Summit NA 2008

Debugging in Xen Environment

- Dearth of debuggers for both the hypervisor and guests.
 - Work started Dec of 2007
 - Two debuggers: kdb and gdbsx
 - First released kdb for xen in spring.
 - debugger for the hypervisor and dom0
 - Next added gdbsx and support for it in the hyp
 - gdbserver stub for PV/HVM guests
 - Both released and can be found at external repo:
<http://xenbits.xensource.com/ext/debuggers.hg>
-
-

Debugging in Xen Environment

Part I

gdbserver stub for xen (gdbsx)

gdbSX : Introduction and motivation

- A simple gdbserver stub that runs on dom0 and connects to an unprivileged guest, PV or HVM
 - Needed a 32bit binary to debug both 32 and 64bit guests
 - 32bit dom0 on 64bit hypervisor is Oracle's preferred combination
 - Very difficult compiling libxc in 32bit mode for 64bit guest support
 - Memory read writes done much easier in the hypervisor, thus removing the need for page table knowledge at the user level.
-
-

gdbsx : overview

- Makes hypercalls directly without going thru libxc
 - Full standard debugger support via gdb: register/memory read and modifications, single step, etc..
 - Support for multiple VCPUs, and single step will execute only one VCPU
 - Changes include a new subdir gdbsx in tools/debugger and minor alterations to the hypervisor
-
-

gdb: hypervisor changes

- hypervisor modified to add 4 new *XEN_DOMCTL_** subcalls
 - *XEN_DOMCTL_guestmemio*
 - will read write guest memory
 - *XEN_DOMCTL_(un)pausevcpu*
 - pause/unpause single vcpu for single step, and delayed bp execution.
 - *XEN_DOMCTL_xgdb_domstatus*
 - return any VCPU event (breakpoint or watchpoint)
-
-

gdbsx: structure

- Simple stub with two parts
 1. gx: interface between gdb and gdbsx
 2. xg: interface between gdbsx and xen
- The APIs exported by xg are generic can be used by any front end other than gdb, a windows debugger for example.
- Coming up soon:
 - Hardware watchpoint

gdbsx : example (2 VCPUs)

```
dom0>gdbsx 127.0.0.1:9999 --attach 3 64
      attach to a 64bit guest running as domid 3
anyhost>gdb vmlinux
gdb>target remote dom0:9999
Remote debugging using mrtest:9999
[New Thread 0]
[Switching to Thread 0]
0xfffffffff801083aa in hypercall_page ()
(gdb) i threads
  2 Thread 1 ffffffff801083aa in hypercall_page() (VCPU1)
*1 Thread 0 ffffffff801083aa in hypercall_page() (VCPU0)
(gdb) thr 2
[Switching to thread 2 (Thread 1)]#0  0xfffffffff801083aa
    in hypercall_page ()
(gdb) i r
rax                0x0                0x0
rbx                0x0                0x0
rcx                0xfffffffff801083aa  0xfffffffff801083aa
```

gdbsx : in conclusion

- Moving things down to hypervisor simplified things greatly
 - Memory read write functions in hypervisor shared by kdb
 - While gdbsx allows source level guest debug, kdb can provide more intense low level debug for hypervisor or dom0 kernel.
-
-

Debugging in Xen Environment

Part II

kdb for xen



kdb for xen - Features

- kdb implemented fully in the hypervisor
 - Need serial access to activate and use it
 - earlykdb: to break into kdb early during boot
 - All CPUs are paused, essentially freezing the system
 - Single step will only step that CPU while others stay in kdb
 - Examine domain and vcpu details in pretty print
 - Set breakpoints in the hypervisor
 - Display special registers, irq table, etc...
-
-

kdb for xen – Entering kdb

- Breaking into kdb: once the serial line is setup, and the system booted with kdb-built hypervisor, ctrl+\ will break into it (xen>make kdb=y)
 - IPI sent to all cpus to pause.
 - All cpus enter kdb. One becomes main kdb cpu, while others enter kdb pause mode.
 - Switching cpu via the cpu command causes target cpu to become kdb main cpu, while the leaving cpu goes into kdb pause mode.
-
-

kdb for xen – commands overview

- h for list of commands
 - dr / mr : display modify registers
 - dw/dd/mw/md : display modify words/dwords
 - dwm/ddm: display words/dwords at machine addr
 - dis : disassemble instructions
 - f : display stack frame
 - ni/ss: single step, over or into functions
 - cpu all : show cpu status for all cpus
 - dom/vcpu: display domain/vcpu details
 - gdt/dirq: display gdt, irq table, etc...
 - kdb*: commands to debug kdb itself
-
-

kdb for xen – Some Examples

Enter debugger, set breakpoint, step..:

```
root#[mrtest]/root>
```

```
Enter kdb (cpu:1 reason:1 vcpu=1 domid:32767 eflg:0x246)
```

```
ffff828c801235e2: idle_loop+52                               jmp  idle_loop+55
```

```
[1]xkdb> bp schedule                                     <===== CPU 1
```

```
BP set for domid:7fff at: 0xffff828c80115370 schedule+0
```

```
[1]xkdb> bp ffff828c801153a3
```

```
BP set for domid:7fff at: 0xffff828c801153a3 schedule+33
```

```
[1]xkdb> go
```

```
Breakpoint on cpu 1 at 0xffff828c80115370
```

```
ffff828c80115370: schedule+0                               subq $0x58, %rsp
```

```
[1]xkdb> go
```

```
Breakpoint on cpu 0 at 0xffff828c801153a3
```

```
ffff828c801153a3: schedule+33                               mov 0xd0(%rbx), %r12
```

```
[0]xkdb> ni                                             <===== CPU 0
```

```
ffff828c801153aa: schedule+3a                               call get_s_time+0
```

```
[0]xkdb> ni
```

```
ffff828c801153af: schedule+3f                               mov %rax, %r13
```

```
[0]xkdb>ss
```

```
ffff828c801153b2: schedule+42                               mov 0xc8(%rbx), %eax
```

kdb for xen: examples contd.

```
[0]xkdb> dr sp
```

```
(XEN) ----[ Xen-3.1.3  x86_64  debug=n  Not tainted ]----  
(XEN) CPU:      0  
(XEN) RIP:      e008:[<ffff828c801153b8>] schedule+0x48/0x270  
(XEN) RFLAGS: 00000000000000312    CONTEXT: hypervisor  
(XEN) rax: 0000000000000000  rbx: ffff828c8020ff28  rcx: 0000000000000002  
(XEN) rdx: 0000000000000000  rsi: 000000012bea18c7  rdi: ffff828c801e2538  
(XEN) rbp: ffff828c8020ff28  rsp: ffff828c8020fe80  r8: 000001c9ecfe0e79  
(XEN) r9:  ffff828c801e2520  r10: 0000000000000005  r11: ffff83007d3fec98  
(XEN) r12: ffff83007c402080  r13: 000001ca27bc4326  r14: 0000000000000000  
(XEN) r15: 0000000000000000  cr0: 000000008005003b  cr4: 000000000000026b0  
(XEN) cr3: 000000007d59c000  cr2: 00000000b5c66bd0  
(XEN) ds: 007b    es: 007b    fs: 0000    gs: 0000    ss: 0000    cs: e008
```

```
Special Registers:
```

```
IDTR: addr: ffff828c80240820  limit: 0fff  
GDTR: addr: ffff820000000000  limit: efff  
cr0: 000000008005003b  cr2: 00000000b5c66bd0  
cr3: 000000007d59c000  cr4: 000000000000026b0  
TR: e040
```

kdb for xen: examples contd.

See CPU state for all:

```
[7]xkdb> cpu all
[0]000000000009cb51d: 0: _stext+ffffffff405c951d mov $0x36, %eax
[1]ffff828c8011faaf: serial_puts+ef addq $0x18, %rsp
[2]ffff828c801235e2: idle_loop+52 jmp idle_loop+55
[3]ffff828c801235e2: idle_loop+52 jmp idle_loop+55
[4]000000000037ad6a: 4: ??? +0 cmp (%edx), %al
[5]ffff828c801235e2: idle_loop+52 jmp idle_loop+55
[6]ffffffff80108401: 2: ??? +0 push %r11
[7]ffff828c801235e2: idle_loop+52 jmp idle_loop+55
```

See memory for dom0:

```
[0]xkdb> dw bf9ad4a8 16 0
00000000bf9ad4a8: 00b2a5b4 00b23202 0000000a 00044504
```

kdb for xen : examples contd.

Switch cpu and examine state:

```
[7]xkdb> cpu 6
ffffffff80108401: 2: ??? +0                push %r11
[6]xkdb> dr
(XEN) ----[ Xen-3.1.3  x86_64  debug=n  Not tainted ]----
(XEN) CPU:      6
(XEN) RIP:      e033:[<ffffffff80108401>]
(XEN) RFLAGS: 0000000000000246  CONTEXT: guest
(XEN) rax: 0000000000000002  rbx: 0000000000000000  rcx: 000000000181f70d
(XEN) rdx: ffffffff800cca1d44  rsi: ffffffff800cca1d44  rdi: 0000000000000004
(XEN) rbp: 0000000000000005  rsp: ffffffff800cca1d28  r8:  ffffffff800087a000
(XEN) r9:  000000000181f708  r10: 0000000000000005  r11: ffffffff5fd000
(XEN) r12: ffffffff800cca1dd8  r13: 0000000000000000  r14: 0000000000000005
(XEN) r15: ffffffff800cca1dd8  cr0: 000000008005003b  cr4: 00000000000006f0
(XEN) cr3: 0000000420105000  cr2: 000000552aade5a0
(XEN) ds: 0000  es: 0000  fs: 0000  gs: 0000  ss: e02b  cs: e033
```

kdb for xen – Examining domain

```
[1]xkdb> dom 0
```

```
DOMAIN :      domid:0x0000 ptr:0xffff83007c622080
  shared:0xffff83007c409000
  pgalk:0x0001 -1 0
  pglist:  0xffff8284012e8000 0xffff828400e1b2c0
  xpulist: 0xffff82840136a258 0xffff82840136a168
  PAGES:  tot:0x00021e00 max:0xffffffff xenheap:0x00000005
  next:0x0000000000000000 hashnext:0x0000000000000000
  rangesets:  nxt:0xffff83007c4f7880 prev:0xffff83007d3f1b80 lk:0x0001
  -1 0
  Evt: ptr:0xffff83007c6220f8 lk:0x0001 -1 0
  Grant table: gp:0xffff83007c406d00
                nr_frames:0x00000004          shpp:0xffff83007d3f1900
  active:0xffff83007c406e00
  maptrk:0xffff83007d3f1080 maphd:0x00000000 maplmt:0x00000200
  mapcnt:0x00000000 lk: 0x0001 -1 0
  hvm:0 priv:1 dbg:0 poll:0 dying:0 paused:0
  .....
  vcpu ptrs:
    0:ffff83007c4f4080 1:ffff83007c4f2080
```

kdb for xen: Examining vcpu

```
[1]xkdb> vcpu ffff83007c4f4080
```

```
VCPU:    vcpu-id:0x00000000    vcpu-ptr:0xffff83007c4f4080
```

```
cpu:0 dom:0xffff83007c622080 domid:0
```

```
pause: flags:0x0000000000000000 count:0
```

```
vcpu:  initdone:1 polling:0 running:0
```

```
nmi:  pend:0 masked:0 shut: def:0 paused:0
```

```
vcpu_info:0xffff83007c409000 next:0xffff83007c4f2080
```

```
periodic: period:0x3d0900 last_event:0x649feff407
```

```
cpu_affinity:0xffffffff vcpu_dirty_cpumask:0x1
```

```
sched_priv:0xffff83007c4f7a80
```

```
arch info: (ffff83007c4f4200)
```

```
guest_context: flags:0 user_regs:
```

```
.....
```

```
TF_flags:0x0000000000000001 guest_table:0x00000000004259f3
```

```
cr3: 0x00000004259f3000 vcpu_info_mfn:0xffffffffffffffff
```

```
paging:
```

```
xlite:0 vtlb:ffff8300cf0fd950
```

```
mode: &:ffff828c801c31a0 guestlevels:3
```

```
&shadow:ffff828c801c31f8
```

kdb for xen – Guest debug support

- Very useful if dom0 is hung
- Examine/modify memory, display context, set breakpoints, single-step a guest domain....
- Symbol table hints must be provided to xen :

```
[1]xkdb>sym 4 c0696084 c068a590 c0696080  
will set symbols for a 32bit guest with domid 4
```

```
where : kallsyms_names == c0696084  
        kallsyms_addresses == c068a590  
        kallsyms_num_syms == c0966080
```

```
grep from guest System.map* file
```

kdb for xen – guest dbg examples

```
[0]xkdb> sym 3 c02c3e64 c02bba70 c02c3e60
Number of symbols:8444
Succesfully saved symbol info
[0]xkdb> bp do_timer 3
BP set for domid:3 at: 0xc0123e98 3:do_timer+0
[0]xkdb> go
Breakpoint on cpu 0 at 0xc0123e98
00000c0123e98: 3:do_timer+0 add $0x1,0xc02990a0
[0]xkdb> bc all
Deleted breakpoint [0] addr:0xc0123e98 domid:3
[0]xkdb> go
```

kdb for xen – guest dbg examples

Display stack frame:

```
[0]xkdb> f
00000000c0123e98: 3:do_timer+0          add $0x1,0xc02990a0
00000000c010dbb0: 3:timer_interrupt+24e pop %ecx
00000000c01094aa: 3:handle_IRQ_event+44 add $0xc, %esp
00000000c0109a38: 3:do_IRQ+122         xchg %ebx, %esp
```

where :

3: indicates the context is for domain id 3

kdb for xen – Major Challenges

- Two major challenges:
 - I. Pausing All CPUs : IPI or NMI
 - ◆ The CPU to receive ctrl+/, or to enter kdb via exception, becomes main kdb cpu. It can pause other CPUs either via an IPI or an NMI.
 - II. Synchronizing breakpoints deletion :
 - ◆ A trap must never be lost or left spurious
-
-

kdb for xen : Pausing All CPUs

- All CPUs must disable as soon as possible upon entering kdb
 - IPI: would result in deadlock if another cpu is trying to send IPI to this initial kdb cpu which is disabled because it hit breakpoint in disabled code
 - NMI:
 - guaranteed immediate delivery irrespective of other CPUs state
 - However, because NMI can't be masked, we can't atomically save state or update a bitmap used to keep track of all cpus that hit INT3 (details later). Hence not usable.
-
-

kdb for xen – Pausing CPUs contd.

- kdb solution is to use IPI and remember if INTs were disabled upon entering kdb.
 - All CPUs enable interrupts as soon as they enter kdb no matter what the reason, a debug event or kdb pause IPI.
 - Once all CPUs except the kdb main cpu (`kdb_initial_cpu`) acknowledge to be in kdb pause, a kdb session begins, and all CPUs disable interrupts.
-
-

kdb for xen – breakpoint synch

- kdb must always tell the hypervisor exactly if a trap was handled or not.
 - Ignoring a legitimate trap may result in panic, or a guest debugger being fed unnecessary trap.
 - Eg: if cpu 2 and 5 hit a bp simultaneously, and if cpu 2 deletes it, cpu 5 must be sure to ignore the bp and tell hypervisor it handled the trap.
 - Problem exacerbated if user deletes and immediately sets the same breakpoint again.
 - Furthermore, what if a guest sets the same breakpoint that kdb in xen does?
-
-

Kdb for xen – breakpoint synch contd

- Solution is to keep a bitmap of all cpus that are sitting on a trap frame.
 - This is failsafe as INTs are disabled upon trap
 - A cpu that hits a bp trap sets its corresponding bit before enabling interrupts
 - Bitmap also needed because a CPU has no easy way of knowing in kdb if it's sitting on a trap
 - A deleted bp is marked for delete, but remains in the breakpoint table
-
-

kdb for xen – breakpoint synch

contd

- When user quits kdb, kdb will not resume normal execution until all CPUs sitting on a trap have a chance to examine the breakpoint table.
 - After all cpus examine and process their traps, the last cpu will flush the breakpoint table, purging deleted breakpoints
 - Eg. cpu 0, 3, 4 hit bp simultaneously. 0 deletes the bp. 3 and 4 will examine the deleted bp and tell the hypervisor they handled the trap. 4 will purge the bp from bp table.
-
-

kdb for xen – little more work to do

- Hardware watchpoints
 - HVM guest support not as robust as in gdbstx (issues with step, modifying state).
 - Timers confuse HVM guest, and dom0. Need to figure all timers that must be frozen.
 - Some sort of exception handling for very occasional protection faults upon bad data entry.
-
-

kdb for xen – In Conclusion

- Though not source level, kdb does provide a reliable method of debugging, or learning the internals of xen hypervisor.
 - Independence from external programs like gdb makes it possible to tailor it specifically for the hypervisor.
-
-

Oracle Open Source Software

The End

