Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Evolution in Kernel Debugging using Hardware Virtualization With Xen

Nitin A. Kamble
nitin.a.kamble@intel.com
Jun Nakajima
jun.nakajima@intel.com
Asit K. Mallick
asit.k.mallick@intel.com

Open Source Technology Center, Intel Corporation

Abstract

Xen's ability to run unmodified guests with the virtualization available in hardware opens new doors of possibilities in the kernel debugging. Now it's possible to debug the Linux kernel or any other PC operating system similar to debugging a user process in Linux. Since hardware virtualization in-processor enables Xen to implement full virtualization of a guest OS, there is no need to change the kernel in any way to debug it.

This paper demonstrates the new evolutionary debug techniques using examples. It also explains how the new technique actually works.

1 Introduction

The Xen[1] open source virtual machine monitor initially started with software virtualization by modifying the guest OS kernel. Since Xen 3.0, it also supports the Intel® Virtualization Technology® [2] to create and run unmodified guests. This Xen capability to run unmodified Linux OS or any other unmodified OS also provides a new opportunity to debug an unmodified OS using the Xen VMM.

With this guest debug capability, it is possible to trap into an unmodified guest such as any Linux, Windows, DOS, or any other PC OS; and check the register state, modify registers, set debug breakpoints anywhere including in the kernel, read and write memory, or inspect or modify the code currently being executed. This new method uses gdb[3] as the front end for debugging. With gdb also comes the source-level debugging of an unmodified Linux kernel. There are some advantages of using this debug approach compared to other kernel debug options, such as the Linux kernel stays unmodified, and ability of setting of breakpoints anywhere in the code. In fact it is also possible to set breakpoints in the boot loader such as grub [4] or inside the guest BIOS code.

2 The Architecture and Design of debugging of an unmodified guest

The virtualization technology in the processor, and Xen's ability to take advantage of it, let an unmodified OS run inside a virtual machine.

The following sections first briefly describe the virtualization technology in the Intel IA32 processors, and how Xen[5] hypervisor utilizes this hardware virtualization to create virtual machines (domain) for unmodified guests.

2.1 Intel Virtualization Technology for IA32 Architecture

Virtualiztion Techinology in the Intel processors augment the IA32 architecture by providing new processor operation modes called VMX operations. And the Virtual-Machine Control Structure controls the operation of the virtual machine running in the VMX operation.

The following subsections introduce the VMX Operation and the Virtual-Machine Control Structure briefly.

2.1.1 Introduction to VMX operation

VT processor support for virtualization is provided by a new form of processor operation called VMX operation. There are two kinds of VMX operations: *VMX root operation* and *VMX nonroot operation*. The Xen VMM runs in VMX root operation and guest software runs in VMX non-root operation. Transitions between VMX root operation and VMX non-root operation are called VMX transitions. There are two kinds of VMX transitions. Transitions into VMX non-root operation are called *VM entries*. Transitions from VMX non-root operation to VMX root operation are called *VM exits*. Figure 1 depicts the interactions between the VMX root and VMX nonroot operations.

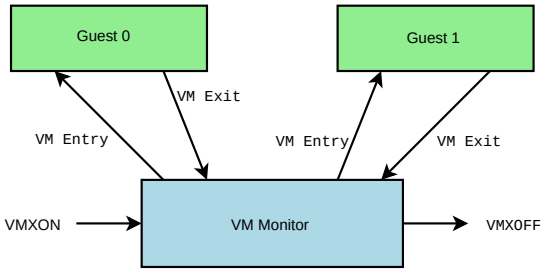

Figure 1: Interaction of Virtual-Machine Monitor and Guests

Processor behavior in VMX root operation is very much as it is outside VMX operation or without the VT feature in the processor. The principal differences are that a set of new instructions (the VMX instructions) is available and that the values that can be loaded into certain control registers are limited. Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, certain instructions (including the new VMCALL instruction) and events cause VM exits to the VMM. Because these VM exits replace ordinary behavior, the functionality of software in VMX non-root operation is limited. It is this limitation that allows the VMM to retain control of processor resources.

Because VMX operation places these restrictions even on software running with current privilege level (CPL) 0, guest software can run at the privilege level for which it was originally designed.

2.1.2 Virtual-Machine Control Structure

VMX non-root operation and VMX transitions are controlled by a data structure called a virtual machine control structure (VMCS). Ac-

cess to the VMCS is managed through a component of processor state called the VMCS pointer (one per logical processor). The value of the VMCS pointer is the 64-bit address of the VMCS. The VMCS pointer can be read and written using the instructions VMPTRST and VMPTRLD. The VMM configures a VMCS using other instructions: VMREAD, VMWRITE, and VMCLEAR.

Please refer to the latest IA-32 SDM[6] for more details on the Virtual Machine Extensions (VMX) in the Intel Processor.

2.2 Xen support for unmodified Guest using the Hardware Virtualization

The processor state of the running vcpu is stored in the VMCS area. Xen uses a different VMCS for each unmodified guest vcpu. So when it is scheduling from one VMX guest to another VMX guest, it switches the VMCS to save and load the processor context automatically. To get into the hypervisor, paravirtualized Guests use hyper calls, similar to a process doing sys-call into OS for privileged operations. On Xen, unmodified guests run in restricted mode (VMX nonroot operation). In that mode all the virtualization-related processor instructions and events cause a VM Exit, switching to the hypervisor. With the VM Exits there is no need to modify the guest OS to add the hyper calls in the kernel.

The unmodified guest OS thinks that it is in control of its physical memory management, such as page tables, but the Xen hypervisor is monitoring the guest page table usage. Xen handles the page faults, TLB flush instructions for the Guest OS, and maintains shadow-translated page tables for the unmodified guests.

2.3 Internals of debugging an unmodified guest on Xen

Figure 3 shows the interactions happening in various Xen components when an unmodified guest is being debugged. Both `gdb` and `gdbserver-xen` are processes running in the Xen-paravirtualized service OS, also known as *domain-0*. `gdb` is totally unmodified. `gdbserver` is a gdb tool used for remote debug. `gdbserver-xen` is a modified gdbserver for utilizing Xen hyper call based interfaces available in domain-0.

The following sections describe interactions and implementation details for the the Xen components exercised while debugging a unmodified guest OS.

2.3.1 gdb and gdbserver-xen interactions

The gdbserver [7] is a standard tool available to use with gdb for remote gdb. It uses a ASCII protocol over the serial line or over the network to exchange debug information. See Figure 2 for a pictorial view of this interaction.

The gdbserver implements `target_ops` for Linux remote debugging. And `gdbserver-xen` basically extends the standard Linux gdbserver by implementing the `target_ops` specific to Xen. The interaction between gdb and gdbserver-xen is no different than gdb and the standard gdbserver.

2.3.2 Communication between gdbserver-xen and libxenctrl library

The `target_ops` from the gdbserver-xen such as `linux_fetch_registers`, `linux_store_registers`, `linux_read_memory`, and `linux_write_memory` use the `xc_`

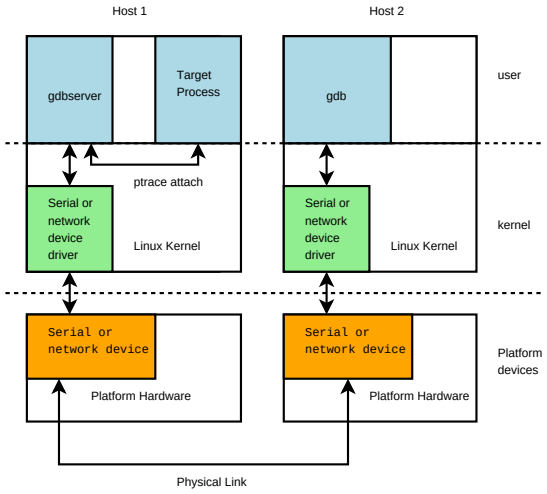

Figure 2: Typical use of gdbserver to remotely debug a Linux process

`ptrace` interface from the `libxenctrl` library to exchange the processor state or memory contents for the VMX domain. The `xc_ptrace()` function implements the ptrace-like mechanism to control and access the guest.

2.3.3 libxenctrl and dom0 privcmd device interactions

Gdb's request to get and set processor registers is implemented in the `libxenctrl` library by `xc_ptrace()` calls like `PTRACE_GETREGS`, `PTRACE_SETREGS`, `PTRACE_GETFPREGS`, `PTRACE_SETFPREGS`, `PTRACE_GETFPXREGS`, and `PTRACE_SETFPXREGS`. Inside the `xc_ptrace()`, the registers are fetched by the calling `fetch_regs()` and changed by calling `xc_vcpu_setcontext()` functions. `fetch_regs()` uses the `dom0_op(DOM0_GETVCPUCONTEXT)` Xen hyper call to get the context of the guest vcpu. The hyper call is performed by making `IOCTL_PRIVCMD_HYPERCALL` ioctl on the `/proc/xen/privcmd` domain-0 xen device.

For gdb's request to get or change code (text) and data memory, the `xc_ptrace` requests like `PTRACE_POKETEXT, PTRACE_POKEDATA` are used. These ptrace requests use the `map_domain_va()` function to map the guest memory in the `gdbserver-xen` process's address space, and do the read/write operations on that mapped memory. The `map_domain_va()` function is also part of the `libxenctrl` library. It finds out the mode the guest vcpu is running in, like real mode or 32-bit protected paging disabled, or paging enabled or paging with PAE enabled, or 64-bit IA32e mode (long mode). If the guest processor is in paging-disabled mode such as real mode then it can get the machine physical address for the address gdb has requested directly using the `page_array[]` for the domain.[1]

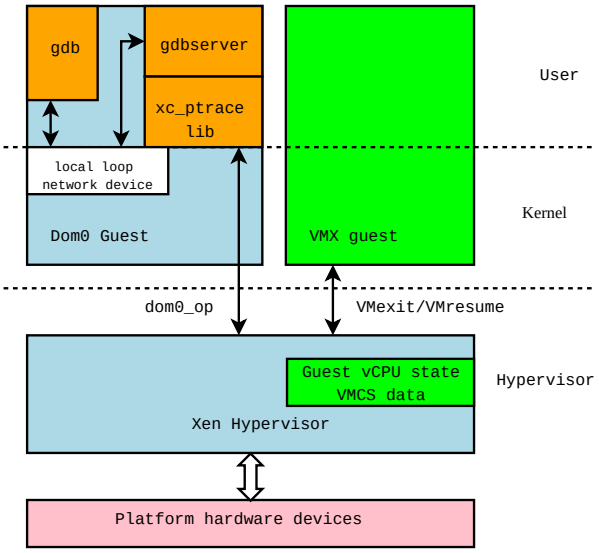

Figure 3: Unmodified guest debug interactions

Then the `xc_map_foreign_range()` function is used to map the machine page frame of

[1]The `page_array[]` is an array of the physical page frames numbers allocated to the guest in the guest physical address order. It is built at the time of creating a new domain. `page_array[guest_physical_pfn]` gives corresponding `machine_physical_pfn`. The `page_array[]` for the guest domain is obtained by making `DOM0_GETMEMLIST` `dom0_op()` hyper call.

the guest page in the gdbserver-Xen's address space. It uses IOCTL_PRIVCMD_MMAP ioctl on the privcmd domain-0 xen device to map the machine page frames into the the gdbserver-xen process's address space. Once it is mapped in the gdbserver-xen process's address space, it then performs simple memory reads or writes to access the guest memory contents.

map_domain_va() determines if the guest vcpu is running in paging-enabled mode or not, by looking at control registers received from the fetch_regs() call for the guest vcpu. The control registers CR0 and CR4 tells which mode the guest vcpu is running such as real, protected paging disabled, protected paging enabled, PAE, or IA32e mode. The control register CR3 points to the guest physical address of the current page table the guest vcpu is using. The IA32 architecture has different-format page tables for different processor paging modes.

The libxenctrl library implements map_domain_va_32, map_domain_va_pae, and map_domain_va_64 functions to handle these different page table formats. The map_domain_va_64() handles page tables for both 4k and 2M pages in the IA32e mode.

After getting the guest physical address for the gdb requested virtual address by traversing the guest page tables, then the rest of the functionality to map the guest page frame and perform read/write is implemented similar to the paging-disabled situation described above.

2.3.4 privcmd domain-0 device and the Xen hyper visor Interactions

The /proc/xen/privcmd device is implemented as a device driver in the dom0's paravirtualized kernel. For the device, the IOCTL_PRIVCMD_MMAP ioctl is implemented by making a HYPERVISOR_mmu_update hyper-call in Xen. And like all the other hyper calls, the IOCTL_PRIVCMD_HYPERCALL ioctl is implemented by making a call at right the offset in the hypercall_page.

The hypervisor_page has the handlers for the hyper calls with various parameters. And it is initialized by the Xen hypervisor at the domain creation time. The initialization of the hypercall_page involves writing the appropriate code in the page for hyper call handlers. For x86_64 domain-0 the hypercall_page is initialized with syscall handlers; for i386 it is it is initialized with int 0x82 calls. For supervisor_mode_kernel i386 domain-0 kernel it is initialized with the long call instruction. All these methods get the processor execution control into the Xen hypervisor, and call the appropriate function from the hypercall_table. After the execution of the hyper call function, the hypervisor returns control to the next instruction after the hyper call in the dom0, by making an iret or long call, or sysret instruction.

2.3.5 The Xen hyper visor infrastructure for debugging

The get and set of vcpu context dom0_ops described in Section 2.3.3 provide gdb with read/write access to the guest vcpu registers. All the dom0_op hyper calls are handled by the do_dom0_op() function in the Xen hypervisor. The DOM0_GETVCPUCONTEXT dom0_op gets the vcpu register context of the guest vcpu by reading the per-vcpu context information for the guest domain stored in the hypervisor. Not all information is available there, because some of the guest register state is stored in VMCS for faster access. It needs to get the register state information from the VMCS to get the complete register state of the guest vcpu.

In a multi-processor system, the domain-0 vcpu running the `dom0_op` can be different than the guest domain vcpu being debugged. And the VMCS structures are per logical processor; one processor can not operate on other processor's VMCS. So if the processors backing these two vcpus are different, then the vcpu running the `dom0_op()` sends an *IPI* (InterProcessor Interrupt) to ask the other vcpu to provide its VMCS guest registers' context state. The VMCS registers' context state is obtained by temporarily loading the VMCS for the guest vcpu and performing VMREAD instructions for the guest registers.

The `DOM0_GETVCPUCONTEXT` `dom0_op` is implemented similarly using the VMWRITE instruction.

The guest memory access is enabled by mapping the guest memory in the gdbserver-xen's address space using the `mmu_update` hyper call; this is described in Section 2.3.4. The `mmu_update` hyper call is implemented in the hypervisor by modifying the page table entries of the gdbserver-xen process running in paravirtualized domain-0, such that the requested virtual address maps to the machine frame number of the requested guest memory location.

The PIT (Programable Interrupt Timer) virtulization for the the unmodified guests is also altered in the Xen hypervisor for so that it does not try to inject the timer ticks guest has missed due to debugger stopping the guest.

2.3.6 Setting breakpoint in the gdb

Gdb sets breakpoints in the guest by placing an `int3` instruction at the breakpoint location. Whenever the processor encounters the `int3` instruction while executing, a `#BP` exception is raised, resulting in a VMexit into the hypervisor. The hypervisor handles VMExits based on the exit reason, and for the breakpoint exception VMexit, it simply pauses the guest domain.

Meanwhile the gdbserver is waiting for the guest domain to pause. Once it discovers that it has paused, it uses the `xc_ptrace` interfaces to get the processor register state of the guest domain's vcpu, and passes it on to gdb. `gdb` was waiting for a response from the gdbserver. Once it gets the response, it shows where the guest is paused, by looking at the `eip/rip` from the guest vcpu register state. At this point, the gdb user can issue various gdb commands to access and manipulate the guest processor and memory state.

When the user asks gdb to `continue` the debugged guest, gdbserver unpauses the guest domain, and waits for it to get into paused state again. The gdbserver also handles the `CTRL-C` press from its terminal, pausing the domain and returning control to `gdb`.

3 Current Limitations

There are two types of limitations. One is limitation in the use of gdb, because not all the gdb commands are implemented in the gdbserver-xen. And then there are limitations due to the Xen environment of the unmodified guests.

3.1 Limitations on gdb use

Currently breakpoints are implemented only using the `int3` instruction and traps. The processor debug registers are not touched. This puts some limitations on the guest debugging.

Currently these gdb capabilities are missing from the gdbserver-xen for debugging an unmodified guest:

- hardware breakpoints
- watchpoints
- single stepping

3.2 Limitations on the guest driver debugging

The unmodified guests running on the Xen sees only those hardware devices emulated by Xen. And as of now, unmodified guests running on Xen can not access the platform devices directly. Hence this debug capability can not be used to debug any arbitrary device driver. Only the device driver for the virtualized devices Xen shows to the unmodified guests can be debugged.

In the future with the Intel Virtualization Technology for Directed I/O [8] and Xen's capability to assign machine devices directly to the unmodified guests, it will be possible to use this debug capability to also debug the device drivers for the platform's devices.

4 Comparison with other Linux Kernel debuggers

There are other debugging [9] options available for debugging the Linux Kernel, such as software debuggers: KDB [10], KGDB [11], and hardware JTAG based debuggers: Arium Intarget probe [12].

4.1 KDB

The Linux Kernel Debugger (KDB) is a patch for the Linux kernel and provides a means of examining kernel memory and data structures while the system is operational.

4.1.1 Advantages compared to KDB

- No modification to the Linux kernel is needed. KDB needs a KDB enabled/ patched Linux kernel. If the KDB patch for the desired kernel is not available, then there will be more effort to port the KDB patch to the desired Linux kernel.
- Not only Linux, any PC operating system can debugged.
- Can set breakpoint anywhere in the kernel, even in the interrupt handlers. With KDB, the kernel code used by KDB can not be debugged.
- Source-level debugging. KDB does not support source-level debugging.
- Can also debug boot loader or the guest BIOS code.

4.1.2 Disadvantages compared to KDB

- Requires a system with a VT-capable processor.
- With today's Xen, arbitrary device drivers can not be debugged.
- Single-step is currently not supported. Instead, breakpoints can be used.
- Does not support extra kernel-aware commands. For example, KDB supports `ps`, `btp`, and `bta` commands to show the running processes and their back traces.

4.2 KGDB

KGDB is a remote host Linux kernel debugger through gdb and provides a mechanism to debug the Linux kernel using gdb.

4.2.1 Advantages compared to KGDB

- The Linux kernel can be debugged with just one system. KGDB needs two systems, one running the Linux kernel, and another controlling it.
- Not only Linux, any PC operating system can debugged.
- No modification to the Linux kernel is needed. KGDB also needs a KGDB enabled/patched Linux kernel. If the KGDB patch for the desired kernel is not available, then there will be more effort to port the KGDB patch to the desired Linux kernel.
- Can also debug boot loader or the guest BIOS code.

4.2.2 Disadvantages compared to KGDB

- Requires a system with a VT-capable processor.
- With today's Xen, arbitrary device drivers can not be debugged.
- Single-step is currently not supported. Instead, breakpoints can be used.

5 Debug environment setup

The following sections provide information on what you need and how to set up your own environment for debugging the linux kernel using Xen and hardware virtualization and use it effectively.

5.1 Requirements

1. Hardware: In order to run modified guest domains on Xen, first you need a system with processor capable of Intel Virtualization Technology. There is a page [13] set up on the xen wiki here for currently released VT-enabled processors; it can help you in finding the right processor. Going forward, all future Intel processors will incorporate Virtualization Technology.
2. Xen VMM: then you need to get a Xen with the gdbserver changes. Any version of Xen after revision `9496:e08dcff87bb2` dated 31 March 2006 should work. Instructions on how to build and install on your Linux box can be found in the Xen user manual [14]. Instead of building, you can take the ready-built 3.0.2 (or newer) rpm from the download section [15] of the Xensource website.
3. gdbserver-xen: this is the remote agent used to attach gdb to a running, unmodified guest. The sources for gdbserver-xen are part of the Xen source code. The `tools/debugger/gdb/README` file from the Xen source code provides information on how to build gdbserver-xen.
4. gdb: If you are running `x86_64` Xen, then you need 64-bit gdb. If you are running `x86_32` or `i386` Xen, then you need 32-bit gdb. `gdb` should be provided by your distribution. Xen allows running a 32-bit OS on the 64-bit Xen using VT—in that case, you will need to use the 64-bit versions of gdb and gdbserver.

5.2 Setting up the debug environment

Once you have all the required components, then you can go ahead with the setup as follows.

1. Start the unmodified guest normally on top of xen. If you are not familiar with Xen, you can refer to the Xen user manual [14].

2. get the `domain_id` for the running guest from the `xm list` command in the service domain (domain 0).

3. attach gdbserver-xen to the running guest with this command:
   `gdbserver-xen localhost:9999 --attach <domain_id>`

4. Then start gdb in domain 0 or on a remote host. If you have the binary file with symbols for the guest kernel, you can pass it to gdb.
   `gdb -s vmlinux`
   You can get this symbol file for various released distributions. Appendix A has more information about it.

   If you do not have such a symbol file for the running guest kernel, you still can debug it by running gdb with no arguments, but you will not be able operate with symbols from gdb.

5. Enter the gdb command as shown in Table 1 at the gdb prompt to set up the right environment for gdb. With this, gdb uses the appropriate protocol to communicate with the gdbserver-xen to exchange the architecture state. These initialization gdb commands can also be placed in the `.gdbinit` configuration file.

6. Enter this at the gdb prompt to attach to the remote gdbserver:
   `target remote <host_running_gdbserver>:9999`

Now you should see the `eip/rip` where the guest is stopped for debugging. Figure 4 shows the screen shot from starting the gdbserver and gdb connection. It would be more convenient to use separate terminals for the gdbserver-xen and gdb, because you can stop execution of the running guest any time by pressing `CTRL-C` in the gdbserver-xen terminal.

Now from gdb you can try these commands:

- get registers

```
info all-registers
info registers
info registers eax
```

- set registers

```
set $rip=$rip+2
set $edi=$esi
```

- look at memory contents

```
p /4d 0xc00abd23
p page_array
```

- change memory contents

```
set *(long *)0xc01231bf=-1
set my_struct[5].my_member=5
set *(long)($rbp +8) = 0
```

- look at the disassembly code

```
x /10i $rip
x /10i $eip
x /10i my_function
```

- look at the back trace

```
bt
where
```

- set breakpoints

```
break *$rip+0x10
break my_function
```

- If you are running on a `x86_64` Xen, set the 64-bit architecture in gdb:
  `set architecture i386:x86-64:intel`
- If you are running on a 32-bit Xen, set the 32-bit architecture in gdb:
  `set architecture i386:intel`

Table 1: gdb environment setup for gdbserver-xen

- In one terminal start the gdbserver-xen

```
[root@localhost ~]# xm list
Name                        ID Mem(MiB) VCPUs State  Time(s)
Domain-0                     0     1024     4 r-----   132.7
ExampleHVMDomain             2      512     1 -b----    11.7

[root@localhost ~]./gdbserver-xen localhost:9999 --attach 2
Attached; pid = 2
Listening on port 9999
```

- In another terminal start the gdb

```
[root@localhost ~]# gdb
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".

(gdb) set architecture i386:x86-64:intel
The target architecture is assumed to be i386:x86-64:intel

(gdb) target remote localhost:9999
Remote debugging using localhost:9999
[New Thread 0]
[Switching to Thread 0]
0x00000000c0109093 in ?? ()
(gdb)
```

Figure 4: starting the gdsberver-xen and gdb connection

- continue to get the breakpoints hit

```
cont
```

- delete breakpoints

```
delete 1
```

- advance to some location

```
advance my_function
```

- jump execution to some other location

```
jump *0xffffffff8010940f
jump crash.c:90
jump my_function
```

6 Examples

- Figure 5 shows how to get the debuginfo for the stock Red Hat Enterprise Linux 3 Update 5 kernel, and use it with gdb to debug the kernel at source level.
- Figure 6 shows a custom-compiled x86_64 2.6.16 Linux kernel debugged at source level on 64-bit Xen.
- Figure 7 shows a freedos beta9rc5 debugging without the symbols information.

7 Summary and Conclusion

The paper describes unmodified Linux kernel debugging at source level using Xen on platforms with hardware virtualization processors. It describes how the gdb commands gets implemented in the various components of Xen. It shows how to set up this debug environment and provides a high-level comparison between other Linux kernel debuggers.

Appendix A

Debug info for stock distribution kernels. Figure 5 show how to use this kernel-debuginfo rpm to debug the Linux kernel at source level.

- Red Hat Fedora Core Distributions: `http://download.fedora.redhat.com/pub/fedora/linux/core/{1,2,3,4,5}/{i386,x86_64}/debug/`
- Red Hat Enterprise Linux Distributions: `http://updates.redhat.com/enterprise/{3AS,3ES,3WS,3desktop,4AS,4ES,4WS,4Desktop}/en/os/Debuginfo/{i386,x86_64}/RPMS/`
- SuSE Linux 10.1 `ftp://ftp.suse.com/pub/projects/kernel/kotd/{i386,x86_64}/SL101_BRANCH/`
- Other Linux Distributions: I could not find the debuginfo rpms for other distributions.

Acknowledgments

Thanks to my colleagues at Intel Open Source Technology Center and friends in the open source community for their continuous support.

Thanks to Keir Fraser and Ian Pratt for providing me comments and suggestions for my gdbserver-related patches for unmodfied (VMX or HVM) guests to Xen.

Thanks to John W. Lockhart for helping me format this paper in LaTeX.

```
[root@localhost ~]# wget http://updates.redhat.com/enterprise/3AS/en/os/Debuginfo\
/i386/RPMS/kernel-debuginfo-2.4.21-32.EL.i686.rpm
[root@localhost ~]$ wget http://updates.redhat.com/enterprise/3AS/en/os/\
> Debuginfo/i386/RPMS/kernel-debuginfo-2.4.21-32.EL.i686.rpm
--11:22:49--  http://updates.redhat.com/.../kernel-debuginfo-2.4.21-32.EL.i686.rpm
           => `kernel-debuginfo-2.4.21-32.EL.i686.rpm'
Length: 48,079,241 [application/x-rpm]
100%[====================================>] 48,079,241   304.66K/s    ETA 00:00
11:25:27 (297.27 KB/s) - `kernel-debuginfo-2.4.21-32.EL.i686.rpm' saved [48,079,241/48,079,241]

[root@localhost ~]# rpm -ivh kernel-debuginfo-2.4.21-32.EL.i686.rpm
warning: kernel-debuginfo-2.4.21-32.EL.i686.rpm: Header V3 DSA signature: NOKEY, key ID db42a60e
Preparing...                ########################################### [100%]
   1:kernel-debuginfo       ########################################### [100%]
[root@localhost ~]# gdb -s /usr/lib/debug/boot/vmlinux-2.4.21-32.EL.debug
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
Using host libthread_db library "/lib64/libthread_db.so.1".

(gdb) set architecture i386:x86-64:intel
The target architecture is assumed to be i386:x86-64:intel
(gdb) target remote localhost:9999
Remote debugging using localhost:9999
[New Thread 0]
[Switching to Thread 0]
default_idle () at process.c:96
96      }
(gdb) list
91                      if (!need_resched())
92                              safe_halt();
93                      else
94                              __sti();
95              }
96      }
97
98      /*
99       * On SMP it's slightly faster (but much more power-consuming!)
100      * to poll the ->need_resched flag instead of waiting for the
(gdb) p /x swapper_pg_dir
$2 = {{pgd = 0x0} <repeats 768 times>, {pgd = 0x102063}, {pgd = 0x103063}, {
    pgd = 0x104063}, {pgd = 0x105063}, {pgd = 0x1063}, {pgd = 0x2063}, {
    pgd = 0x3063}, {pgd = 0x4063}, {pgd = 0x5063}, {pgd = 0x6063}, {
    pgd = 0x7063}, {pgd = 0x8063}, {pgd = 0x9063}, {pgd = 0xa063}, {
    pgd = 0xb063}, {pgd = 0xc063}, {pgd = 0xd063}, {pgd = 0xe063}, {
    pgd = 0xf063}, {pgd = 0x10063}, {pgd = 0x11063}, {pgd = 0x12063}, {
    pgd = 0x13063}, {pgd = 0x14063}, {pgd = 0x15063}, {pgd = 0x16063}, {
    pgd = 0x17063}, {pgd = 0x18063}, {pgd = 0x19063}, {pgd = 0x1a063}, {
    pgd = 0x1b063}, {pgd = 0x1c063}, {pgd = 0x1d063}, {pgd = 0x1e063}, {
    pgd = 0x1f063}, {pgd = 0x20063}, {pgd = 0x21063}, {pgd = 0x22063}, {
    pgd = 0x23063}, {pgd = 0x24063}, {pgd = 0x25063}, {pgd = 0x26063}, {
    pgd = 0x27063}, {pgd = 0x28063}, {pgd = 0x29063}, {pgd = 0x2a063}, {
    pgd = 0x2b063}, {pgd = 0x2c063}, {pgd = 0x2d063}, {pgd = 0x2e063}, {
    pgd = 0x2f063}, {pgd = 0x30063}, {pgd = 0x31063}, {pgd = 0x32063}, {
    pgd = 0x33063}, {pgd = 0x34063}, {pgd = 0x35063}, {pgd = 0x36063}, {
    pgd = 0x37063}, {pgd = 0x38063}, {pgd = 0x39063}, {pgd = 0x3a063}, {
    pgd = 0x3b063}, {pgd = 0x3c063}, {pgd = 0x0}, {pgd = 0x0}, {
    pgd = 0x1bb0067}, {pgd = 0x0} <repeats 185 times>, {pgd = 0x3e067}, {
    pgd = 0x0}, {pgd = 0x0}, {pgd = 0x3d067}}
(gdb)
```

Figure 5: An Example of source-level debugging of a 32-bit Red Hat RHEL3 Update 5 stock Linux kernel on 64-bit Xen.

```
[root@localhost linux-2.6.16]$ gdb -s vmlinux
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
(no debugging symbols found)
Using host libthread_db library "/lib64/libthread_db.so.1".

The target architecture is assumed to be i386:x86-64:intel
[New Thread 0]
[Switching to Thread 0]
0x000000000040046f in ?? ()
(gdb) break schedule
Breakpoint 6 at 0xffffffff803c3044
(gdb) cont
Continuing.

Breakpoint 6, 0xffffffff803c3044 in schedule ()
(gdb) x /5i $rip
0xffffffff803c3044 <schedule+4>:        push   %r15
0xffffffff803c3046 <schedule+6>:        push   %r14
0xffffffff803c3048 <schedule+8>:        push   %r13
0xffffffff803c304a <schedule+10>:       push   %r12
0xffffffff803c304c <schedule+12>:       push   %rbx
(gdb) delete 6
(gdb) break pcnet32_start_xmit
Breakpoint 7 at 0xffffffff80288590
(gdb) cont
Continuing.

Breakpoint 7, 0xffffffff80288590 in pcnet32_start_xmit ()
(gdb) x /5i $rip
0xffffffff80288590 <pcnet32_start_xmit>:        sub    $0x48,%rsp
0xffffffff80288594 <pcnet32_start_xmit+4>:      mov    %r15,0x40(%rsp)
0xffffffff80288599 <pcnet32_start_xmit+9>:      mov    %rbx,0x18(%rsp)
0xffffffff8028859e <pcnet32_start_xmit+14>:     mov    %rsi,%r15
0xffffffff802885a1 <pcnet32_start_xmit+17>:     mov    %rbp,0x20(%rsp)
(gdb) delete 7
(gdb) break ret_from_intr
Breakpoint 8 at 0xffffffff8010af64
(gdb) cont
Continuing.

Breakpoint 8, 0xffffffff8010af64 in ret_from_intr ()
(gdb) delete 8
(gdb) cont
Continuing.
```

Figure 6: Example of source-level debugging of a 64-bit custom-compiled 2.6.16 Linux kernel on 64-bit Xen.

```
[root@localhost linux-2.6.16]$ gdb -s vmlinux
[root@ljrl4 ~]# gdb
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb) set architecture i386:intel
The target architecture is assumed to be i386:intel
(gdb) target remote localhost:9999
Remote debugging using localhost:9999
[New Thread 0]
[Switching to Thread 0]
0x000001eb in ?? ()

(gdb) x/10i $eip
0x1eb:  lock push %ebx
0x1ed:  incl   (%eax)
0x1ef:  lock push %ebx
0x1f1:  incl   (%eax)
0x1f3:  lock push %ebx
0x1f5:  incl   (%eax)
0x1f7:  lock push %ebx
0x1f9:  incl   (%eax)
0x1fb:  lock push %ebx
0x1fd:  incl   (%eax)
(gdb) info registers
eax            0x301e1  197089
ecx            0x40006  262150
edx            0x6      6
ebx            0x35a    858
esp            0xa2c    0xa2c
ebp            0xd0a38  0xd0a38
esi            0x4b0    1200
edi            0xd04b0  853168
eip            0x1eb    0x1eb
eflags         0x23246  143942
cs             0x70     112
ss             0xcf     207
ds             0x70     112
es             0xcf     207
fs             0xf000   61440
gs             0xf000   61440
(gdb) x /16x 0x10*$ds + $esi
0x175a: 0x007004b0      0x147c0a94      0x0aa0fd8e      0x0a7e21b5
0x176a: 0x73eb000e      0x14220700      0x007e145c      0x147c0000
0x177a: 0x218f218f      0x90909090      0xfd8e0000      0x218f147c
0x178a: 0xfd8e0203      0x0000218f      0x00000000      0x00000000
(gdb) cont
Continuing.
```

Figure 7: Example of debugging freedos beta9rc5 on 32-bit Xen.

References

[1] The xen virtual machine monitor. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/.

[2] Intel virtulization technology. http://www.intel.com/technology/computing/vptech/.

[3] Gdb: The gnu project debugger. http://www.gnu.org/software/gdb.

[4] Grub: Gnu grand unified bootloader. http://www.gnu.org/software/grub/.

[5] Xen architecture and design documents. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/architecture.html.

[6] Ia-32 intel®architecture software developer's manual, volume 3b, chapters 19-23. http://developer.intel.com/design/mobile/core/duodocumentation.htm.

[7] Remote debugging with gdb. http://www.kegel.com/linux/gdbserver.html.

[8] Intel virtualization technology for directed i/o architecture. ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf.

[9] Steve Best. Linux debugging techniques article. Technical report, IBM Linux Technology Center. http://www-128.ibm.com/developerworks/linux/library/l-debug/.

[10] Kdb: Built-in kernel debugger. http://oss.sgi.com/projects/kdb/.

[11] Kgdb: Linux kernel source level debugger using gdb conenction over serial line. http://sourceforge.net/projects/kgdb.

[12] Arium in-taget probe. http://www.arium.com/products/ecmxdpice.html.

[13] Intel processors with vt feature. http://wiki.xensource.com/xenwiki/IntelVT.

[14] Xen user manual. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/user/user.html.

[15] Xen readymade rpms. http://xensource.com/xen/downloads/index.html.

Disclaimer

The opinions expressed in this paper are those of the author and do not necessarily represent the position of the Intel Corporation.