kvm: the Linux Virtual Machine Monitor

Avi Kivity
Qumranet
avi@qumranet.com

Yaniv Kamay
Qumranet
yaniv@qumranet.com

Dor Laor
Qumranet
dor.laor@qumranet.com

Uri Lublin
Qumranet
uril@qumranet.com

Anthony Liguori
IBM
aliguori@us.ibm.com

Abstract

Virtualization is a hot topic in operating systems these days. It is useful in many scenarios: server consolidation, virtual test environments, and for Linux enthusiasts who still can not decide which distribution is best. Recently, hardware vendors of commodity x86 processors have added virtualization extensions to the instruction set that can be utilized to write relatively simple virtual machine monitors.

The Kernel-based Virtual Machine, or **kvm**, is a new Linux subsystem which leverages these virtualization extensions to add a virtual machine monitor (or hypervisor) capability to Linux. Using **kvm**, one can create and run multiple virtual machines. These virtual machines appear as normal Linux processes and integrate seamlessly with the rest of the system.

1 Background

Virtualization has been around almost as long as computers. The idea of using a computer system to emulate another, similar, computer system was early recognized as useful for testing and resource utilization purposes. As with many computer technologies, IBM led the way with their VM system. In the last decade, VMware's software-only virtual machine monitor has been quite successful. More recently, the Xen [xen] open-source hypervisor brought virtualization to the open source world, first with a variant termed *paravirtualization* and as hardware became available, full virtualization.

2 x86 Hardware Virtualization Extensions

x86 hardware is notoriously difficult to virtualize. Some instructions that expose privileged state do not trap when executed in user mode, e.g. *popf*. Some privileged state is difficult to hide, e.g. the current privilege level, or *cpl*.

Recognizing the importance of virtualization, hardware vendors [Intel][AMD] have added extensions to the x86 architecture that make virtualization much easier. While these extensions are incompatible with each other, they are essentially similar, consisting of:

- A new guest operating mode – the processor can switch into a guest mode, which has all the regular privilege levels of the normal operating modes, except that system software can selectively request that certain instructions, or certain register accesses, be trapped.
- Hardware state switch – when switching to guest mode and back, the hardware switches the control registers that affect processor operation modes, as well as the segment registers that are difficult to switch, and the instruction pointer so that a control transfer can take effect.
- Exit reason reporting – when a switch from guest mode back to host mode occurs, the hardware reports the reason for the switch so that software can take the appropriate action.

3 General `kvm` Architecture

Under `kvm`, virtual machines are created by opening a device node (`/dev/kvm`.) A guest has its own memory, separate from the userspace process that created it. A virtual cpu is not scheduled on its own, however.

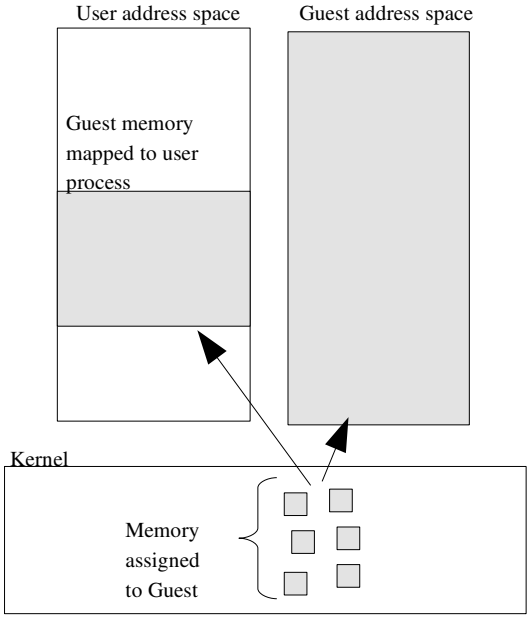

Figure 1: `kvm` Memory Map

3.1 `/dev/kvm` Device Node

`kvm` is structured as a fairly typical Linux character device. It exposes a `/dev/kvm` device node which can be used by userspace to create and run virtual machines through a set of `ioctl()`s.

The operations provided by `/dev/kvm` include:

- Creation of a new virtual machine.
- Allocation of memory to a virtual machine.
- Reading and writing virtual cpu registers.
- Injecting an interrupt into a virtual cpu.
- Running a virtual cpu.

Figure 1 shows how guest memory is arranged. Like user memory in Linux, the kernel allocates discontiguous pages to form the guest address space. In addition, userspace can `mmap()` guest memory to obtain direct access. This is useful for emulating dma-capable devices.

Running a virtual cpu deserves some further elaboration. In effect, a new execution mode, *guest mode* is added to Linux, joining the existing *kernel mode* and *user mode*.

Guest execution is performed in a triply-nested loop:

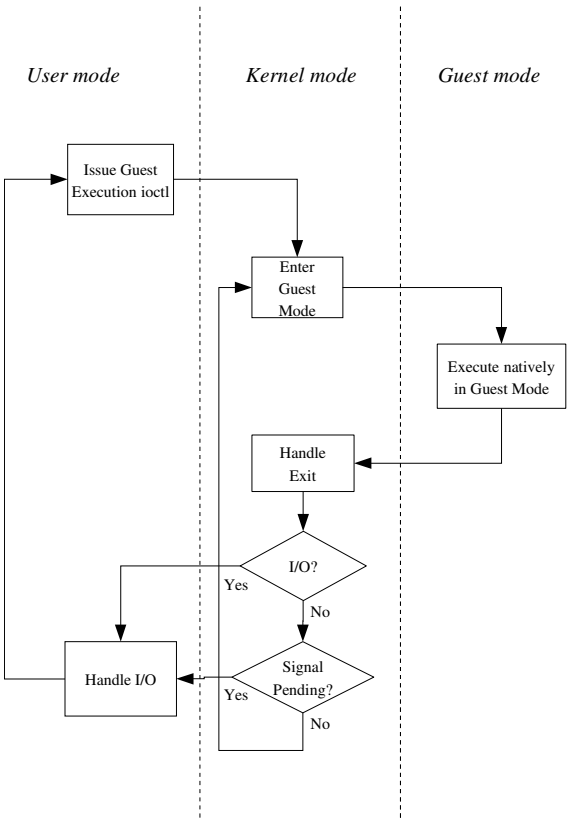

Figure 2: Guest Execution Loop

- At the outermost level, userspace calls the kernel to execute guest code until it encounters an I/O instruction, or until an external event such as arrival of a network packet or a timeout occurs. External events are represented by signals.

- At the kernel level, the kernel causes the hardware to enter guest mode. If the processor exits guest mode due to an event such as an external interrupt or a shadow page table fault, the kernel performs the necessary handling and resumes guest execution. If the exit reason is due to an I/O instruction or a signal queued to the process, then the kernel exits to userspace.

- At the hardware level, the processor executes guest code until it encounters an instruction that needs assistance, a fault, or an external interrupt.

Refer to Figure 2 for a flowchart-like representation of the guest execution loop.

3.2 Reconciling Instruction Set Architecture Differences

Unlike most of the x86 instruction set, where instruction set extensions introduced by one vendor are adopted by the others, hardware virtualization extensions are not standardized. Intel and AMD processors have different instructions, different semantics, and different capabilities.

`kvm` handles this difference in the traditional Linux way of introducing a function pointer vector, `kvm_arch_ops`, and calling one of the functions it defines whenever an architecture-dependent operation is to be performed. Base `kvm` functionality is placed in a module, `kvm.ko`, while the architecture-specific functionality is placed in the two arch-specific modules, `kvm-intel.ko` and `kvm-amd.ko`.

4 Virtualizing the MMU

As with all modern processors, x86 provides a virtual memory system which translates user-visible virtual addresses to physical addresses that are used to access the bus. This translation is performed by the *memory management unit*, or *mmu*. The mmu consists of:

- A radix tree ,the *page table*, encoding the virtual-to-physical translation. This tree is provided by system software on physical memory, but is rooted in a hardware register (the `cr3` register)
- A mechanism to notify system software of missing translations (page faults)
- An on-chip cache (the *translation lookaside buffer*, or *tlb*) that accelerates lookups of the page table
- Instructions for switching the translation root in order to provide independent address spaces
- Instructions for managing the tlb

The hardware support for mmu virtualization provides hooks to all of these components, but does not fully virtualize them. The principal problem is that the mmu provides for one level of translation ($guest virtual \rightarrow guest physical$) but does not account for the second level required by virtualization ($guest physical \rightarrow host physical$.)

The classical solution is to use the hardware virtualization capabilities to present the real mmu with a separate page table that encodes the combined translation ($guest virtual \rightarrow host physical$) while emulating the hardware's interaction with the original page table provided by the guest. The shadow page table is built incrementally; it starts out empty, and as translation failures are reported to the host, missing entries are added.

A major problem with shadow page tables is keeping the guest page table and the shadow page table synchronized. Whenever the guest writes to a page table, the corresponding change must also be performed on the shadow page table. This is difficult as the guest page table resides in ordinary memory and thus is not normally trapped on access.

4.1 Virtual TLB Implementation

The initial version of shadow page tables algorithm in `kvm` used a straightforward approach that reduces the amount of bugs in the code while sacrificing performance. It relies on the fact that the guest must use the tlb management instructions to synchronize the tlb with its page tables. We trap these instructions and apply their effect to the shadow page table *in addition* to the normal effect on the tlb.

Unfortunately, the most common tlb management instruction is the context switch, which effectively invalidates the entire tlb.[1] This means that workloads with multiple processes suffer greatly, as rebuilding the shadow page table is much more expensive than refilling the tlb.

4.2 Caching Virtual MMU

In order to improve guest performance, the virtual mmu implementation was enhanced to allow page tables to be cached across context switches. This greatly increases performance at the expense of much increased code complexity.

As related earlier, the problem is that guest writes to the guest page tables are not ordinarily trapped by the virtualization hardware. In order to receive notifications of such guest writes, we *write-protect* guest memory pages that are shadowed by `kvm`. Unfortunately, this causes a chain reaction of additional requirements:

[1] Actually, kernel mappings can be spared from this flush; but the performance impact is nevertheless great.

- To write protect a guest page, we need to know which translations the guest can use to write to the page. This means we need to keep a *reverse mapping* of all writable translations that point to each guest page.
- When a write to a guest page table is trapped, we need to emulate the access using an x86 instruction interpreter so that we know precisely the effect on both guest memory and the shadow page table.
- The guest may recycle a page table page into a normal page without a way for `kvm` to know. This can cause a significant slowdown as writes to that page will be emulated instead of proceeding at native speeds. `kvm` has heuristics that determine when such an event has occurred and decache the corresponding shadow page table, eliminating the need to write-protect the page.

At the expense of considerable complexity, these requirements have been implemented and `kvm` context switch performance is now reasonable.

5 I/O Virtualization

Software uses *programmed I/O* (*pio*) and *memory-mapped I/O* (*mmio*) to communicate with hardware devices. In addition, hardware can issue *interrupts* to request service by system software. A virtual machine monitor must be able to trap and emulate pio and mmio requests, and to simulate interrupts from virtual hardware.

5.1 Virtualizing Guest-Initiated I/O Instructions

Trapping pio is quite straightforward as the hardware provides traps for pio instructions and partially decodes the operands. Trapping mmio, on the other hand, is quite complex, as the same instructions are used for regular memory accesses and mmio:

- The `kvm` mmu does *not* create a shadow page table translation when an mmio page is accessed
- Instead, the x86 emulator executes the faulting instruction, yielding the direction, size, address, and value of the transfer.

In `kvm`, I/O virtualization is performed by userspace. All pio and mmio accesses are forwarded to userspace, which feeds them into a *device model* in order to simulate their behavior, and possibly trigger real I/O such as transmitting a packet on an Ethernet interface. `kvm` also provides a mechanism for userspace to inject interrupts into the guest.

5.2 Host-Initiated Virtual Interrupts

`kvm` also provides interrupt injection facilities to userspace. Means exist to determine when the guest is ready to accept an interrupt, for example, the interrupt flag must be set, and to actually inject the interrupt when the guest is ready. This allows `kvm` to emulate the APIC/PIC/IOAPIC complex found on x86-based systems.

5.3 Virtualizing Framebuffers

An important category of memory-mapped I/O devices are framebuffers, or graphics adapters. These have characteristics that are quite distinct from other typical mmio devices:

- *Bandwidth* – framebuffers typically see very high bandwidth transfers. This is in contrast to typical devices which use mmio for control, but transfer the bulk of the data with direct memory access (*dma*).
- *Memory equivalence* – framebuffers are mostly just memory: reading from a framebuffers returns the data last written, and writing data does not cause an action to take place.

In order to efficiently support framebuffers, `kvm` allows mapping non-mmio memory at arbitrary addresses such as the pci region. Support is included for the VGA windows which allow physically aliasing memory regions, and for reporting changes in the content of the framebuffer so that the display window can be updated incrementally.

6 Linux Integration

Being tightly integrated into Linux confers some important benefits to `kvm`:

- On the *developer* level, there are many opportunities for reusing existing functionality within the kernel, for example, the scheduler, NUMA support, and high-resolution timers.
- On the *user* level, one can reuse the existing Linux process management infrastructure, e.g., `top(1)` to look at cpu usage and `taskset(1)` to pin virtual machines to specific cpus. Users can use `kill(1)` to pause or terminate their virtual machines.

7 Live Migration

One of the most compelling reasons to use virtualization is *live migration*, or the ability to transport a virtual machine from one host to another without interrupting guest execution for more than a few tens of milliseconds. This facility allows virtual machines to be relocated to different hosts to suit varying load and performance requirements.

Live migration works by copying guest memory to the target host in parallel with normal guest execution. If a guest page has been modified *after* it has been copied, it must be copied again. To that end, `kvm` provides a *dirty page log* facility, which provides userspace with a bitmap of modified pages since the last call. Internally, `kvm` maps guest pages as read-only, and only maps them for write after the first write access, which provides a hook point to update the bitmap.

Live migration is an iterative process: as each pass copies memory to the remote host, the guest generates more memory to copy. In order to ensure that the process converges, we set the following termination criteria:

- Two, not necessarily consecutive, passes were made which had an *increase* in the amount of memory copied compared to previous pass, or,
- Thirty iterations have elapsed.

8 Future Directions

While already quite usable for many workloads, many things remain to be done for `kvm`. Here we describe the major features missing; some of them are already work-in-progress.

8.1 Guest SMP Support

Demanding workloads require multiple processing cores, and virtualization workloads are no exception. While `kvm` readily supports SMP hosts, it does not yet support SMP guests.

In the same way that a virtual machine maps to a host process under `kvm`, a virtual cpu in an SMP guest maps to a host thread. This keeps the simplicity of the `kvm` model and requires remarkably few changes to implement.

8.2 Paravirtualization

I/O is notoriously slow in virtualization solutions. This is because emulating an I/O access requires exiting guest mode, which is a fairly expensive operation compared to real hardware.

A common solution is to introduce paravirtualized devices, or virtual "hardware" that is explicitly designed for virtualized environments. Since it is designed with the performance characteristics of virtualization in mind, it can minimize the slow operations to improve performance.

8.3 Memory Management Integration

Linux provides a vast array of memory management features: demand paging, large pages (*hugetlbfs*), and memory-mapped files. We plan to allow a `kvm` guest address space to directly use these features; this can enable paging of idle guest memory to disk, or loading a guest memory image from disk by demand paging.

8.4 Scheduler Integration

Currently, the Linux scheduler has no knowledge that it is scheduling a virtual cpu instead of a regular thread. We plan to add this knowledge to the scheduler so that it can take into account the higher costs of moving a virtual cpu from one core to another, as compared to a regular task.

8.5 New Hardware Virtualization Features

Virtualization hardware is constantly being enhanced with new capabilities, for example, full mmu virtualization, a.k.a. *nested page tables* or *extended page tables*, or allowing a guest to securely access a physical device [VT-d]. We plan to integrate these features into `kvm` in order to gain the performance and functionality benefits.

8.6 Additional Architectures

`kvm` is currently only implemented for the `i386` and `x86-64` architectures. However, other architectures such as `powerpc` and `ia64` support virtualization, and `kvm` could be enhanced to support these architectures as well.

9 Conclusions

`kvm` brings an easy-to-use, fully featured integrated virtualization solution for Linux. Its simplicity makes extending it fairly easy, while its integration into Linux allows it to leverage the large Linux feature set and the tremendous pace at which Linux is evolving.

10 References

[qemu] Bellard, F. (2005). *Qemu, a Fast and Portable Dynamic Translator.* In Usenix annual technical conference.

[xen] Barham P., et al. *Xen and the art of virtualization.* In Proc. SOSP 2003. Bolton Landing, New York, U.S.A. Oct 19-22, 2003.

[Intel] Intel Corp. *IA-32 Intel® Architecture Software Developer's Manual, Volume 3B: System Programming Guide, Part 2.* Order number 25366919.

[AMD] AMD Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming.*

[VT-d] Abramson, D.; Jackson, J.; Muthrasanallur, S.; Neiger, G.; Regnier, G.; Sankaran, R.; Schoinas, I.; Uhlig, R.; Vembu, B.; Wiegert, J. *Intel® Virtualization Technology for Directed I/O.* Intel Technology Journal. `http://www.intel.com/technology/itj/2006/v10i3/` (August 2006).