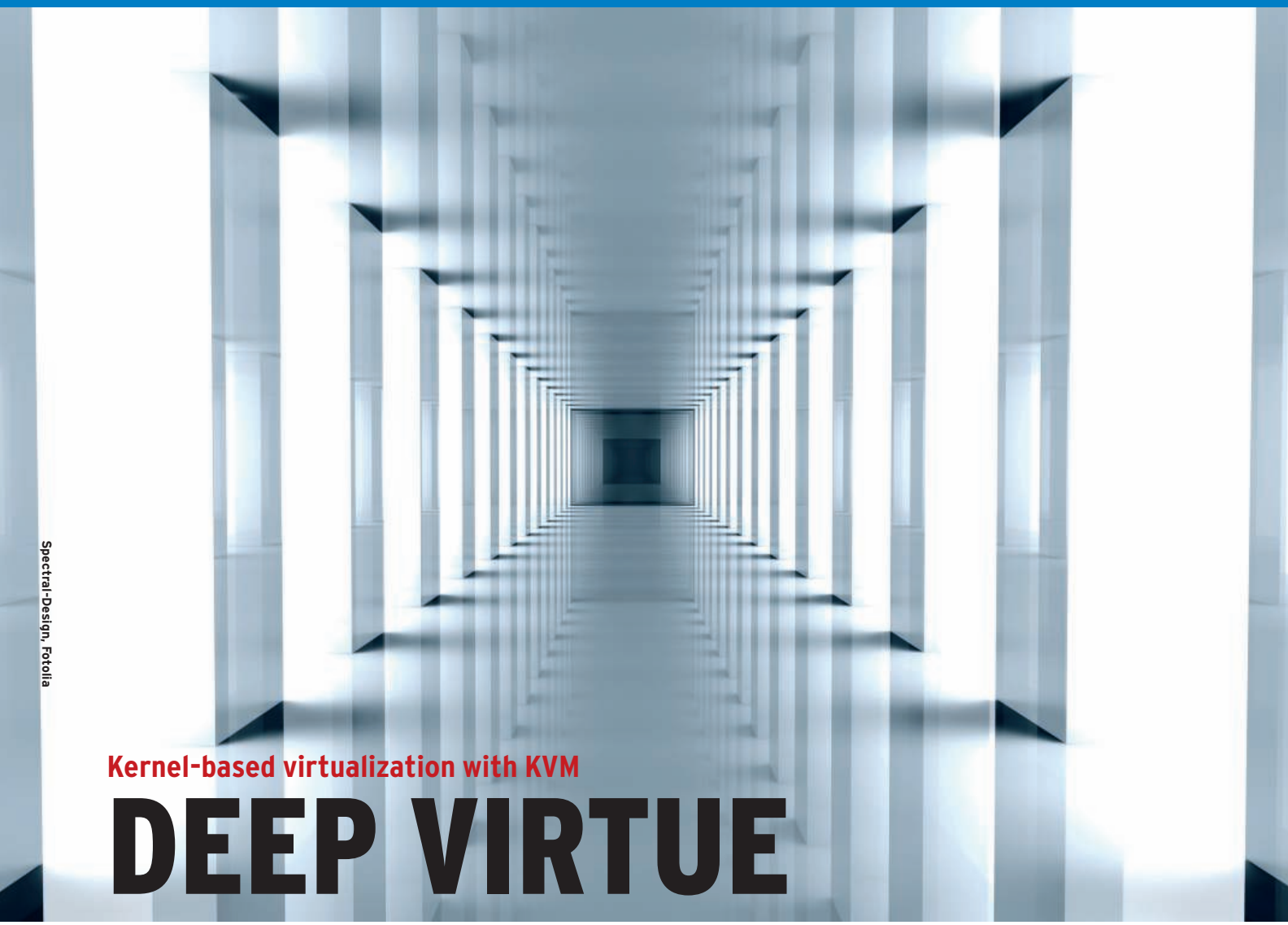

Spectral-Design, Fotolia

Kernel-based virtualization with KVM

DEEP VIRTUE

KVM brings the kernel into the virualization game. We'll explain why the Linux world is so interested in this promising virtualization alternative.

BY AMIT SHAH

In December 2006, Linus Torvalds announced that new versions of the Linux kernel would include the virtualization tool known as KVM (Kernel Virtual Machine Monitor). KVM appeared on the scene relatively recently, and its sudden rise to prominence speaks to the power of the kernel-based virtualization model. Kernel-based virtualization offers several potential advantages, including better performance and more uniform support for the complete Linux environment. This article shows how KVM works and helps you get started with setting up your own KVM-based virtual systems.

The KVM Way

In a common virtualization scenario, a component known as the *hypervisor* serves as an interface between the guest operating system and the host. The hypervisor sits on top of the host system, handling tasks such as scheduling and memory management for the guests.

KVM merges the hypervisor with the kernel, thus reducing redundancy and speeding up execution times. A KVM driver communicates with the kernel and acts as an interface for a userspace virtual machine. Scheduling of processes and memory management is handled through the kernel itself. A small Linux kernel module introduces the guest mode, sets up page tables for the guest, and emulates certain key instructions.

Current versions of KVM come with a modified version of the Qemu emulator, which manages I/O and operates as a virtual home for the guest system (Figure 1). The guest system runs within Qemu, and Qemu runs as an ordinary process in user space. The resulting environment is similar to the scenario depicted in Figure 2, in which several virtual machine processes run alongside other userspace tasks managed directly by the kernel. Each guest consists of two parts: the userspace part (Qemu) and the guest part (the guest itself). The guest physical memory is mapped in the task's virtual memory space, so guests can be swapped as well. Virtual processors within a virtual machine simply are threads in the host process.

This model fits nicely into the Unix mindset of doing one thing and doing it right. The KVM module is all about enabling the guest mode and handling virtualized accesses to registers. From a user's perspective, there's almost no difference between running a Qemu virtual machine with KVM disabled and running a virtual machine with KVM enabled, except, of course, the significant speed difference.

KVM follows the development and release philosophy Linux is built on: release early and often. The latest stable

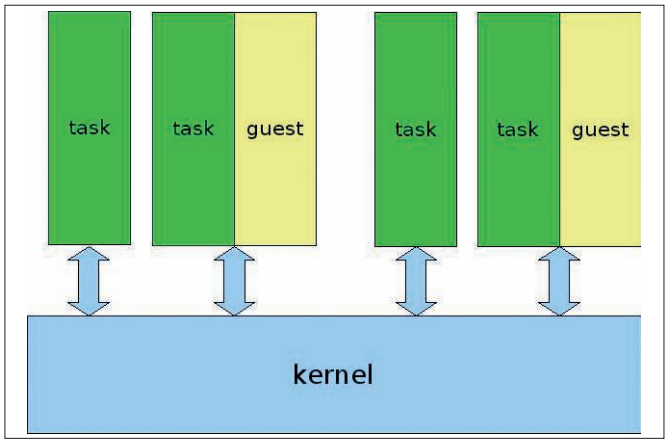

Figure 1: KVM comes with a modified version of the Qemu emulator.

release is part of Linux 2.6.x, with bug fixes going in 2.6.x.y. The KVM source code is maintained in a git tree. To get the latest KVM release or the latest git tree, head to the KVM wiki [1] for download details.

Using KVM

Since KVM only exploits the recent hardware advances, you should make sure you have a processor that supports virtualization extensions. To find out:

```
egrep '^flags.*(vmx|svm)' ➥
/proc/cpuinfo
```

If there's output, the necessary capability to run KVM exists on the CPU.

If you have the hardware support, you're halfway there. You now need to run a recent 2.6 Linux kernel. If you already run a recent Linux kernel with KVM either compiled in the kernel or compiled as modules, you can use it if you don't want to compile the modules yourself. However, the KVM project recommends you use the latest version from the website, as KVM continuously gets new features and bug fixes (not to mention new bugs, so you might get hurt from them too once in a while).

Download the KVM source from the KVM download page [2]. The tarball has two parts. The *kernel/* directory contains the sources for the kernel modules. The other files are the userspace portion, a slightly modified version of Qemu. If you download the KVM tarball and install it, you shouldn't have KVM compiled in the kernel; otherwise, the built module will fail to load.

Building the userspace utilities from the tarball requires a few libraries. The detailed list and instructions are available through the KVM wiki [3].

You'll need to use the GCC 3 compiler; part of the Qemu code isn't friends with GCC 4, the default compiler on recent Linux distros.

Once you have the kernel module and the userspace tools installed (by building or installing from your distribution packages), the first thing you will need to do is create a file that will hold the guest OS. Creating such a file is easy:

```
$ qemu-img create -f qcow ➥
debian-etch.img 10G
```

This will create a 10GB file called *debian-etch.img* in the qcow format. A few other file formats are supported, each with advantages and disadvantages. See the Qemu documentation.

Once an image file is created, you're ready to install a guest OS within it. First, insert the KVM kernel modules in the kernel if they have been compiled as modules.

```
$ sudo modprobe kvm
$ sudo modprobe kvm-intel
OR
sudo modprobe kvm-amd
```

```
$ qemu-system-x86_64 -boot d ➥
-cdrom /images/debian-etch.iso ➥
-hda debian-etch.img
```

This command starts a VM session. The window displays *QEMU/KVM* in its titlebar, signifying that KVM has been enabled. Once the install finishes, you can run the guest with

```
$ qemu-system-x86_64 ➥
debian-etch.img
```

You can also pass the *-m* parameter to set the amount of RAM the VM gets. The default value is 128MB. Recent KVM releases have support for swapping guest memory, so the RAM allocated to the guest isn't pinned down on the host.

You may occasionally run into some bugs running VMs with KVM. The output in the host kernel logs will help you search for similar problems reported earlier and any solutions that might be available. Upgrading to the latest KVM release might fix the problem.

In case you don't find a solution, running the VM by passing the *-no-kvm* command line to Qemu will start Qemu without KVM support. If this doesn't solve the problem, it means the problem lies in Qemu and not with KVM. Another thing to try is to pass the *-no-kvm-irqchip* parameter while starting a VM. You can also ask the friendly KVM mailing list [4].

Qemu Monitor

The Qemu monitor is entered with the key combination Ctrl + Alt + 2 when the

Types of Virtual Machine Monitors

The various virtual machine solutions fall into a number of categories:

- **Native hypervisors**: A native hypervisor is associated with an operating system. A complete software-based implementation will need a scheduler, a memory management subsystem, and an I/O device model to export to the guest OS. Examples are VMWare ESX server, Xen, KVM, and IBM mainframes. In IBM mainframes, the virutal machine monitor is an integral part of the architecture.
- **Containers**: In this type of virtualization, the guest OS and the host OS share the same kernel. Different namespaces are allocated for different guests. For example, the process identifiers, file descriptors, etc., are "virtualized" in the sense that a PID obtained for a process in the guest OS will only be valid within that guest. The guest can have a different userland (e.g., a different distribution) from the host. Examples are OpenVZ, FreeVPS, and Linux-Vserver.
- **Emulation**: Each and every instruction in the guest is emulated. It is possible to run code compiled for different architectures on a computer. For example, you can run ARM code on a PowerPC machine. Examples are Qemu and Pearpc.

Qemu window is selected. The monitor gives access to some debugging commands and some commands that can help you inspect the state of the VM. For example, *info registers* shows the contents of the registers of the virtual CPU. You can also attach USB devices to a VM using commands in the Qemu monitor.

Migration of VMs

Migrating virtual machines is very important for load-balancing and reducing downtime during upgrades. The migration process entails moving a guest from one physical machine to another. The advantage of the KVM approach is that guests are not involved in the migration. Also, you don't need special components to tunnel a migration through an SSH session or compress the image being migrated. You can even pass the image through a program before it's transmitted to the target machine. Unless specific hardware or host-specific features are enabled, the migration can occur between any two machines. Moreover, stopped guests can be migrated as well as live guests. The migration facility is within Qemu, so no kernel-side changes are needed for enabling it. The device state-sync for achieving migration and the VM state are seamlessly provided and managed within user space.

On the target machine, you can run Qemu with the same command-line options used for the virtual machine on the source machine, with additional parameters for migration-specific commands:

```
$ qemu-system-x86 -incoming ➥
<protocol:params>
```

For example:

```
$qemu-system-x86 -m 512 ➥
-hda /images/a.img ➥
-incoming stdio
```

On the source machine, start migration with the *migrate* Qemu monitor command

```
(qemu) migrate ➥
<migration-protocol:params>
```

such as (on the source Qemu monitor):

```
(qemu) migrate ➥
tcp://dst-ip:dst-port
```

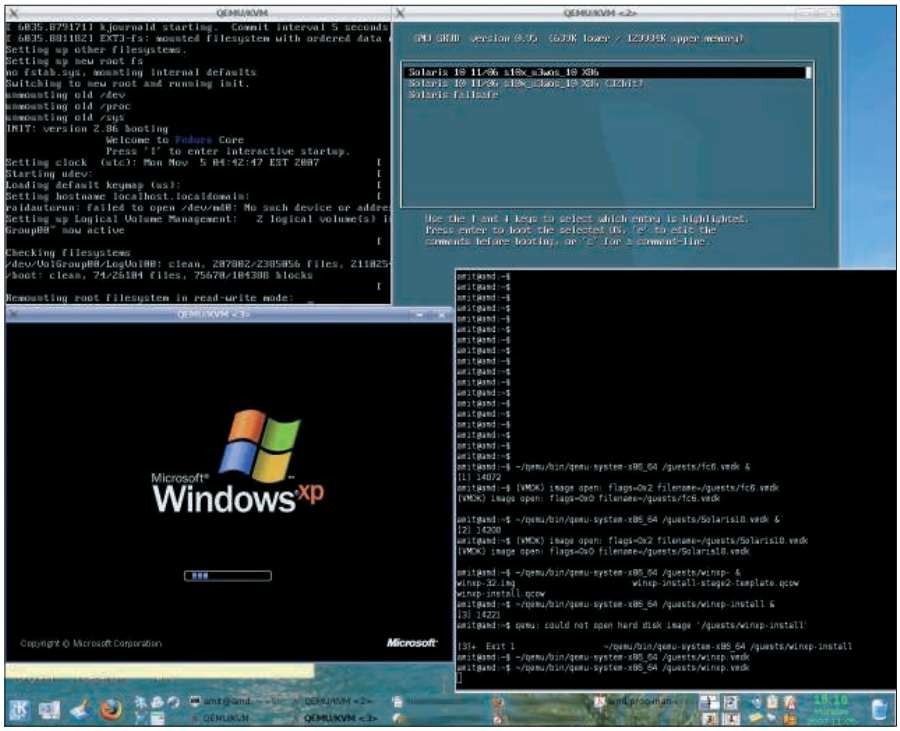

Figure 2: A virtual machine process runs alongside other userspace tasks and is managed directly by the kernel.

The target Qemu migration command-line parameter is:

```
-incoming tcp://0:port
```

If the source Qemu monitor command is

```
(qemu) migrate ssh://dst-ip
```

the target Qemu migration command-line parameter is:

```
-incoming ssh://0
```

You can use similar command-line options to achieve gzip compression or gpg encryption, or even to pass the data through a script before sending it.

Advantages of the KVM Approach

The KVM approach offers several advantages. You can reuse all the existing software and infrastructure, and you don't need to learn new commands. For example, *kill* and *top* work as expected on the guest task of the host system.

KVM was originally designed to support x86 hosts, and the focus was on full virtualization (no modifications to guest OS) with no modifications to the host kernel. However, as KVM started gaining developers and interesting use cases, developers began working on porting KVM to other architectures. Paravirtualization support is also under development. If a guest can communicate with the host, activities such as network activity or disk I/O can speed up. Also, modifications to the host operating system (Linux) that will improve scheduling and swapping have been proposed and accepted.

KVM seamlessly works across all machine types – servers, desktops, laptops, and embedded boards – and you can use the same management tools and infrastructure Linux uses. The KVM system integrates with the Linux scheduler, IO stack, and all available filesystems. Other benefits include live migration and support for NUMA and 4096-processor machines. If you are looking for an efficient virtualization alternative that is well integrated with Linux, take the time to explore KVM. ■

INFO

[1] KVM wiki: *http://kvm.qumranet.com*
[2] KVM download page: *http://kvm.qumranet.com/kvmwiki/Downloads*
[3] KVM HOWTO: *http://kvm.qumranet.com/kvmwiki/HOWTO*
[4] KVM mailing list: *kvm-devel@lists.sourceforge.net*