Kernel-based Virtual Machine Technology

● Yasunori Goto

The kernel-based virtual machine (KVM) has been attracting attention in recent years for application to open source server virtualization. Since its introduction in October 2006, the simplicity of this idea has aroused the interest of Linux kernel developers, who have helped to rapidly extend KVM functionality. KVM is now formally supported by Red Hat Enterprise Linux and has been supported by Fujitsu since version 6 of Red Hat Enterprise Linux. This paper begins by explaining the mechanism of KVM and then describes its components. Next it introduces hardware and software support for KVM virtualization and briefly describes some enhancements planned by Fujitsu to enable KVM to be used in mission-critical work.

1. Introduction

Virtualization technology for servers in the x86 family of CPUs has been attracting attention in recent years for various reasons. Server virtualization itself is a technology that has been around for some time, and the provision of Intel Virtualization Technology (Intel VT)[1] and AMD-Virtualization (AMD-V)[2] virtualization support functions in Intel and AMD CPUs has provided developers with an environment that can achieve virtualization relatively inexpensively at practical levels of performance using x86 hardware. Various types of software for achieving server virtualization have also appeared.

Amidst these developments, the kernel-based virtual machine (KVM)[3] has rapidly come to the forefront as a server virtualization function provided as open source software (OSS). KVM, which was designed assuming use of the Intel VT-x[note 1)] or AMD-V function, achieves virtualization using a relatively simple structure. The idea of implementing KVM was first announced in October 2006 by Avi Kivity, then of Qumranet, an Israeli firm.[4] The support of Linux kernel developers, attracted by KVM's simple design approach, was soon obtained, and KVM functionality was rapidly extended. KVM is now formally supported by Red Hat and has been supported by Fujitsu since version 6 of Red Hat Enterprise Linux (RHEL6).

This paper describes the simple mechanism of KVM and provides a brief introduction to software and hardware functions supporting KVM. It assumes Intel VT-x CPU functions.

2. KVM mechanism

To provide the reader with a basic understanding of the KVM mechanism, this section first describes Intel VT-x and then

note 1) Intel VT is the generic name of Intel's virtualization support function consisting of Intel VT-x for x86 processors like Core 2 Duo and Intel Xeon, Intel VT-i for Itanium 2, and Intel VT-d supporting I/O virtualization.

explains the quick emulator called "QEMU," an important OSS component of KVM.

2.1 Intel VT-x and sensitive instructions

Assuming that KVM would make use of the Intel VT-x function, KVM designers implemented KVM as a function in the Linux kernel.[note 2] The following provides an overview of Intel VT-x to facilitate a basic understanding of KVM.

Intel VT-x can be viewed as a "function that switches processing to the hypervisor on detecting the execution of a sensitive instruction by the CPU." As shown in **Figure 1**, hypervisor is a control program for operating VMs (guest systems) on a physical machine. Two types of sensitive instructions are defined.[5]

1) "Control-sensitive instructions" attempt to change the state of system resources.
2) "Behavior-sensitive instructions" operate in accordance with the state of the system resources.

Conceptually, control-sensitive instructions executed by a program on a VM affect the operation of the physical machine, and behavior-sensitive instructions executed by a program on a VM reveal that they were executed on a VM since the results differ from those when they are executed on the physical machine

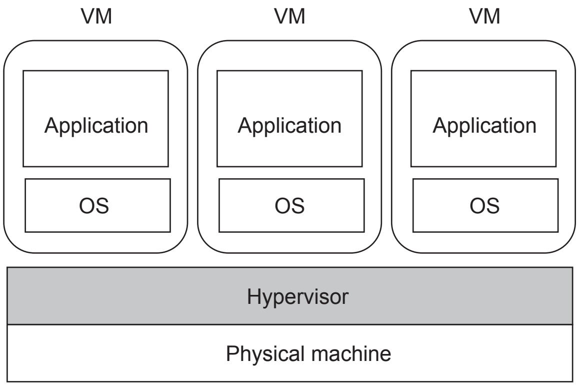


Figure 1
Hypervisor and virtual machines.

If a program attempts to execute these instructions on a guest system without any intervention, it will cause serious problems for the hypervisor and guest system. It is therefore necessary for the CPU to detect that the execution of a sensitive instruction is beginning and to direct the hypervisor to execute that instruction on behalf of the program.

However, x86 CPUs were not designed with the need for virtualization in mind, so there are sensitive instructions that the CPU cannot detect as such when a guest system attempts to execute them. As a result, the hypervisor cannot execute such instructions on behalf of the guest system. Intel VT-x was developed in response to this problem. It adds new execution modes to the processor and switches between these modes once the CPU detects such an instruction so that the hypervisor can execute that instruction on behalf of the initiating program.

Specifically, Intel VT-x adds two program execution modes: VMX root operation and VMX non-root operation, where VMX stands for "virtual machine extension." As shown in **Figure 2**, VMX non-root operation mode is the execution mode for guest systems. If an

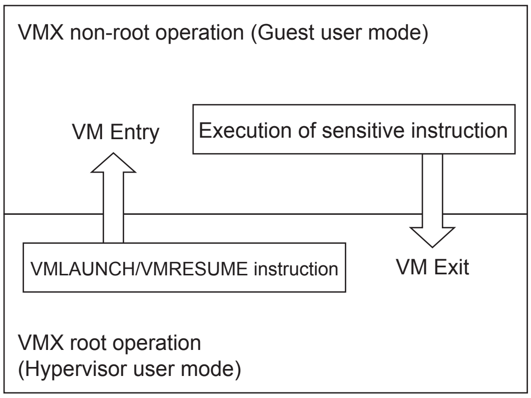

VMX non-root operation (Guest user mode)
VM Entry
Execution of sensitive instruction
VMLAUNCH/VMRESUME instruction
VM Exit
VMX root operation
(Hypervisor user mode)

Figure 2
Relationship between hypervisor and program execution modes of Intel VT-x.

[note 2] The advantage of this is that a kernel function can easily support hypervisor operation given the high affinity between the kernel and hypervisor.

attempt is made to execute a sensitive instruction when in this mode, the CPU detects the attempt and switches execution to VMX root operation mode, which is the execution mode for hypervisor use. This transition, called “VM Exit,” signals the transfer of control to the hypervisor, enabling it to execute the sensitive instruction on behalf of the guest system.

Two new instructions introduced by Intel VT-x–VMLAUNCH and VMRESUME–enable switching to VMX non-root operation mode, which is called “VM Entry.”

The main role of KVM is the handling of VM Exits and the execution of VM Entry instructions. KVM is implemented as a module in the Linux kernel.

2.2 KVM and QEMU

The KVM kernel module cannot, by itself, create a VM. To do so, it must use QEMU, a user-space process.[6)]

QEMU is inherently a hardware emulator. It is provided as OSS for emulating standard x86 personal computers (PCs) and other architectures. It existed before the release of KVM and can operate without KVM.

Given that QEMU is a software-based emulator, it interprets and executes CPU instructions one at a time in software, which means its performance is limited. However, it is possible to greatly improve QEMU performance while also achieving a VM function if three conditions are met.

1) A target instruction can be directly executed by the CPU.
2) That instruction can be given without modification to the CPU for direct execution in VMX non-root operation mode.
3) A target instruction that cannot be directly executed can be identified and given to QEMU for emulator processing.

The development of KVM was based on this idea. Application of this idea enables the creation of VMs while maximizing the use of existing OSS resources with minimal modifications. Much support has been received from Linux kernel developers for this reason.

The QEMU/KVM execution flow is shown in **Figure 3**. First, a file named /dev/kvm is created by the KVM kernel module (step 0 in the figure). This file enables QEMU to convey a variety of requests to the KVM kernel module to execute hypervisor functions. When QEMU starts up to execute a guest system, it repeatedly makes ioctl() system calls specifying this special file (or file descriptors derived from it). When it is time to begin executing the guest system, QEMU again calls ioctl() to instruct the KVM kernel module to start up the guest system (step 1). The kernel module, in turn, performs a VM Entry (step 2) and begins executing the guest system. Later, when the guest system is about to execute a sensitive instruction, a VM Exit is performed (step 3), and KVM identifies the reason for the exit. If QEMU intervention is needed to execute an I/O task or another task, control is transferred to the QEMU process (step 4), and QEMU executes the task. On execution completion, QEMU again makes an ioctl() system call and requests the KVM to continue guest processing (i.e., execution flow returns to step 1). This

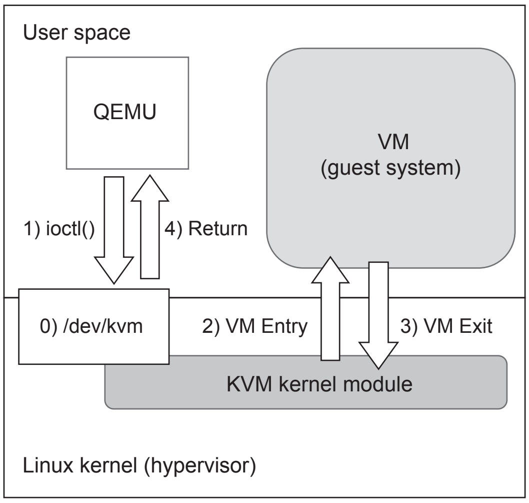


Figure 3
QEMU/KVM execution flow.

QEMU/KVM flow is basically repeated during the emulation of a VM.

QEMU/KVM thus has a relatively simple structure.

1) Implementation of a KVM kernel module transforms the Linux kernel into a hypervisor.
2) There is one QEMU process for each guest system. When multiple guest systems are running, the same number of QEMU processes are running.
3) QEMU is a multi-thread program, and one virtual CPU (VCPU) of a guest system corresponds to one QEMU thread. Steps 1–4 in Figure 3 are performed in units of threads.
4) QEMU threads are treated like ordinary user processes from the viewpoint of the Linux kernel. Scheduling for the thread corresponding to a virtual CPU of the guest system, for example, is governed by the Linux kernel scheduler in the same way as other process threads.

3. KVM Components

This section introduces KVM-related components other than QEMU and describes the overall KVM structure.

QEMU itself is launched by entering a simple command of the same name via a character user interface (CUI). If the process status (ps) command is entered while a guest system is running, the status of QEMU execution is displayed, as illustrated by this example.

```
[goto@lifua ˜]$ ps auxw |grep qemu
qemu        ......        /usr/bin/qemu-kvm -S -M
 fedora-13 -enable-kvm -m 1024 -smp 1, sockets=1,
cores=1, threads=1 .....
 -drive file=/home/goto/kvm_image/fedora13.img,
 .....   -device   rtl8139,   vlan=0,   id=net0,
mac=52:54:00:65:03:a0 ......
```

The command options “-m 1024” and “-smp 1” indicate the memory capacity and the number of CPUs, respectively, of the guest system. Though the example shown above contains about one-third the actual number of lines, all QEMU settings, such as device settings, have been passed as QEMU-command parameters.

As might be expected, it is not necessarily easy to specify all of these options directly with the QEMU command. For this reason, a graphical user interface (GUI), referred to as “virt-manager,”[7] has been prepared for Red Hat Enterprise Linux (RHEL) as well as for other operating systems to enable the user to operate and manage one or more guest systems. **Figure 4** shows a screen shot when multiple guest systems are executing. The window at the upper-left is the virt-manager screen, and the windows at the lower-left and right are guest-system screens, indicating that two guest systems are running. This screen shot shows an example of executing VMs on Fedora Linux distribution; the corresponding display for RHEL6 may differ slightly.

In addition to such GUIs, virt-manager can also take the form of a CUI called “virsh,” which can also be used to operate guest systems.

The overall structure of KVM, from the GUI to the Linux kernel, includes five main components.

1) virt-manager

A GUI/CUI user interface used for managing VMs; it calls VM functions using libvirt, which is described next.

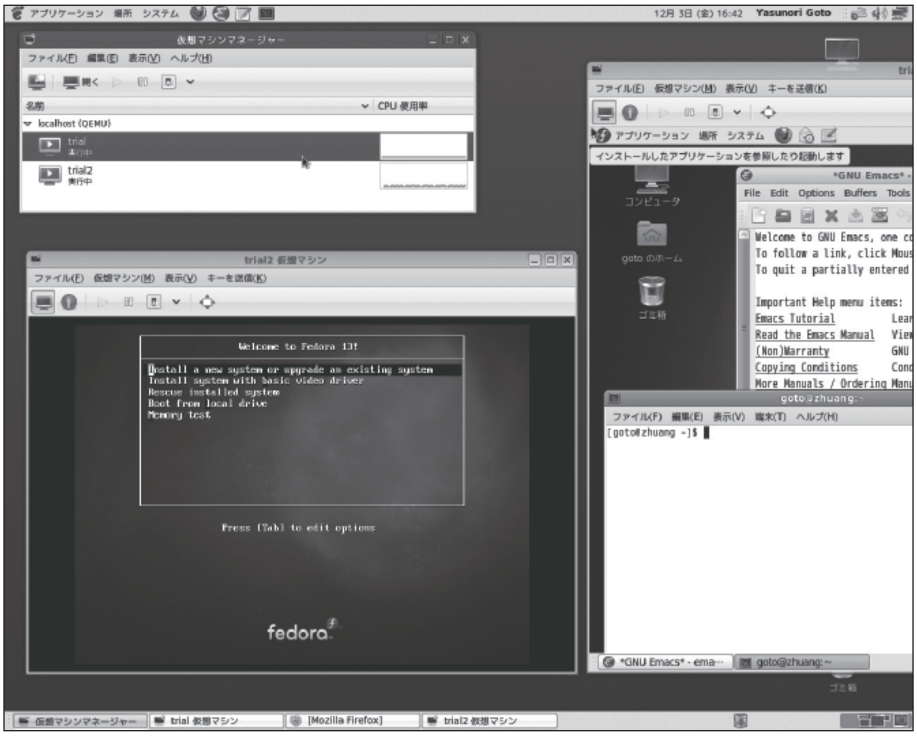
2) libvirt

A tool-and-interface library[8] common to server virtualization software supporting Xen, VMware ESX/GSX, and, of course, QEMU/KVM

3) QEMU

An emulator that interacts with the KVM kernel module and executes many types of guest-system processing such as I/O; one QEMU process corresponds to one guest system.

4) KVM kernel module


アプリケーション 場所 システム
12月 3日 (金) 16:42 Yasunori Goto
仮想マシンマネージャー
ファイル(F) 編集(E) 表示(V) ヘルプ(H)
開く
名前
CPU 使用率
localhost (QEMU)
trial
実行中
trial2
実行中
trial2 仮想マシン
ファイル(F) 仮想マシン(M) 表示(V) キーを送信(K)
fedora
tri
ファイル(F) 仮想マシン(M) 表示(V) キーを送信(K)
アプリケーション 場所 システム
インストールしたアプリケーションを参照したり起動します
コンピュータ
goto のホーム
ゴミ箱
GNU Emacs -
File Edit Options Buffers Tools
goto@zhuang:~
ファイル(F) 編集(E) 表示(V) 端末(T) ヘルプ(H)
GNU Emacs - ema···
goto@zhuang:~
ゴミ箱
仮想マシンマネージャー
trial 仮想マシン
[Mozilla Firefox]
trial2 仮想マシン

```
Welcome to Fedora 13!

Install a new system or upgrade an existing system
Install system with basic video driver
Rescue installed system
Boot from local drive
Memory test

Press [Tab] to edit options
```

```
Welcome to GNU Emacs, one co
To follow a link, click Mous
To quit a partially entered

Important Help menu items:
Emacs Tutorial          Lear
Read the Emacs Manual   View
(Non)Warranty           GNU
Copying Conditions      Cond
More Manuals / Ordering Manu
```

```
[goto@zhuang ~]$ 
```

Figure 4
Screen when guest systems are executing.

In a narrow sense, KVM is a Linux kernel module; it handles VM Exits from guest systems and executes VM Entry instructions.

5) Linux kernel

Since QEMU runs as an ordinary process, scheduling of the corresponding guest system is handled by the Linux kernel itself.

If we revise Figure 3 to include virt-manager and the other components, we get the overall KVM configuration shown in **Figure 5**. All of the components are OSS.

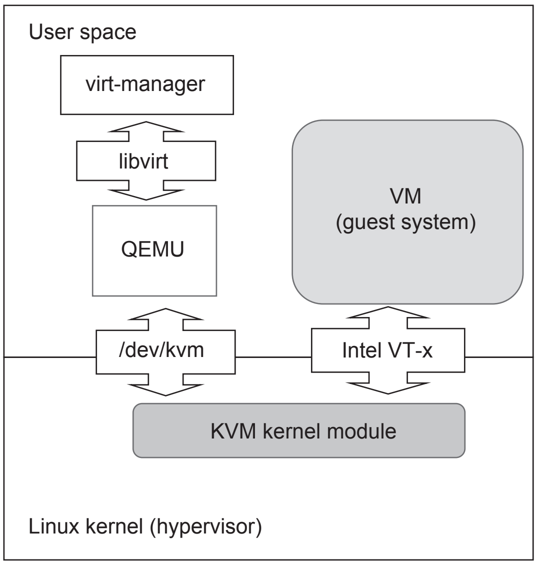


Figure 5
Whole image of KVM.

4. KVM hardware and software support functions

The matter of most concern in server virtualization is performance. Ideally, the performance of a VM is no worse than that of the physical machine. To this end, there is a variety of hardware and software support functions available for virtualization in addition to the basic KVM mechanism described above. The basic aim is to use them to improve KVM performance. This section introduces several of these functions.

1) Extended Page Table (EPT)

The EPT function extends the address

conversion mechanism (MMU: memory management unit) provided by the CPU. Since a VM has a two-part structure in the form of a hypervisor and a guest system, a conventional MMU structure, designed without a VM in mind, cannot be applied as is.

Before EPT was available, it was necessary to perform address conversion processing in software using a technique called "shadow paging." The development of EPT has enabled "physical addresses" in the VM to be converted by the CPU into actual physical addresses in the physical machine, making software conversion unnecessary. The KVM makes full use of EPT, resulting in significantly improved VM performance.

2) VT-d

VT-d is an address conversion mechanism for I/O devices (IOMMU). It provides a memory address conversion table (having a data structure identical to that of the MMU page table) for each device function. With VT-d, memory addresses on the guest system can be specified as data transfer destinations from a device; i.e., data can be directly transferred to a guest OS.

VT-d is a chip-set function and must be supported by firmware. Accordingly, if VT-d is supported by both firmware and the Linux kernel, the latter will recognize it and put it to use.

3) virtio

VM devices are usually created and processed by QEMU device emulation. The overhead for this emulation is high, however, so I/O performance is quite poor. To overcome this problem, a mechanism called "virtio" has been introduced. Virtio prepares a buffer that can be accessed from both a guest system and QEMU. Using this buffer, I/O processing for multiple items of data can be performed together, thereby reducing the overhead associated with QEMU emulation and achieving high-speed processing. This mechanism can be accessed from the guest system as a virtio peripheral component interconnect (PCI) device.

The beneficial effect of virtio can be obtained and high-speed I/O achieved by implementing a virtio driver in each guest OS (there is, of course, a virtio driver for Windows).

4) Kernel Samepage Merging (KSM)

Different VMs on the same physical machine can sometimes be executing the same OS and the same applications. In such a situation, there is a high possibility that they will have memory areas with the same content. Consolidating such areas into one memory area would reduce memory usage.

With this in mind, the KSM function[9] was added for KVM in the Linux kernel. The KSM function uses the "ksmd" kernel thread to periodically monitor process-memory usage and to automatically merge duplicate pages into a common page.

Ideally, all memory pages would be compared to identify duplicate memory content, but continuously comparing all pages in use by all system processes would be extremely inefficient. For this reason, Linux makes it possible to specify which memory area is to be a candidate for KSM by using the third parameter of the madvise() system call, i.e., the "advice" parameter. QEMU uses this function when allocating memory for a guest so that the user can immediately enjoy the benefits of the KSM function. In RHEL6 as well, this function is effective in the system's initial state.

5. Future KVM enhancements

The goal is to enable KVM to be used in mission-critical operations and applications, but there are still many things that need to be enhanced to improve functionality and quality. The following introduces several items scheduled for development at Fujitsu with the aim of enhancing the KVM function.

1) Enhancement of libvirt functionality and quality

The libvirt function is still far from

complete compared to the mature level of the KVM kernel module and QEMU. It is not, as a result, at a state where the power of KVM can be fully demonstrated, and it has not yet reached a stable level of quality. Enhancing the functionality and quality of the libvirt function is therefore an urgent development item.

2) Enhancement of resource management functions

It is expected that KVM will link with Cgroup[10], resource management functions that came into use with RHEL6. At present, the Cgroup functions can be used for various types of resource allocation such as allocating a certain number of actual CPUs to a general process. However, when considering the linking of KVM and Cgroup functions from the viewpoint of controlling guest systems, there are still problems that need to be dealt with such as inadequate I/O control. There are therefore plans to enhance the Cgroup functions.

3) Machine-check support

Information on the occurrence of uncorrectable faults such as multi-bit errors in the hardware of the physical machine must be passed on to guest systems. A basic framework for this has been incorporated in KVM, but machine-check support specifically for KVM needs to be enhanced to enable memory-error recovery to be performed on a guest OS.

6. Conclusion

This paper described the basic mechanism of the kernel-based virtual machine (KVM) and server-virtualization-support functions that are standard in Red Hat Enterprise Linux 6. It also introduced Fujitsu's efforts in developing and enhancing KVM functions. Fujitsu expects these development activities to make KVM an important component of mission critical systems for customers in the future.

Yasunori Goto
Fujitsu Ltd.
Mr. Goto is engaged in the development and support of Linux kernel/KVM.

References

1) Intel: Virtualization (Intel VT). *http://www.intel.com/technology/virtualization/technology.htm?iid=tech_vt+tech*

2) AMD Virtualization. *http://sites.amd.com/us/business/it-solutions/virtualization/Pages/virtualization.aspx*

3) Main Page—KVM. *http://www.linux-kvm.org/page/Main_Page*

4) [PATCH 0/7] KVM: Kernel-based Virtual Machine. *http://marc.info/?l=linux-kernel&m=116126591619631&w=2*

5) Gerald J. Popek, Robert P. Goldberg: Formal Requirements for Virtualizable Third Generation Architectures, (1974). *http://portal.acm.org/ft_gateway.cfm?id=361073&type=pdf&CFID=5009197&CFTOKEN=14838199*

6) About—QEMU. *http://wiki.qemu.org/Main_Page*

7) Virtual Machine Manager. *http://virt-manager.et.redhat.com/*

8) Libvirt: The virtualization API. *http://libvirt.org/*

9) [PATCH 0/4] ksm - dynamic page sharing driver for linux. *http://marc.info/?l=linux-kernel&m=122640987623679&w=2*

10) H. Ishii: Fujitsu's Activities for Improving Linux as Primary OS for PRIMEQUEST. *Fujitsu Sci. Tech. J.,* Vol. 47, No. 2, pp. 239–246 (2011).