KVM - The kernel-based virtual machine

Timo Hirt
timohirt@gmx.de

13. Februar 2010

Abstract

Virtualization has been introduced in the 1960s, when computing systems were large and expensive to operate. It allowed users to share the computing resources of those machines by running their tasks in parallel. Nowadays virtualization is a hype topic again and there are a plenty of scopes of application.

Virtualization allows to run several virtual machines on top of a host. This work focuses on the system-virtualization approach, whereby a whole operating system runs in an isolated virtual machine. A Virtual Machine Monitor is used to operate these virtual machines. It represents a small layer of indirection between the physical hardware and the virtual machines. Scheduling of computing time and memory management is also part of the virtual machine monitors responsibilities. In recent years Intel and AMD released hardware support for implementing these, which promises better performance.

KVM is the first virtualization solution that has been integrated into the vanilla linux kernel. This work presents its brief history. Followed by more technical considerations of its architecture and execution model.

Contents

1 Introduction

Virtualization is not a new technology. In the **1960s** computing systems were as large as a room and very expensive to operate. In those days only one application could be executed on one piece of hardware at a particular time. Then time-sharing had been introduced to execute several applications simultaneously. One major drawback of this approach was the lack of isolation of the running applications. If application A caused a hardware error all other running applications were affected. To isolate these, virtualization provided several isolated environments to ran them into.
In the **1970s** hardware architectures became virtualization aware. IBM mainframes allowed the administrators to partition the real hardware and provide isolated environments for each application.
In the **1980s**, as the x86 architecture arose and the prices of hardware felt, it became affordable to run one computer per application. Also operating systems supported multi tasking and there was no need for time-sharing any more. As a consequence virtualization became history.
In **the last couple of years** virtualization experienced a comeback. Intel and AMD extended the IA32 instructon set of x86 processors to support virtualization. Since these are the big players on the CPU market, nearly any recent PC and server supports virtualization.

Today, virtualization is mainly used for consolidation. There are many types of consolidation and the following examples should give a basic idea about it.
A lot of servers are running at a very low load but still consuming a huge amount of energy. *Server consolidation* means work-load optimization of these servers by running each of them as a *Virtual Machine (VM)* on virtualization hosts. When contention is low these VMs are dynamically migrated to fewer virtualization host and shut down the others to reduce energy consumption and lower costs. If the load increases and more hosts are needed to fulfill server level objectives, these are started again and some VMs are migrated onto them.
Another example is *application consolidation*, where virtualization is used to replace the old hardware of a legacy system. It helps to provide an environment

which mimics the old hardware and runs the legacy system.
Sandboxing is another purpose of virtualization. It is mainly used to increase security by running potentially insecure applications inside a VM. So an application runs in its isolated environment, while specialists can observe its behaviour. Thus malware and other malicious software could be found before it's deployed on a machine with access to the network of a company.

There are various techniques to provide and operate VMs, one of those are *Virtual Machine Monitors (VMM)*. Such a VMM represents a software layer of indirection, running on top of the hardware. It operates all VMs running upon it. In section 2 we cover this approach and today's hardware support for implementing a VMM. Based on that in section 3 we present KVM as a linux extension that turns it into a VMM.

2 System-Virtualization

2.1 Definition

Since virtualization is a settled topic, there are several definitions on it. The following is a general definition of virtualization given by [CB05]:

> "Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and many others"

This means, that virtualization uses techniques to abstract from the real hardware and provides isolated environments, so called Virtual Machines. These are capable to run various applications or even a whole operating system. A goal not mentioned in the definition is to have nearly to native performance for running VMs. This is a very important point, because the users always want to get the most out of their hardware. Most of them are not willing to introduce virtualization

technology, if a huge amount of CPU power is wasted by managing VMs.

As well as virtualization in general, system virtualization is well defined too:

> "A system VM provides a complete environment in which an operating system and many processes, possibly belonging to multiple users, can coexist." [Smi05]

The complete environment, in this case, means an environment that provides usual hardware like ethernet controllers, CPUs or hard disk drives to an operating system (OS) which runs inside of it. A server with real hardware attached to it commonly runs several VM's. Such a server is called *virtualization host* and the VM's running on top of it are called *guests*. The OS that runs inside a guest is called *guest OS*.

For this work, unless otherwise specified, by virtualization we understand system-virtualization.

2.2 Virtual Machine Monitor

A Virtual Machine Monitor, also called hypervisor, is a piece of software or hardware that runs on top of the hosts hardware (Figure 1). It represents a layer of indirection between the physical hardware and the VMs running one layer above. All guests are controlled and monitored by the VMM. It provides tools to the users to manage them. These tools allow to do several operations like starting or stoping a guest or migrating VMs between hosts.

A VM usually has at least one virtual CPU. The VMM maps the virtual CPU(s) of all actually running VMs to the physical CPU(s) of the host. Hence, there are usually more VMs running on a host than physical CPUs are attached to it, causes the need of some kind of scheduling. Therefore a VMM uses a certain scheduling mechanism to assign a certain share of the physical CPUs to each virtual CPU.

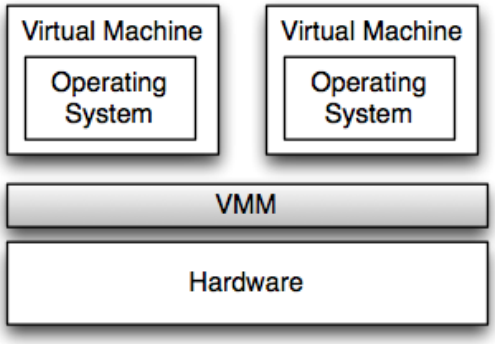

Figure 1: **VMM Architecture** [Qum06]

A VMM has to deal with memory management, also. It maps an amount of physical memory into the VMs address space and also has to handle fragmentation of memory and swapping. Since some VMs need more memory than others, the amount of assigned memory is defined and often dynamically adjusted by using the management tools.

Usually, the VMs don't have access to the physical hardware and don't even know about it either. Only if direct access is desired, devices may be passed through directly. For running legacy software this may be a point. But in more common scenarios the VMM provides virtual I/O devices like network cards, hard disks and cd drives. Since a VMM provides different VMs mostly with same hardware, it is much easier to migrate them between hosts running the same VMM. The drivers for the virtual I/O devices need to be installed only once in this case.

2.3 Hardware Support

To implement a Virtual Machine Monitor on a x86 architecture, hardware assistance is needed. The privilege levels implemented by the CPU to restrict tasks that processes can do, are one aspect. Another one is the memory management that is emulated by the VMM which tends to be inefficient. Hardware support could lead to an increased performance of the virtual machines by supporting a VMM.

2.3.1 Privilege Levels

The most modern operating systems don't allow applications to execute certain operations. Only the OS may load drivers or access the hardware directly, for example. To restrict all running applications to only a subset of the resources, the OS and the CPU conspire using privilege levels.

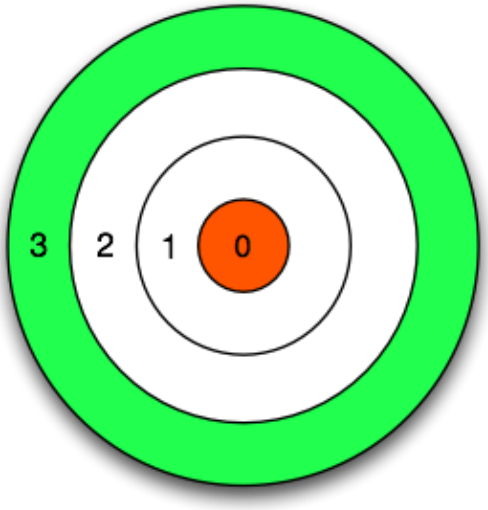

Figure 2: **CPU privilege levels**

As described in [Dua08] a x86 CPU runs in a specific privileged level at any given time. Figure 2 shows these levels as rings. Ring 0 is the most privileged and ring 3 is the least privileged. The resources that are protected through the rings are: memory, I/O ports and CPU instructions. The operating system typically runs in ring 0. It needs the most privileged level to do resource management and provide access to the hardware. All the applications run in ring 3. Ring 1 and 2 are widely unused. From a OSs point of view ring 0 is called kernel-mode and ring 3 user-mode.

As mentioned in section 2.2 the VMM needs to access the memory, CPU and I/O devices of the host. Since only code running in ring 0 is allowed to perform these

operations, it needs to run in the most privileged ring, next to the kernel. An operating system installed in a VM also expects to access all the resources and in order of that running in ring 0 like the VMM does. Due to the fact that only one kernel can run in ring 0 at the same time, the guest OSs have to run in another ring with less privileges or have to be modified to run in user-mode.

Intel and AMD realized that this is a major challenge of virtualization on the x86 architecture. So they introduced Intel VT and AMD SVM as an extension of the IA-32 instruction set for better support of virtualization. These extensions allow the VMM to run a guest OS that expects to run in kernel-mode, in a lower privileged ring.

2.3.2 Memory Management

In order to run several VMs on top of a server, a multiple of the amount of memory that is attached to a common server is needed. Since each VM runs an entire operating system and applications on that, it is recommended to assign as much memory to a VM as a comparable physical machine would have. The VMM splits the physical memory of the host into contiguous blocks of fixed length and maps it into the address space provided to a VM.

Most modern systems are using virtual memory management. This technique allows to provide the previously mentioned contiguous blocks of memory to a VM, although it is fragmented all over the physical memory or even partially stored on the hard disk. In this case it has to be copied back to memory by the virtual memory management first, when accessed. Since a VM is unaware of the physical address of its address space, it can't figure out whether parts of its virtual memory has to be copied or not. To achieve that, the VMM holds a so called *shadow page table* that stores the physical location of the virtual memory of all VMs. Thus, any time a VM writes to its memory, the operation has to be intercepted to keep the shadow pages up to date. When a swapped address is accessed the VMM first uses the virtual memory management to restore it.

With the introduction of Intel's Extended Paging Tables (EPT) and AMD's Nested Paging Tables (NPT) a VMM can use hardware support for the translation between virtual and physical memory. This reduces the overhead of holding shadow pages and increases the performance of a VMM [Bha09].

2.4 Virtualization techniques

With the problems and solutions mentioned in the previous section in mind, we now take a look at two techniques to realize system virtualization.

2.4.1 Paravirtualization

The paravirtualization approach allows each guest to run a full operating system. But these do not run in ring 0. Due to that all the privileged instructions can't be executed by a guest. In order of that, modifications to the guest operating systems are required to implement an interface. This is used by the VMM to take over control and handle the restricted instructions for the VM. The paravirtualization approach promises nearly to native performance but lacks in the support for closed source operating systems [NAL+06]. To apply the mentioned modifications, the source code of the kernel of an operating system has to be patched. Thus, running Microsoft Windows in a VM is impossible using paravirtualization.

2.4.2 Fullvirtualization

This approach allows to operate several operating systems on top of a hosting system, each running into its own isolated VM. The VMM uses hardware support as described in section 2.3.1 to operate these, which allows to run the guest operating systems without modifications. The VMM provides I/O devices for each VM, which is commonly done by emulating older hardware. This ensures that a guest OS has driver support for these devices. Because of the emulated parts fullvirtualization is not as fast as paravirtualization. But if one needs to run a closed source OSs, it is the only viable technique to do so.

3 KVM – Kernel-based Virtual Machine

KVM has been initially developed by Qumranet, a small company located in israel. Redhat acquired Qumranet in september 2008, when KVM became more production ready. They see KVM as the next generation of virtualization technology. Nowadays it is used as the default VMM in Redhat Enterprise Linux (RHEL) since version 5.4 and the Redhat Enterprise Virtualization for Servers[1].

Qumranet released the code of KVM to the open source community. Today, well known companies like IBM, Intel and AMD count to the list of contributors of the project. Since version 2.6.20 KVM is part of the vanilla linux kernel and thus available on the most linux-based operating systems with a newer kernel. Furthermore it benefits from the world class development of the open source operating system, because if linux gains better performance through new algorithms, drivers or whatsoever KVM also performs better.

KVM is a system-virtualization solution that uses fullvirtualization to run VMs. It has a small code base, since it was designed to leverage the facilities provided by hardware support for virtualization as described in section 2.3. KVM runs mainly on the x86 architecture, but IA64 and IBM s390 support was added recently.

3.1 Architecture

3.1.1 Linux as a VMM

Linux has all the mechanisms a VMM needs to operate several VMs. We already mentioned these mechanisms in section 2.2. So the developers didn't reinvent the wheel and added only few components to support virtualization. KVM is implemented as a kernel module that can be loaded to extend linux by these capabilities. Thus, linux is turned into a VMM, as shown in figure 3.

In a normal linux environment each process runs either in user-mode or in kernel-mode. KVM introduces a third mode, the guest-mode. Therefore it relies on a

[1] http://www.de.redhat.com/virtualization/rhev/server/

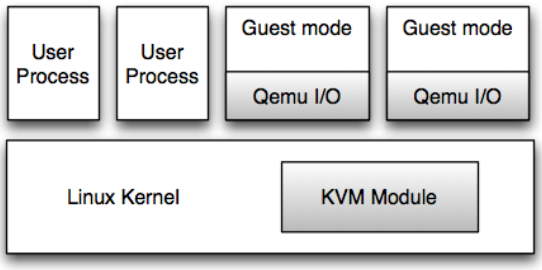

Figure 3: **KVM Architecture** [Qum06]

virtualization capable CPU with either Intel VT or AMD SVM extensions. A process in guest-mode has its own kernel-mode and user-mode. Thus, it is able to run an operating system. Such **processes are representing the VMs** running on a KVM host. In [Qum06] the author states what the modes are used for from a hosts point of view:

- user-mode: I/O when guest needs to access devices
- kernel-mode: switch into guest-mode and handle exits due to I/O operations
- guest-mode: execute guest code, which is the guest OS except I/O

3.1.2 Resource management

The KVM developers aimed to reuse as much code as possible. Due to that they mainly modified the linux **memory management**, to allow mapping physical memory into the VMs address space. Therefore they added shadow page tables, that were needed in the early days of x86 virtualization, when Intel and AMD had not released EPT respectively NPT yet. On May 2008 support for these technologies has been introduced.

In modern operating systems there are many more processes than CPUs available to run them. The **scheduler** of an operating system computes an order in that each process is assigned to one of the available CPUs. In this way, all running processes are share the computing time. Since the KVM developers wanted to

reuse most of the mechanisms of linux, they simply implemented each VM as a process, relying on its scheduler to assign computing power to the VMs.

3.1.3 The KVM control interface

Once the KVM kernel module has been loaded, the */dev/kvm* device node appears in the filesystem. This is a special device node that represents the interface of KVM. It allows to control the hypervisor through a set of *ioctls*. These are commonly used in certain operating systems as an interface for processes running in user-mode to communicate with a driver. The *ioctl()* system call allows to execute several operations to create new virtual machines, assign memory to a virtual machine, assign and start virtual CPUs.

3.1.4 Emulation of hardware

To provide hardware like hard disks, cd drives or network cards to the VMs, KVM uses a highly modified QEMU[2]. This is a so called platform virtualization tool, which allows to emulate an entire pc platform including graphics, networking, disk drives and many more. For each VM a QEMU process is started in user-mode and certain emulated devices are virtually attached to these. When a VM performs I/O operations, these are intercepted by KVM and redirected to the QEMU process regarding to the guest.

3.2 Execution-Model

Figure 4 depicts the execution model of KVM. This is a loop of actions used to operate the VMs. These actions are separated by the three modes we mentioned earlier in section 3.1.1.

In [KKL+07] Kivity et al. described the KVM execution model and stated which tasks are done in which mode:

- user-mode: The KVM module is called using *ioclt()* to execute guest code until I/O operations initiated by the guest or an external event occurs. Such

[2]http://www.qemu.org/

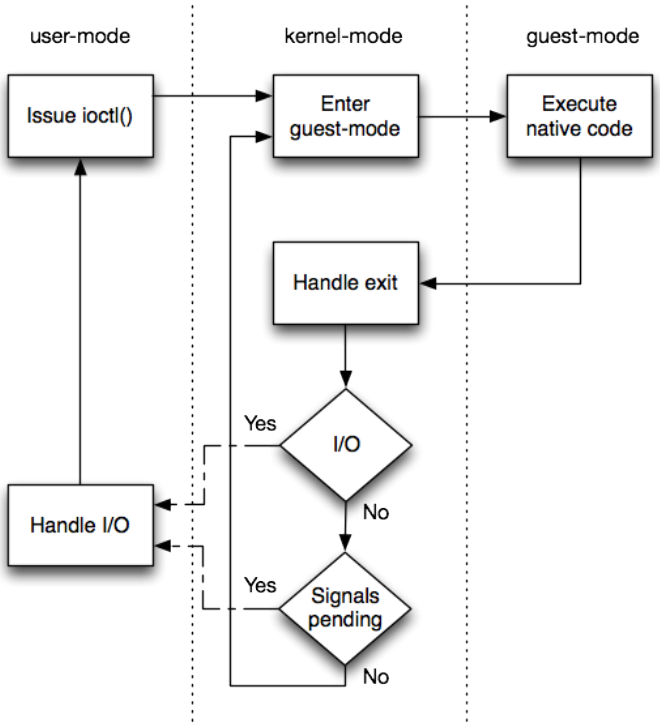

Figure 4: **KVM execution model** [KKL+07]

an event may be the arrival of a network package, which could be the reply of a network package sent by the host earlier. Such events are expressed as signals that leads to an interruption of guest code execution.

- kernel-mode: The kernel causes the hardware to execute guest code natively. If the processor exits the guest due to pending memory or I/O operations, the kernel performs the necessary tasks and resumes the flow of execution. If external events such as signals or I/O operations initiated by the guest exists, it exits to the user-mode.

- guest-mode: This is on the hardware level, where the extended instruction set of a virtualization capable CPU is used to execute the native code, until an instruction is called that needs assistance by KVM, a fault or an external interrupt.

While a VM runs, there are plenty of switches between these modes. From kernel-

mode to guest-mode switches and vice versa are very fast, because there is only native code that is executed on the underlying hardware. When I/O operations occur and the flow of execution switches to the user-mode, emulation of the virtual I/O devices comes into play. Thus, a lot of I/O exits and switches to user-mode are expected. Imagine an emulated hard disk and a guest reading certain blocks from it. Then QEMU emulates the operations by simulating the behaviour of the hard disk and the controller it is connected to. To perform the guests read operation, it reads the corresponding blocks from a large file and returns the data to the guest. Thus, user-mode emulated I/O tends to be a bottleneck which slows down the execution of a VM.

3.3 Paravirtual device drivers

With the support for the *virtio*[Rus08] paravirtual device model, KVM addresses the performance limitations by using QEMU emulated devices. Virtio is common framework to write VMM independent drivers promising bare-metal speed for these, since paravirtual devices attached to a VM are not emulated any more.

Instead, a backend for the paravirtual drivers is used to perform I/O operations either directly or through a user-mode backend. KVM uses QEMU as such a backend which handles I/O operations directly. Thus, the overhead to mimic the behaviour of a IDE hard disk is tremendously decreased to simply using kernel drivers to performing certain operations and responding.

4 Conclusion

Virtualization can be used for a plenty of scopes of application. Since CPU manufacturer introduced facilities to build VMMs more efficiently, those can be run on the popular and widespread x86 architecture. KVM is an open source virtualization solution that leverages the CPU facilities to operate VMs using system-virtualization. It allows to run various operating systems in several isolated virtual machines running on top of a host.

KVM is designed as an kernel module, once loaded it turns linux into an VMM. Since the developers didn't want to reinvent the wheel, KVM relies on the mechanisms of the kernel to schedule computing power and benefits from the of the box driver support. But the memory management has been extended to be capable to manage memory that is assigned to the address space of a VM.

Since emulated devices provided to the VMs does not perform well, the virtio device model is supported by KVM. It allows to heavily increase the I/O performance since paravirtual drivers allows to omit emulating certain devices.

References

[Bha09] Nikhil Bhatia. Performance Evaluation of Intel EPT Hardware Assist, 2009.

[CB05] Susanta Nanda Tzi-cker Chiueh and Stony Brook. A Survey on Virtualization Technologies. *RPE Report*, pages 1–42, 2005.

[Dua08] Gustavo Duarte. CPU Rings, Privilege, and Protection, 2008.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual machine monitor. In *Proceedings of the Linux Symposium*, Ottawa, 2007.

[NAL+06] Gil Neiger, Santoni Amy, Felix Leing, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(03), 2006.

[Qum06] Qumranet. KVM - Kernel-based Virtualiztion Machine White paper, 2006.

[Rus08] Rusty Russell. virtio: Towards a De-Facto Standard For Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[Smi05] J.E. Smith. The architecture of virtual machines. *Computer*, 38(5):32–38, Mai 2005.