Qumranet

KVM: Kernel-based Virtualization Driver

White Paper

Overview

The current interest in virtualization has led to the creation of several different hypervisors. Most of these, however, predate hardware-assisted x86 virtualization, and are therefore somewhat complex pieces of software. With the advent of Intel VT (Virtualization Technology) and AMD's SVM (Secure Virtual Machine), writing a hypervisor has become significantly easier and it is now possible to enjoy the benefits of virtualization while leveraging existing open source achievements to date.

Common Hypervisor Model

The common hypervisor model in use today consists of a software layer which multiplexes the hardware among several "guest" operating systems. The hypervisor performs basic scheduling and memory management, and typically delegates management and I/O functions to a special, privileged, guest.

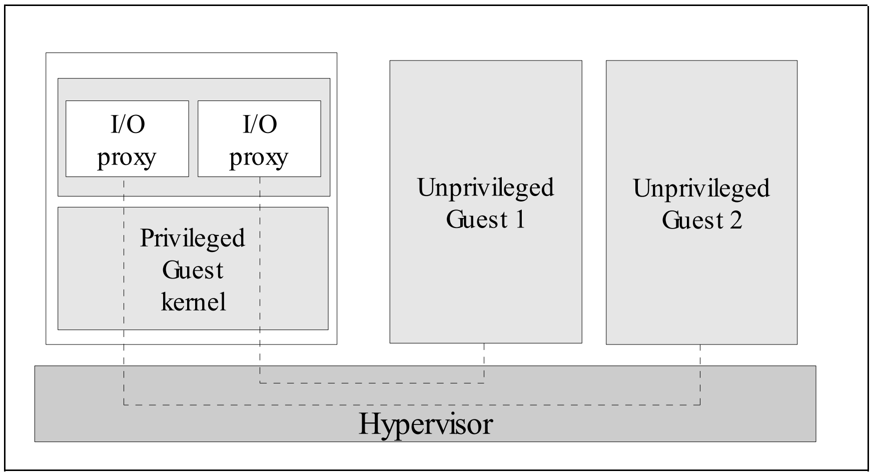

Figure 1 – Hypervisor Based Architecture

Today's hardware, however is becoming increasingly complex. The so-called "basic" scheduling operations have to take into account multiple hardware threads on a core, multiple cores on a socket, and multiple sockets on a system. Similarly, on-chip memory controllers require that memory management take into effect the Non-Uniform Memory Access (NUMA) characteristics of a system.

While great effort is invested into adding these capabilities to hypervisors, we already have a mature scheduler and memory management system that handles these issues very well – the Linux kernel.

Linux as a Hypervisor

By adding virtualization capabilities to a standard Linux kernel, we can enjoy all the fine-tuning work that has gone (and is going) into the kernel, and bring that benefit into a virtualized environment. Under this model, every virtual machine is a regular Linux process scheduled by the standard Linux scheduler. Its memory is allocated by the Linux memory allocator, with its knowledge of NUMA and integration into the scheduler.

A normal Linux process has two modes of execution: kernel and user. Kvm adds a third mode: guest mode (which has its own kernel and user modes, but these do not interest the hypervisor at all).

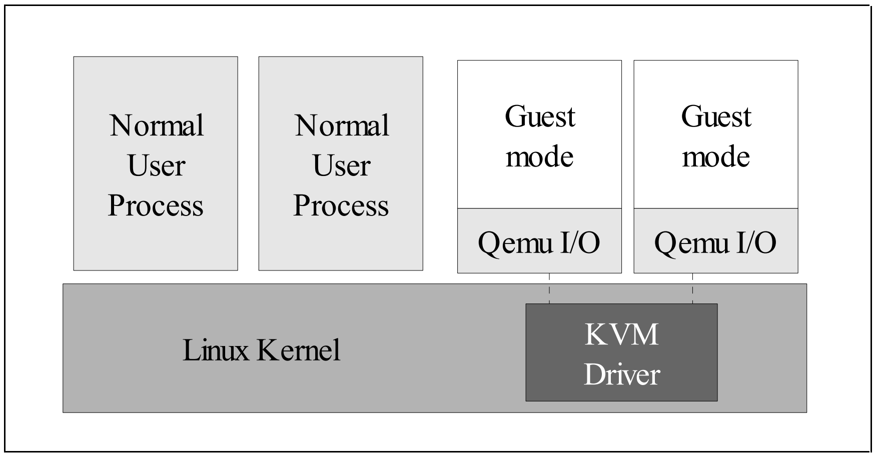

Figure 2 – kvm Based Architecture

The division of labor among the different modes is:

- Guest mode: execute non-I/O guest code
- Kernel mode: switch into guest mode, and handle any exits from guest mode due to I/O or special instructions.
- User mode: perform I/O on behalf of the guest.

By integrating into the kernel, the kvm 'hypervisor' automatically tracks the latest hardware and scalability features without additional effort.

A Minimal System

One of the advantages of the traditional hypervisor model is that it is a minimal system (see *Figure 2* above), consisting of only a few hundred thousands lines of code. However, this view does not take into account the privileged guest. This guest has access to all system memory, either through hypercalls or by programming the DMA hardware. A failure of the privileged guest is not recoverable as the hypervisor is not able to restart it if it fails.

A kvm based system's privilege footprint is truly minimal: only the host kernel plus a few thousand lines of the kernel mode driver have unlimited hardware access.

kvm Components

The simplicity of kvm is exemplified by its structure; there are two components:

- A device driver for managing the virtualization hardware; this driver exposes its capabilities via a character device `/dev/kvm`
- A user-space component for emulating PC hardware; this is a lightly modified `qemu` process

The modified `qemu` process `mmap()`s the guest's physical memory and calls the kernel mode driver to execute in guest mode.

The I/O model is directly derived from `qemu`'s, with support for copy-on-write disk images and other `qemu` features.

I/O Performance

The kvm model has some performance advantage when performing I/O on behalf of the guest[1]. Consider the sequence of events in a privileged-guest system:

- *Guest issues an I/O instruction*
- *Hypervisor traps the I/O instruction and forwards it to the privileged guest (some instructions may be handled internally)*
- *The hypervisor's scheduler has to schedule the privileged guest*
- *A guest switch is performed into the privileged guest*
- *The privileged guest's interrupt handler is invoked, which causes an I/O process to be scheduled using the privileged guest's scheduler*
- *A process context switch is performed*
- *The I/O process initiates the I/O on behalf of the guest*
- *The I/O process signals (through the privileged guest kernel and hypervisor) that the I/O initiation is complete*
- *The hypervisor switches back into the original guest*

1 We acknowledge the fact the mentioned performance benefit is of unknown measure at this point, no comparative benchmarks were ran to quantify it. This should be viewed qualitatively.

- *The hypervisor resumes the guest code*

Same sequence on a kvm host:

- *Guest issues an I/O instruction*
- *Host kernel traps the instruction and exits into the VM's userspace (some instructions may be handled in kernel mode)*
- *The VM's user-space initiates the I/O on behalf of the guest*
- *The VM's user-space returns to the kernel*
- *The kernel resumes guest code*

Management

Since a virtual machine is simply a process, all of the standard Linux process management tools apply: one can destroy, pause, and resume a virtual machine with the `kill` command (or even using Ctrl-C and similar keyboard shortcuts) and view resource usage with `top`. Permissions are handled by the normal Linux method: the virtual machine belongs to the user who started it (which need not be `root`; all that is required is access to `/dev/kvm`), and all accesses are verified by the kernel.

This allows system administrators to manage virtual machines with existing tools, allowing systems to be virtualized incrementally.

Conclusions

Leveraging new silicon capabilities, the kvm model introduces an approach to virtualization that is fully aligned with Linux architecture and all of its latest achievements. Furthermore, integrating the hypervisor capabilities into a host Linux kernel as a loadable module can simplify management and improve performance in virtualized environments, while minimizing impact on existing systems.