Anatomy of a Linux hypervisor

An introduction to KVM and Lguest

Skill Level: Intermediate

M. Tim Jones (mtj@mtjones.com)
Independent Author
Emulex Corp.

31 May 2009

One of the most important modern innovations of Linux® is its transformation into a *hypervisor* (or, an operating system for other operating systems). A number of hypervisor solutions have appeared that use Linux as the core. This article explores the ideas behind the hypervisor and two particular hypervisors that use Linux as the platform (KVM and Lguest).

Hypervisors do for operating systems what operating systems roughly do for processes. They provide isolated virtual hardware platforms for execution that in turn provide the illusion of full access to the underlying machine. But not all hypervisors are the same, which is a good thing, because Linux is about flexibility and choice. This article begins with a quick introduction to virtualization and hypervisors, then explores a couple of Linux-based hypervisors.

Virtualization and hypervisors

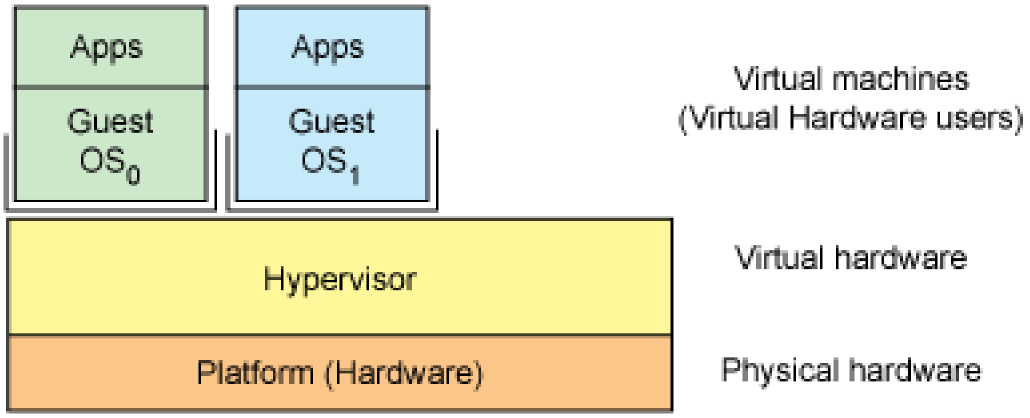
Read more by Tim Jones on developerWorks

- Tim's *Anatomy of...* articles
- All of Tim's articles on developerWorks

Let's first spend a little time understanding why virtualization is important and the role that hypervisors play. (And for more information on both topics, see the Resources section.)

Virtualization, in the context of this article, is the process of hiding the underlying physical hardware in a way that makes it transparently usable and shareable by multiple operating systems. This architecture is more popularly known as *platform virtualization.* In a typical layered architecture, the layer that provides for the platform virtualization is called the *hypervisor* (sometimes called the *virtual machine monitor,* or VMM). Each instance of a guest operating system is called a *virtual machine* (VM), because to these VMs, the hardware is virtualized to appear as dedicated to them. A simple illustration of this layered architecture is shown in Figure 1.

Figure 1. Simple layered architecture showing the virtualization of common hardware



The benefits of platform virtualization are many. But one interesting statistic reported by the U.S. Environmental Protection Agency (EPA) stood out. The EPA study on server and data center energy efficiency found that only around 5% of server capacity was actually used. The rest of the time, the server was dormant. Virtualizing platforms on a single server can improve server utilization, but the benefits of reducing server count are a force multiplier. With reduced servers comes reduced real estate, power consumption, cooling (less energy costs), and management costs. Less hardware also means improved reliability. All in all, platform virtualization brings not only technical advantages but cost and energy advantages, as well.

As you see in Figure 1, the hypervisor is the layer of software that provides the virtualization of the underlying machine (in some cases, with processor support). Not all virtualization solutions are equal, and you can learn more about the various styles of virtualization in Resources. Continuing with the processes theme, operating systems virtualize access to the underlying resources of the machine to processes. Hypervisors do the same thing, but instead of processes, they accomplish this task for entire guest operating systems.

Hypervisor classifications

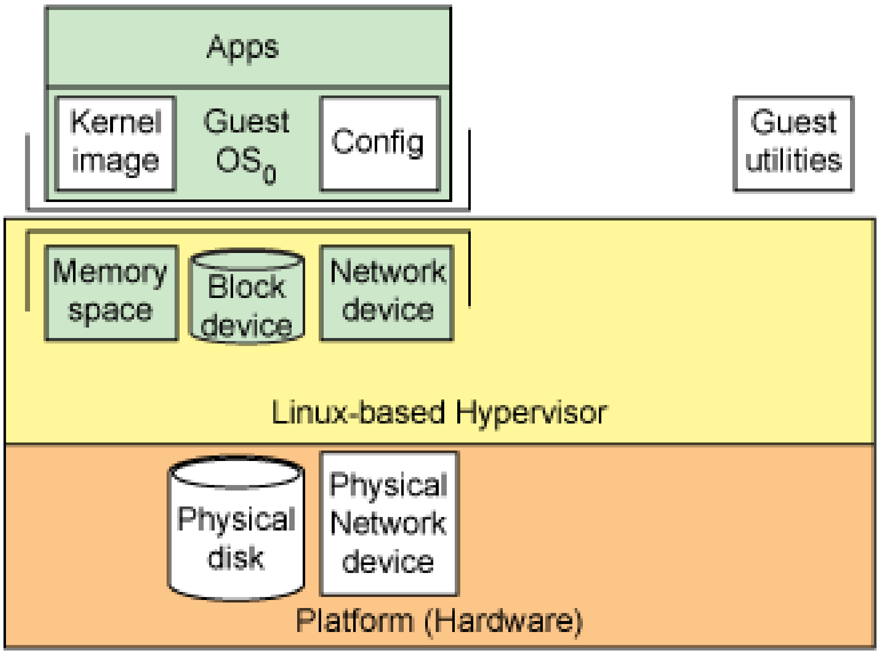
Hypervisors can be classified into two distinct types. The first, type 1 hypervisors, are those that natively run on the bare-metal hardware. The second, type 2, are hypervisors that execute in the context of another operating system (that runs on the bare metal). Examples of type 1 hypervisors include Kernel-based Virtual Machine (KVM—itself an operating system-based hypervisor). Examples of type 2 hypervisors include QEMU and WINE.

Elements of a hypervisor

So a hypervisor (regardless of the type) is just a layered application that abstracts the machine hardware from its guests. In this way, each guest sees a VM instead of the real hardware. Let's now look generically at the internals of a hypervisor and also its presentation to the VMs (guest operating systems).

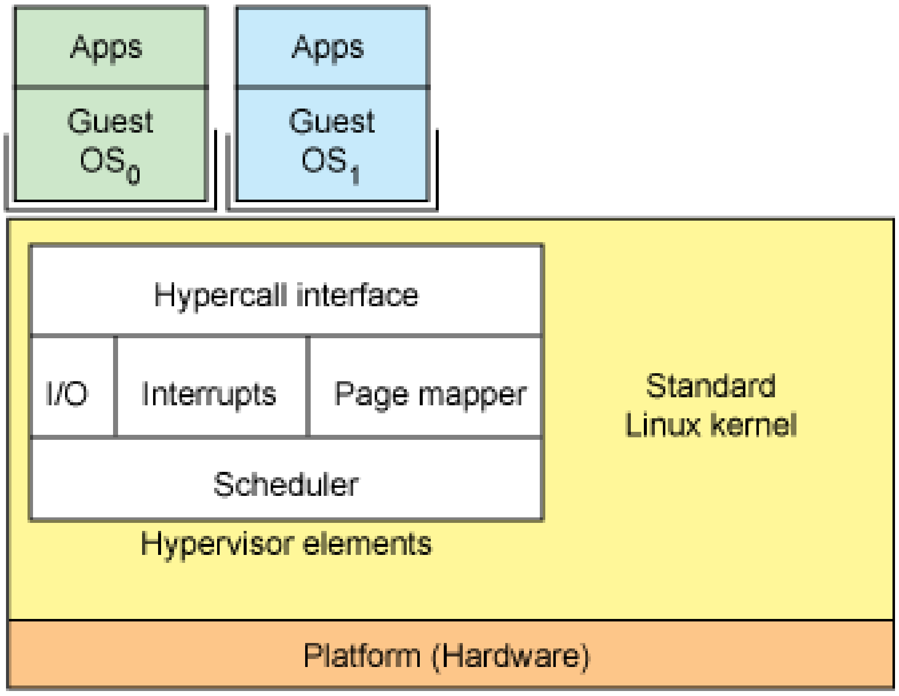
At a high level, the hypervisor requires a small number of items to boot a guest operating system: a kernel image to boot, a configuration (such as IP addresses and quantity of memory to use), a disk, and a network device. The disk and network device commonly map into the machine's physical disk and network device (as shown in Figure 2). Finally, a set of guest tools is necessary to launch a guest and subsequently manage it.

Figure 2. Minimal mapping of resources in a hypothetical hypervisor



A simplified hypervisor architecture then implements the glue that allows a guest operating system to be run concurrently with the host operating system. This functionality requires a few specific elements, shown in Figure 3. First, similar to system calls that bridge user-space applications with kernel functions, a hypercall layer is commonly available that allows guests to make requests of the host operating system. Input/output (I/O) can be virtualized in the kernel or assisted by code in the guest operating system. Interrupts must be handled uniquely by the hypervisor to deal with real interrupts or to route interrupts for virtual devices to the guest operating system. The hypervisor must also handle traps or exceptions that occur within a guest. (After all, a fault in a guest should halt the guest but not the hypervisor or other guests.) A core element of the hypervisor is a *page mapper,* which points the hardware to the pages for the particular operating system (guest or hypervisor). Finally, a high-level scheduler is necessary to transfer control between the hypervisor and guest operating systems (and back).

Figure 3. Simplified view of a Linux-based hypervisor



Linux hypervisors

This article explores two Linux-based hypervisor solutions. The first—KVM—was the first hypervisor module to be integrated into the Linux kernel, implementing full virtualization, and the second—Lguest—is an experimental hypervisor that provides paravirtualization in a surprisingly small number of changes.

KVM

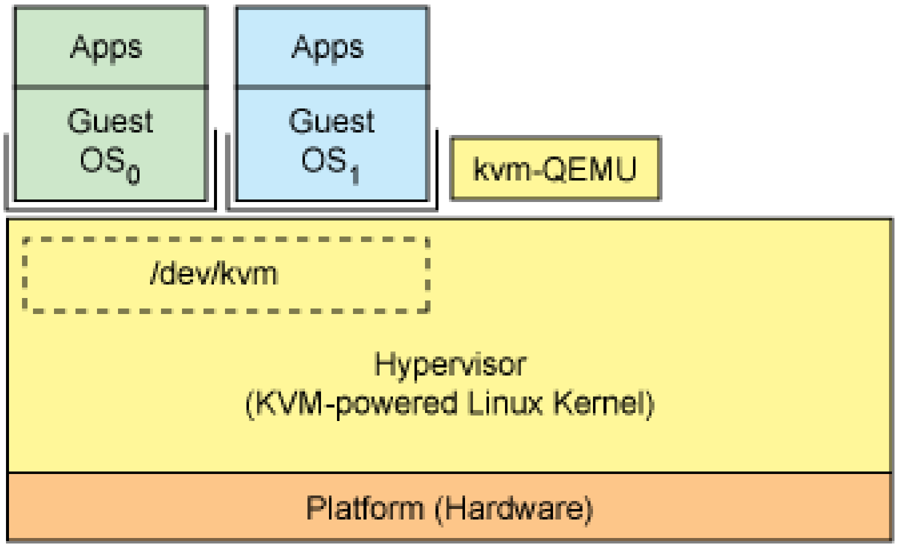
KVM is a kernel-resident virtualization infrastructure for Linux on x86 hardware. KVM was the first hypervisor to become part of the native Linux kernel (2.6.20) and was developed and is maintained by Avi Kivity through the Qumranet startup, now owned by Red Hat.

This hypervisor provides x86 virtualization, with ports to the PowerPC® and IA64 in process. Additionally, KVM has recently added support for symmetrical multiprocessing (SMP) hosts (and guests) and supports enterprise-level features such as live migration (to allow guest operating systems to migrate between physical servers).

KVM is implemented as a kernel module, allowing Linux to become a hypervisor simply by loading a module. KVM provides full virtualization on hardware platforms that provide hypervisor instruction support (such as the Intel® Virtualization Technology [Intel VT] or AMD Virtualization [AMD-V] offerings). KVM also supports paravirtualized guests, including Linux and Windows®.

This technology is implemented as two components. The first is the KVM-loadable module that, when installed in the Linux kernel, provides management of the virtualization hardware, exposing its capabilities through the /proc file system (see Figure 4). The second component provides for PC platform emulation, which is provided by a modified version of QEMU. QEMU executes as a user-space process, coordinating with the kernel for guest operating system requests.

Figure 4. High-level view of the KVM hypervisor



When a new operating system is booted on KVM (through a utility called `kvm`), it becomes a process of the host operating system and therefore scheduleable like any other process. But unlike traditional processes in Linux, the guest operating system is identified by the hypervisor as being in the "guest" mode (independent of the kernel and user modes).

Each guest operating system is mapped through the `/dev/kvm` device, having its own virtual address space that is mapped into the host kernel's physical address space. As previously mentioned, KVM uses the underlying hardware's virtualization support to provide full (native) virtualization. I/O requests are mapped through the host kernel to the QEMU process that executes on the host (hypervisor).

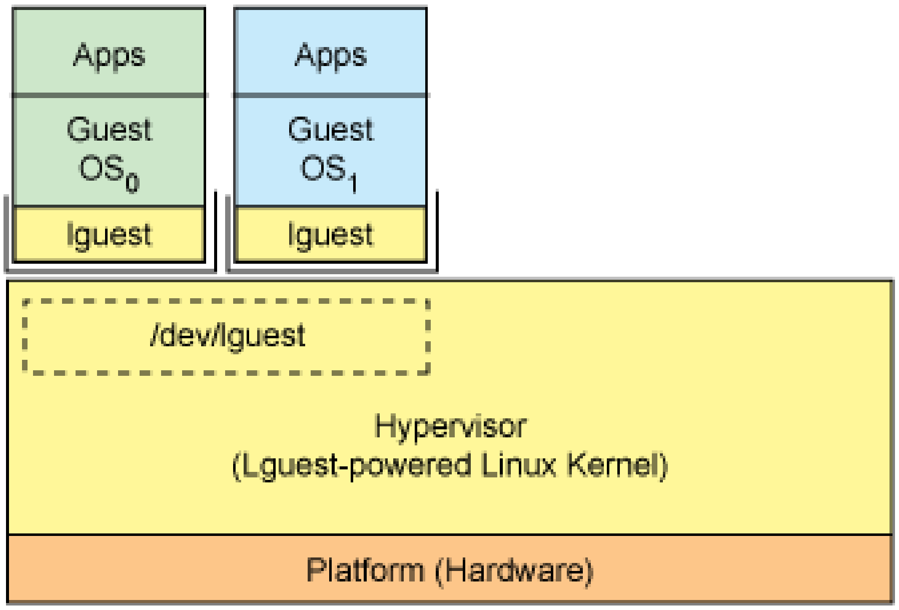
KVM operates in the context of Linux as the host but supports a large number of guest operating systems, given underlying hardware virtualization support. You can find a list of the supported guests in Resources.

Lguest (previously lhype)

The Lguest hypervisor, developed by Rusty Russell of IBM in Australia, takes a decidedly different approach to virtualization. Instead of providing full virtualization support to run arbitrary operating systems, Lguest provides lightweight paravirtualization for Lguest-enabled x86 Linux guests (otherwise called *Linux-on-Linux virtualization*). This means that guest operating systems know that they're being virtualized, and this knowledge comes with performance enhancements. However, Lguest provides reasonable performance without the need for QEMU providing platform virtualization (as is the case for KVM). The Lguest approach also simplifies the overall code requirements, requiring only a thin layer in the guest and also in the host operating system. Let's now explore these changes and review the high-level architecture of an Lguest environment.

As shown in Figure 5, the guest operating system includes a thin layer of Lguest code (by definition, *paravirtualization*). This code provides a number of services. At the highest level, there's code to determine whether the kernel being booted is being virtualized. There's also an abstraction layer (implemented through `paravirt_ops`) to route privileged operations to the host operating system through hypercalls. For example, the guest cannot disable interrupts, so these requests are performed in the host operating system. You'll also find a bus that implements a device abstraction for guests as well as a set of simple drivers implementing a console, virtual block driver, and virtual network driver (which permits communication with other guests).

Figure 5. Decomposition of the Lguest approach to x86 paravirtualization



The kernel side of things is implemented as a loadable module called *lg.ko.* This module contains the guest operating system's interface to the host kernel. The first element is the switcher, which implements the method by which guest operating systems are context-switched for execution. The /proc file system code (for

`/dev/lguest`) is also implemented in this module, which implements the user-space interfaces to the kernel and drivers, including hypercalls. There's code to provide the memory mapping through the use of shadow page-tables and management of x86 segments.

Finally, the Documentation subdirectory in the kernel contains the launcher utility (`lguest`) to launch a new guest operating system instance. This file does double duty as utility and documentation.

Lguest has been in the mainline kernel since 2.6.23 (October 2007) and was developed and is maintained by Rusty Russell. It consists of approximately 5000 source lines of code, including user-space utilities. Although (reportedly) simple, Lguest provides for true paravirtualization. Along with this simplicity come constraints, however. For example, Lguest virtualizes only other Lguest-enabled guest operating systems and currently only for the x86 architecture. But even with those constraints, Lguest provides an interesting approach to virtualization that is accessible by anyone willing to study Rusty's code.

Linux hypervisor benefits

Developing hypervisors using Linux as the core has real, tangible benefits. Most obviously, basing a hypervisor on Linux benefits from the steady progression of Linux and the large amount of work that goes into it. From the typical optimizations and bug fixes, scheduling, and memory-management innovations to support for different processor architectures, Linux is a platform that continues to advance (to quote John of Salisbury "standing on the shoulders of giants").

KVM proved not long ago that through the addition of a kernel module, one could transform the Linux kernel into a hypervisor. The Lguest hypervisor took this a step further, and with the constraints of paravirtualization, minimized the solution even further.

Another intriguing benefit of using Linux as the platform is that you can take advantage of that platform as an operating system in addition to a hypervisor. Therefore, in addition to running multiple guest operating systems on a Linux hypervisor, you can run your other traditional applications at that level. So instead of worrying about a new platform with new application programming interfaces (APIs), you have your standard Linux platform for application development (in the event a monitoring application or hypervisor management application is needed). The standard protocols (TCP/IP) and other useful applications (Web servers) are available alongside the guests. Recall Figure 4 in the KVM discussion: In addition to the guest operating systems, there's the KVM-modified QEMU. This is a standard process and illustrates the power behind Linux as a hypervisor. KVM makes use of QEMU for platform virtualization, and with Linux as the hypervisor, it immediately supported the idea of guest operating systems executing in concert with other Linux applications.

Conclusion

One thing is clear with the hypervisor developments that are occurring: The hypervisor is a new battleground. Thirty years ago, the operating system was the focus of control and dominated a small number of players. Today, this battleground has shifted to the hypervisor, and Linux has a clear role to play.

But Linux as a hypervisor is not without its critics, and much of the criticism comes from arguments of bloat. These same arguments were used not too many years ago in the embedded domain. Today, Linux as an embedded operating system is a powerhouse and hasn't stopped yet. But that's not to say that there's nothing to this criticism. Perhaps some architectural changes are in order to make a great, ubiquitous operating system even more flexible.

Resources

Learn

- The EPA Report on Server and Data Center Energy Efficiency includes a great survey on where energy is consumed in the data center. From this report, it's clear that virtualization plays a large role in improving the energy efficiency of data centers through server consolidation.
- The Kernel-based Virtualization Machine is one option for a Linux-based hypervisor. You can learn more about KVM at the project's Web site. Here, you'll also find an interesting white paper that describes the ideas behind the technology. KVM continues to evolve, and you can learn about what's happening and what's ahead in the last KVM Forum 2008. You can also find a list of the supported guests at the KVM guest support status page.
- Virtualization comes in many flavors. This article explored two such solutions encompassing full virtualization and paravirtualization. You can learn about some of the other options in "Virtual Linux" (developerWorks, December 2006). You can also dig into more details of KVM and QEMU in "Discover the Linux Kernel Virtual Machine" (developerWorks, April 2007) and "System emulation with QEMU" (developerWorks, September 2007).
- This article touched on some other interesting Linux topics, such as loadable kernel modules and the /proc file system. For more details on these two subjects, check out "Anatomy of Linux loadable kernel modules" (developerWorks, July 2008) and "Access the Linux kernel using the /proc filesystem" (developerWorks, March 2006).
- The Lguest (Simple x68 Hypervisor) shows how to build a simple x86 hypervisor with Linux with a minimum of changes. The Lguest Web site provides the latest details and documentation to learn more.
- Although this article explores the high-level theory behind Linux-based hypervisors, you can explore the installation and use of Lguest at Enterprise Linux Tips.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tutorials and Linux tips on developerWorks.
- Stay current with developerWorks technical events and webcasts.

Get products and technologies

- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the My developerWorks community; with your personal profile and custom home page, you can tailor developerWorks to your interests and interact with other developerWorks users.

About the author

M. Tim Jones
M. Tim Jones is an embedded firmware architect and the author of *Artificial Intelligence: A Systems Approach*, *GNU/Linux Application Programming* (now in its second edition), *AI Application Programming* (in its second edition), and *BSD Sockets Programming from a Multilanguage Perspective.* His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Senior Architect for Emulex Corp. in Longmont, Colorado.