

Ten years of KVM

This article brought to you by LWN subscribers

Subscribers to LWN.net made this article — and everything that surrounds it — possible. If you appreciate our content, please [buy a subscription](#) and make the next set of articles possible.

November 2, 2016

This article was contributed by Amit Shah

We recently [celebrated](#) 25 years of the Linux project. [KVM](#), or Kernel-based Virtual Machine, a part of the Linux kernel, celebrated its [10th anniversary](#) in October. KVM was first announced on 19 October 2006 by its creator, Avi Kivity, in [this post](#) to the Linux kernel mailing list.

That first version of the KVM patch set had support for the [VMX instructions](#) found in Intel CPUs that were just being introduced around the time of the announcement. Support for AMD's [SVM instructions](#) followed [soon after](#). The KVM patch set was [merged](#) in the upstream kernel in December 2006, and was [released](#) as part of the 2.6.20 kernel in February 2007.

Background

Running multiple guest operating systems on the x86 architecture was quite difficult without the new virtualization extensions: there are instructions that can only be executed from the highest privilege level, ring 0, and such access could not be given to each operating system without it also affecting the operation of the other OSes on the system. Additionally, some instructions do not cause a trap when executed at a lower privilege level — despite them requiring a higher privilege level to function correctly — so running a "hypervisor" that ran in ring 0, while running other OSes in lower-privileged rings was also not a solution.

The VMX and SVM instructions introduced a new ring, ring -1, to the x86 architecture. This is the privilege level where the virtual machine monitor (VMM), or the hypervisor, runs. This VMM arbitrates access to the hardware for the various operating systems so that they can continue running normally in the regular x86 environment.

There are several reasons to run multiple operating systems on one hardware system: deployment and management of OSes becomes easier with tools that can provision virtual machines (VMs). It also leads to lower power and cooling costs by hosting multiple OSes and their corresponding applications and services to run on newer, more capable hardware. Moreover, running legacy operating systems and applications on newer hardware without any changes to adapt to the newer hardware now becomes possible by emulating older hardware via the hypervisor.

The functionality of KVM itself is divided in multiple parts: the generic host kernel KVM module, which exposes the architecture-independent functionality of KVM; the architecture-specific kernel module in the host system; the user-space part that emulates the virtual machine hardware that the guest operating system runs on; and optional guest additions that make the guest perform better on virtualized systems.

At the time KVM was introduced, Xen was the de facto open source hypervisor. Since Xen was introduced before the virtualization extensions were available on x86, it had to use a different design.

First, it needed to run a modified guest kernel in order to boot virtual machines. Second, Xen took over the the role of the host kernel, relegating Linux to only manage I/O devices as part of Xen's special "Dom0" virtual machine. This meant that the system couldn't truly be called a Linux system — even the guest operating systems were modified Linux kernels with (at the time) non-upstream code.

Kivity started KVM development while working at Israeli startup Qumranet to fix issues with the Xen-related work the company was doing. The original Qumranet product idea was to replicate machine state across two different VMs to achieve fault tolerance. It was soon apparent to the engineers at Qumranet that Xen was too limiting and a poor model for their needs. The virtualization extensions were about to be introduced in AMD and Intel CPUs, so Kivity started a side-project, KVM, that was based on the new hardware virtualization specifications and would be used as the hypervisor for the fault-tolerance solution.

Development model

Since the beginning, Kivity wrote the code with upstreaming it in mind. One of the goals of the KVM model was as much reuse of existing functionality as possible: using Linux to do most of the work, with KVM just being a driver that handled the new virtualization instructions exposed by hardware. This enabled KVM to gain any new features that Linux developers added to the other parts of the system, such as improvements in the CPU scheduler, memory management, power management, and so on.

This model worked well for the rest of the Linux ecosystem as well. Features that started their life with only virtualization in mind began being useful and widely-adopted in general use cases as well, like [transparent huge pages](#). There weren't two separate communities for the OS and for the VMM; everyone worked as part of one project.

Also, management of the VMs would be easier as each VM could be monitored as a regular process — tools like `top` and `ps` worked out of the box. These days, `perf` can be used to monitor guest activity from the host and identify bottlenecks, if any. Further chipset improvements will also enable guest process `perf` measurement from the host.

The other side of KVM was in user space, where the machine that is presented to the guest OS is built. `kvm-userspace` was a fork of the [QEMU](#) project. QEMU is a machine emulator — it can run unmodified OS images for a variety of architectures that it supports, and emulate those architecture's instructions for the host architecture it runs on. This is of course very slow, but the advantage of the QEMU project was that it had quite a few devices already emulated for the x86 architecture — such as the chipset, network cards, display adapters, and so on.

What `kvm-userspace` did was short-circuit the emulation code to only allow x86-on-x86 and use the [KVM API](#) for actually running the guest OS on the host CPU. When the guest OS performs a privileged operation, the CPU will exit to the VMM code. KVM takes over; if it can service the request itself, it would do so, and give control back to the guest. This was a "lightweight exit". For requests that the KVM code can't serve, like any device emulation, it would defer to QEMU. This implied exiting to user space from the host Linux kernel, and hence this was called a "heavyweight exit".

One of the drawbacks in this model was the maintenance of the fork of QEMU. The early focus of the developers was on stabilizing the kernel module, and getting more and more guests to work without a hitch. That meant much less developer time was spent on the device emulation code, and hence the work to redo the hacks to make them suitable for upstream remained at a lower priority.

Xen too used a fork of QEMU for its device emulation in its HVM mode (the mode where Xen used the new hardware virtualization instructions). In addition, QEMU had its own non-upstream Linux

kernel accelerator module ([KQEMU](#)) for x86-on-x86 that eliminated the emulation layer, making x86 guests run faster on x86 hardware. Integrating all of this required a maintainer who would understand the various needs from all the projects. Anthony Liguori stepped up as a maintainer of the QEMU project, and he had the trust of the Xen and KVM communities. Over time, in small bits, the forks were eliminated, and now KVM as well as Xen use upstream QEMU for their device model emulation.

The "do one thing, do it right" mantra, along with "everything is a file", was exploited to the fullest. The KVM API allows one to create VMs — or, alternatively, sandboxes — on a Linux system. These can then run operating systems inside them, or just about any code that will not interfere with the running system. This also means that there are other user-space implementations that are not as heavyweight or as featureful as QEMU. Tools that can quickly boot into small applications or specialized OSes with a KVM VM started showing up — with [kvmtool](#) being the most popular one.

Developer Interest

Since the original announcement of the KVM project, many hackers were interested in exploring KVM. It helped that hacking on KVM was very convenient: a system reboot wasn't required to install a new VMM. It was as simple as re-compiling the KVM modules, removing the older modules, and loading the newly-compiled ones. This helped immensely during the early stabilization and improvement phases. Debugging was a much faster process, and developers much preferred this way of working, as contrasted with compiling a new VMM, installing it, updating the boot loader, and rebooting the system. Another advantage, perhaps of lower importance on development systems but nonetheless essential for my work-and-development laptop, was that root permissions were not required to run a virtual machine.

Another handy debugging trick that was made possible by the separation of the KVM module and QEMU was that if something didn't work in KVM mode, but worked in emulated mode, the fault was very likely in the KVM module. If some guest didn't work in either of the modes, the fault was in the device model or QEMU.

The early KVM release model helped with a painless development experience as well: even though the KVM project was part of the upstream Linux kernel, Kivity maintained the KVM code on a separate release train. A new KVM release was made regularly that included the source of the KVM modules, a small compatibility layer to compile the KVM modules on any of the supported Linux kernels, and the kvm-userspace piece. This ensured that a distribution kernel, which had an older version of the KVM modules, could be used unchanged by compiling the modules from the newest KVM release for that kernel.

The compatibility layer required some effort to maintain. It needed to ensure that the new KVM code that used newer kernel APIs that were not present on older kernels continued to work, by emulating the new API. This was a one-time cost to add such API compatibility functions, but the barrier to entry for new contributors was significantly reduced. Hackers could download the latest KVM release, compile the modules against whichever kernel they were running, and see virtual machines boot. If that did not work, developers could post bug-fix patches.

Widespread adoption

Chip vendors started taking interest and porting KVM to their architectures: Intel added support for IA64 along with features and stability fixes to x86; IBM added support for s390 and POWER architectures; ARM and Linaro contributed to the [ARM port](#); and Imagination Technologies added MIPS support. These didn't happen all at once, though. ARM support, for example, came rather late

("it's the reality that's not timely, not the prediction", quipped Kivity during a KVM Forum keynote when he had predicted the previous year that an ARM port would materialize).

Developer interest could also be seen at the KVM Forums, which is an annual gathering of people interested in KVM virtualization. The first KVM Forum in 2007 had a handful of developers in a room where many discussions about the current state of affairs, and where to go in the future, took place. One small group, headed by Rusty Russell, took over the whiteboard and started discussions on what a paravirtualized interface for KVM would look like. This is where [VIRTIO](#) started to take shape. These days, the KVM Forum is a whole conference with parallel tracks, tens of speakers, and hundreds of attendees.

As time passed, it was evident the KVM kernel modules were not where most of the action was — the instruction emulation, when required, was more or less complete, and most distributions were shipping recent Linux kernels. The focus had then switched to the user space: adding more device emulation, making existing devices perform better, and so on. The KVM releases then focused more on the user-space part, and the maintenance of the compatibility layer was eased. At this time, even though the kvm-userspace fork existed, effort was made to ensure new features went into the QEMU project rather than the kvm-userspace project. Kivity too started feeding in small changes from the kvm-userspace repository to the QEMU project.

While all this was happening, Qumranet had changed direction, and was now pursuing desktop virtualization with KVM as the hypervisor. In September 2008, Red Hat announced it would acquire Qumranet. Red Hat had supported the Xen hypervisor as its official VMM since the Red Hat Enterprise Linux 5.0 release. With the RHEL 5.4 release, Red Hat started supporting both Xen and KVM as hypervisors. With the release of RHEL 6.0, Red Hat switched to only supporting KVM. KVM continued enjoying out-of-the box support in other distributions as well.

Present and future

Today, there are several projects that use KVM as the default hypervisor: OpenStack and oVirt are the more popular ones. These projects concern themselves with large-scale deployments of KVM hosts and several VMs in one deployment. These come with various use cases, and hence ask of different things from KVM. As guest OSes grow larger (more RAM and virtual CPUs), they become more difficult to live-migrate without incurring too much downtime; Telco deployments need low latency network packet processing, so [realtime KVM](#) is an area of interest; and faster disk and network I/O is always an area of research. Keeping everything secure and reducing the hypervisor footprint are also being worked on. The ways in which a malicious guest can break out of its VM sandbox and how to mitigate such attacks is also a prime area of focus.

A lot of advancement happens with new hardware updates and devices. However, a lot of effort is also spent in optimizing the current code base, writing new algorithms, and coming up with new ways to improve performance and scalability with the existing infrastructure.

For the next ten years, the main topics of discussion may well not be about the development of the hypervisor. More interesting will be to see how Linux gets used as a hypervisor, bringing better sandboxing for running untrusted code, especially on mobile phones, and running the cloud infrastructure, by being pervasive as well as invisible at the same time.

Index entries for this article

[GuestArticles](#)

[Shah, Amit](#)

to post comments