Introduction to paravirtualization

G. Lettieri

2 Nov. 2017

Paravirtualization has been introduced in the first releases of the Xen hypervisor. Xen was built to run x86 virtual machines on x86 systems, before the introduction of the Intel and AMD hardware extensions for virtualization. As we have seen, normal x86 systems are not easy to virtualize by a standard trap-and-emulate hypervisor, since some crucial instructions are not trapped when executed at lower privilege. We note that this problem arises because of one of the requirements of virtualization: we don't want to change the target software. If we could change the target software, we could replace the difficult-to-virtualize instructions with something else. This is the main idea of paravirtualization: make the target software "aware" of the fact that it is running inside a virtual machine.

Of course, we should find a good compromise: rewriting all the target software from scratch is out of the question. We note that kernels are typically ported to several, completely different machines, like x86 and ARM. This is achieved partly by the use of high level languages (typically C) and partly by relegating the system-specific parts that must be written in assembly language to a few well-defined routines (e.g., routines to access the MMU or save/restore the CPU registers). Porting a well designed kernel to a new architecture is a matter of replacing the system-specific routines and recompiling the rest.

In Xen they observed that a virtual machine can be seen as just another architecture to which a kernel may be ported. In this architecture, for example, you access the MMU by actually issuing calls to the hypervisor, instead of writing into registers. This idea was put forward to simplify the task of building virtual machines in the x86 architecture, but it also has potential performance advantages: in the port, the kernel may be optimized for the virtual architecture. It also has the obvious disadvantage that you cannot install whatever you want inside a Xen virtual machine: you have to be able to modify the target kernel. Essentially, this was only fully completed for Linux and FreeBSD.

Note that Xen no longer uses paravirtualization for the purpose of virtualizing the CPU, as in the original implementation. Instead, it relies on the newly available hardware extensions to implement hardware-assisted virtualization. Therefore, you can install any OS inside a modern Xen VM. Still, paravirtualization has become an important topic in other areas, e.g. in I/O: instead of trying to (inefficiently) emulate some hardware I/O device, we can define a new virtual-only device, and simply provide a driver for it. Note that the idea is

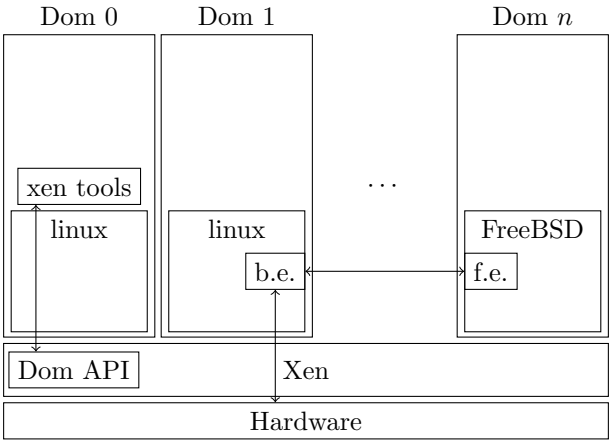

Figure 1: Xen architecture

still the same: we reuse some existing infrastructure in the original kernel (in this case, the ability to install new drivers) to make some limited change (a new driver) designed to work well in a virtualized environment. This very important topic is examined in the virtio seminaries.

We now briefly describe the general architecture of Xen, which is an important hypervisor on its own (see Figure 1). Xen is a very small kernel that is loaded first on the machine and gains direct control to the hardware On top of Xen, at lower privilege level, we have so-called *domains*. There may be as many domains as needed, and inside each domain we may run an entire OS with its own applications. Domains are isolated from each other, and are used to implement the virtual machines. One special domain, Dom 0, has access to the Xen API to create and destroy other domains. Inside Dom 0 you can install any OS you like, so that you can develop your own tools to manage the other domains. Typically, Dom 0 contains a Debian Linux with Xen management tools installed. Domains can be given direct access to some I/O devices, or they can use fully virtualized devices (taken from QEMU), or they can use paravirtual devices. Paravirtual devices are split in a front-end and a back-end, each running into a different domain and exchanging I/O requests across the domain boundaries. The typical setup is to run the back-end in a domain that has direct access to hardware devices and gives indirect access to possibly several front-ends running in non-privileged domains. In Figure 1 we have a linux driver running in Dom 1 working as a back end for a FreeBSD front-end driver in Dom n.

The kernels running in each domain may be either standard, unmodified kernels (using hardware-assisted virtualization), or they may directly use some of the Xen APIs to improve performance. Let us consider, for example, virtual memory, to see how a paravirtual kernel (i.e., a kernel that is aware that it is running inside a virtual machine) may run faster than an unmodified one,

assuming that nested page-tables are not available (as was the case in the original Xen). We have seen that, in this case, the guest kernel uses a set of page tables that are not the ones actually used by the MMU. The reason for keeping them distinct is that the translation created by the guest kernel (G) must be composed with the translation from guest-physical to host-physical addresses created by the Hypervisor (H) before we can let the MMU use it. There are two distinct reasons for having H:

1. to create the illusion of contiguous memory in the guest kernel;
2. to limit access from the guest kernel to the pages that have been assigned to it.

Note that to enforce point two we only need to deny write access to the host page tables, but we could still grant read access to them. This would benefit performance, since now the guest kernel would be able to read the MMU-updated A and D bits from the page tables, without the need for the hypervisor to synchronize them from the host tables. However, in a standard "fully" virtualized machine (i.e., non paravirtual), we cannot even grant read access because of point one. This is a consequence of the guest kernel ignoring the distinction between guest and host physical addresses and thinking that it has access to a range of contiguous physical addresses starting at 0, while the hypervisor may have assigned to it a set of pages scattered anywhere in host memory. Therefore, the (unmodified) guest kernel would not be able to correctly interpret the host page tables. The situation changes for a paravirtual kernel: this kernel may be fully aware of the distinction between guest and host physical addresses, and so it can make good use of a read access to the host page tables. To enforce point two, write access is still denied, and the paravirtual kernel must call an hypervisor protected routine (an hypercall) whenever it wants to update its page tables. The hypervisor will then check that the updates do not grant the guest kernel access to portions of memory not assigned to it.