Advanced virtualization topics

G. Lettieri

14 Dec. 2016

1 Introduction

In this lecture we briefly review some more advanced topics related to virtualization. The common observation to all these topics is that the most important use-case for virtualization is the safe sharing of hardware resources among different applications. Typically, each virtual machine only runs just one application. Since our real interest is in the application, not in the OS, do we really need to insist that the OS must be *unmodified*, i.e., it must be the same OS that runs on real hardware? Most of the costs of virtualization come from the need to intercept and emulate actions performed by the guest OS kernel that assumes it has direct access to the hardware. But we should actually be free to change the internals of the guest OS as long as the OS API does not change, so that the application is unchanged. This is the idea of *Paravirtualization*.

This idea can be pushed even further. Do we really need to run a full Operating System inside each virtual machine, just for the purpose of running a single application on top of it? Most of the functions of this guest OS will be essentially duplicated in the host OS, thus potentially wasting a lot of resources. The answer to this observation are *Unikernels*, the second topic that we are going to survey. These are very small and simple kernels, designed to run just one application, possibly only in a virtualized environment.

Finally, we can observe that sharing the resources among applications is what Operating Systems should be designed for, so do we really need Virtual Machines at all for this use case? The answer is not so simple, since unfortunately existing OSs have increasingly become less good at isolating user applications from each other. However, *Containers*, our last topic, try to improve the situation by adding new isolation features to the traditional host OS. They are put forward as an alternative to Virtual Machines.

1.1 Paravirtualization

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 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 have to be able to modify the 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. Still, paravirtualization has become an important topic in other areas, namely 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 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.

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 tool 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 (usually Dom 0) and gives indirect access to possibly several front-ends running in non-privileged domains.

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 to use it. There are two distinct reasons for having H:

1. it is used to create the illusion of contiguous memory in the guest kernel;
2. it is used 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, 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.

1.2 Unikernels

Unikernels are kernels that are directly linked to an application, like a normal library. The resulting executable can then be loaded and run directly on the hardware machine. They usually only provide a single process and are not harware-protected from the "user" code: unikernel primitive calls are implemented like standard function calls and do not involve a change in privilege level. During the linking process, only the functions that are actually used by the application are included in the final executable. In practice, they differ from a standard library only for the kind of code that they contain: low level access to the hardware.

Unikernels are an old idea, but they have become much more important today because they make a very good match with virtual machines:

- virtual machines can provide a stable and well defined set of I/O interfaces, greatly simplifiying the task of maintaining a unikernel, which does not have to provide drivers for the daunting variety of hardware devices normally available;
- hypervisors already provide all the isolation that is needed to run several applications, one for virtual machine; inside the virtual machine there is no need to replicate this functionality, so a simpler kernel is sufficient.

Having a stripped down unikernel has many adantages with respect to a full blown operating system. First, the memory needed by the virtual machine is much less, and this implies that we can have more of them and that migrating them is faster. Second, no unneded service is included, and this means faster boostrap and generally improved performance.

From the point of view of protection we may reason that, since the unikernel is running just one application, there is no need to protect it. Some systems still provide this kind of protection, mostly to isolate bugs in the applications, but they check for protection violations statically, during the compilation and linking phase. This is only practically possibile, however, if the applications are written in languages amenable to this kind of static analysis. This is not the case for C, for example, where unchecked array bounds, unions and casts to pointers allow a program to access any memory location at run time, in a way that is essentially impossible to prevent by analyzing the code. Java, or any language interpreted by the Java Virtual Machine, is a more viable option and is used, e.g., by the OSv unikernel.

1.3 Containers

Containers (or jails) are a feature of some host operating systems, but we limit our considerations on Linux only, since this is what is commonly used in this area. Containers are a way to isolate a set of processes and make them think that they are the only ones running on the machine. The machine they see may feature only a subset of the resources actually available on the entire machine (e.g., less memory, less disk space, less CPUs, less network bandwitdh). Many different containers may coexist on the same machine.

Containers are *not* virtual machines, even if they may look like ones in some cases. Processes running inside a container are normal processes running on the host kernel. There is no guest kernel running inside the container, and this is the most important limitation of containers with respect to virtual machines: you cannot run an arbitrary operating system in a container, since the kernel is shared with the host (Linux, in our case). The most important advantage of containers with respect to virtual machines is performance: there is no performance penalty in running an application inside a container compared to running it on the host. There is also a point of contention between containers and virtual machines, about which one is more secure. VMs are considered (by some) to be more secure of containers, because they have a smaller *attack surface*. By attack surface we mean the amount of code and features that a malicius attacker may

probe for exploitable bugs: the entire host kernel, in the case of containers, and the hypervisor in the case of virtual machines. The KVM and Xen hypervisors, for example, are very small. In the KVM case, it must be noted that KVM is a small module, but it actually uses facilities from the rest of the linux kernel (for scheduling, virtual memory, etc.). However, this is still less than the amount of code involved in the implementation of containers.

Container in Linux are implemement using two distinct kernel features: namespaces and control groups. We briefly examine each one in turn.

1.3.1 Namespaces

Namespaces provide a means to segregate system resources so that they can be hidden from selected processes. An old feature of Unix systems, which has a similar purpose, is the `chroot()` system call, which works as follows:

- The kernel remeber for each process, the inode of the the process *root directoy*;
- This directory is used as a starting point whenever the process passes the kernel (e.g., in a `open()`) a filesystem path that starts with "`/`";
- whenever the kernel walks through the components of any filesystem path used by the process and reaches the process root directory, a subsequent "`..`" path element is ignored;
- only root can call `chroot()`;
- the process root directory is inherited by its children.

Normally, all processes have the same root directory, which coincides with the root directory of the file system. But, by using `chroot()`, we can make a subset of the filesystem look like it was the full filesystem for a set of processes. This is typically used to segregate untrusted processes that provide network services and, because of possible bugs in their implementations, may be forces by remote attackers to execute arbitrary code. The idea is to prepare a subtree in the filesystem that contains only the things that are needed for the execution of the server, and nothing else—a chroot environment. Then, the server process is started after a chroot to the root directory of the chroot environment. Even if the server is subverted, it cannot access any file outside of the chroot environment. [1]

Chroot enviroments, however, are not full containers. Contrary to popular belief, infact, not everything is a file in Unix. For example network interfaces, network ports, users and processes are not files. While we can have as many instances as we want of, say, `/etc/passwd`, each different and living in its own chroot environment, we can only have one port 80 throughout the system (thus, only one web server), only one process with pid 1 (thus, only one init process),

[1] Note that in the earlier implementations of the mechanism, root was able to escape a chroot, so this strategy was only effective is the server did not run as root.

and user and process ids will have the same meaning in all chroot environments. Thus, for example, a process running in a chroot environment will still be able to see all the processes running in the system, and it will be able to send signals to all the processes belonging to any user with the same user id as its own.

Namespaces have been introduced to create something similar to chroot environments for all these other identifiers. Each process in Linux has its own network namespace, pid namespace, user namespace and a few others. Network interfaces and ports are only defined inside a namespace, and the same port number may be reused in two different namespaces without any ambiguity. The same holds true for processes and users. Normally, all processes share the same namespaces, but a process can start a new namespace that will be then inherited by all its children, grandchildren, and so on. This is done when the process is created using the `clone()` system call. This system call (taken from the Plan 9 OS) is the new, preferred way to create new processes in Linux, since it generalizes the behaviour of `fork()` and can be also used to implement `pthread_create()`. The idea is that both processes and threads share something with their creating process, and have a private copy of something else. For example, processes share open files with their parent, but have a private copy of all the process memory. Threads, instead, also share the process memory. The `clone()` system call is passed a set of flags using which the programmer may choose what to share and what to copy. This same system call has been extended to implement namespaces, essentially by adding flags for the sharing or copying of the network, pid, user namespaces and so on.

1.3.2 Control groups

While namespaces can be used to hide and create private copies of all the system entities, they are not sufficient in isolating sets of processes so that thay cannot interfere with each other. Processes may interfere also by abusing the system resources, e.g., allocating to much memory, using to much CPU time, or disk and network bandwidth. To properly implement containarers, therefore, we also need to limit the usage of resources by the processes that live in the container. This is another thing that was not done very well before the introduction of *control groups*. The problem is that, before enforcing a limit on a set of processes, we need to know which processes belong to the set, and the processes must not be able to escape from the set. We would also like some flexibility in the definition of the set. Traditional Unix has a concept of process groups, but unfortunately any process is free to enter or leave a group. Processes are also grouped by user id (the user that is running them), but this is not very flexible. Control groups, instead, are groups explicitly created by the administrator, who can later assign processes to them. Processes cannot escape a control group, only possibily create other control groups that will always be subgroups of the original group, and therefore the association with the group is never lost. Control groups can be linked to so-called *subsystem*. Subsystems are used to control the resources assigned to the linked group. There is, for example, a memory subsystem, used to enforce limits on the usage of the system memory, and a

CPU subsystem, used to force the processes in the linked control group to run only on some of the available CPU cores.