Lecture 11 - Virtualisation

Gidon Rosalki

2025-06-15

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Introduction

The idea is to decouple the software form the hardware. It is one of the most important advances in computer systems in the last 25 years. For organisations, it allows server consolidation, and reduces hardware costs (in cooling, electricity, maintenance, etc.). For users, it allows us to run more than one OS on one machine, and use the best one for each task. This is the basis for cloud computing.

In lecture 1, we discussed virtualisation, and state that the OS virtualises the hardware decoupling from physical limitations, and supporting the illusion that there are more resources available, where each app thinks that it has the machine for itself. This is implemented by introducing a level of indirection, by juggling resources among applications, but needs hardware support.

1.1 Technical definition

Virtualisation is

- Decoupling from physical limitations - what you get does not exist in reality
- Using a level of indirection - how we hide the limited reality
- Providing the same interfaces - distinction from abstraction

1.2 Virtual machines

1.2.1 Processes as VMs

A process is a virtual machine **as applications see it**. The CPU, but only non privileged instructions. Virtual memory, as an abstraction of physical RAM, and also new abstractions supported by the OS, eg a file system.

1.2.2 Real virtual machines

Here we virtualise the complete hardware **as the operating system sees it**. This means cirtualising the CPU, with all its instructions, including privileged ones, the physical memory, along with address mapping, and all the peripheral devices. This type of virtual machine can run an OS.

Real virtual machines have the virtualisation layer. This is the **hypervisor**, or **virtual machine monitor**. It adds extra levels of indirection, decoupling the hardware, and the OS. It also has multiplexing for the actual hardware, with strong isolation between those using it, and it manages physical resources by performing functions similar to that of an OS.

Let us consider some examples of virtualisation with multiplexing. The virtual memory, supports multiple address spaces on a single, smaller physical memory, each accessed with byte addressing. There are also disk partitions, which make a single disk look like multiple smaller disks, each providing independent block storage. Virtual machines are for server consolidation, and will be discussed further later.

We can also virtualise with aggregation. Here we create a virtual device using multiple physical devices. For example RAID, which is an array of disks, with replication. Data is replicated internally to improve reliability, while the interface stays conventional block storage, any file system stored in traditional storage can be stored in RAID instead.

1.2.3 Some history

In the 1960s/70s there was virtualisation on mainframes, since they had single user OSs, and VMs allowed multiple users. In the 80s/90s there was a shift to personal machines, with the development of unvirtualisable hardware, eg read only unprivileged access to privileged data, and instructions that silently behave differently in privileged and unprivileged modes. Finally, in the 2000s, there was a ”rediscovery of the benefits of virtualisation”, which has extensive use in cloud infrastructure, with new hardware support.

2 Virtualisation benefits

Why should we bother? Well, it allows OS diversity, where we may run many different OSs on the same machine, and server consolidation supports flexibility and efficiency in server farms. The security is also an important factor, the isolation of VMs, and monitoring system behaviour, for example for intrusion detection. This security is far greater than can be granted on a standard machine. Additionally, the availability is big, VMs can run anywhere, and benefit from live migration. The cloud benefits are also big, where the cloud is infrastructure as a service. We rent VMs from cloud providers, strongly isolated from other VMs running on the same physical server, thus avoiding the expense of in house infrastructure, and maintenance personnel.

2.1 Consolidation

It also grants us server consolidation, servers often run a single major application or service. This provides strong isolation between services and instances, sometimes including performance isolation, but is an inefficient and inflexible use of hardware and energy, where some servers will be overloaded, and others idle. Virtualisation resolves this by allowing us to consolidate many servers onto fewer machines when the workload is low, reducing costs, and as demand for a particular service increases, we can spread it over more hardware in real time.

2.2 Legacy

There are also other uses, such as support for legacy applications. We can emulate an old OS in a VM in order to run them. It is also helpful in low level software development, if software crashes it might crash the VM, but not the whole system. Additionally, the security from isolation allows things like malware analysis, in a secured environment.

2.3 Encapsulation

Here we have the entire VM in a file, including the OS, application, and data, along with the memory and device state. This enables snapshots and clones, where we can capture the VM state on the fly, and preserve it / copy it elsewhere as and when we want, restoring it to the exact state in which it was. This allows rapid system creation (we can just copy paste new VMs if we want many of the same format), makes backups easy, and allows easy migration of VMs between machines for load balancing. Consider when we have a web server run by a virtual machine, which is being overloaded, we can copy this VM, and create a new instance very easily, and then share the load between our old VM, and the new one.

3 Types of hypervisors

There are a few types of hypervisor. Instead of a type that is given direct access to the hardware, we can run a hypervisor on top of another OS. This is useful in a home environment, where perhaps we want to use both our host machine, and the VM simultaneously. We also have containers, where there is a virtualisation layer (not hypervisor) which runs containers on top of the OS, sharing an OS rather than emulating its own. The hypervisor on bare metal is called a Type I hypervisor, hypervisors on top of the OS is Type II, and containers are there own beast, and simply described as application encapsulation.
This will be discussed further in exercise 5.

4 Virtualisation basics

The hypervisor is like an OS kernel, and VMs are like processes. The hypervisor controls everything, from scheduling VMs, allocating memory, and multiplexing IO devices. We then need to ask, how do the VM OSs think that they run on bare metal and control everything? Well, the hypervisor needs to fake it.

4.1 Virtualisation mechanics (how to fake it)

In order to virtualise these mechanics / fake it, we need to create the following systems:

- Trap and emulate
- Binary translation
- Paravirtualisation
- Hardware assistance

We will focus on Type 1, though most is applicable to other hypervisors as well.

4.1.1 Trap and emulate

Only the hypervisor runs on bare metal in privileged mode (ring 0). The VMS all run directly on HW in **user mode**. Both user apps, **and the OS of the VM**. So we then trap and emulate sensitive instructions, where applications making system calls trap into the hypervisor, and the hypervisor calls the OS handler (in user mode). If the OS uses privileged instructions, it traps to the hypervisor (General Protection Fault), and the hypervisor executes the instructions on its behalf. This can work if the number of traps is not too large (and if the architecture supports trap and emulate).

This operates under the assumption that all sensitive instruction are privileged, and so cause a trap if attempted in user mode. This was not the case on Intel x86 architecture in the 1980s, but is the case on modern architectures.

4.1.2 Sensitive but Unprivileged Instructions (Red Pills)

WE have unprivileged instructions that read data that can only be set by privileged instructions (such as PUSHFD, which writes the interrupt enable flag to the stack), and behave differently in privileged / unprivileged mode, for example POPFD which reads the interrupt enable flag into the EFLAGS register (if in ring 0), and does nothing if it is in ring 3. To resolve this, we use dynamic binary translation, which is needed if not all sensitive instructions are privileged:

1. Translate all VM code blocks into safe code, where for normal code it uses identity translation, and for sensitive code it replaces it with ”hypercalls” (which are calls to the hypervisor, like system calls)
2. These are then put into the code cache, and from now on, whenever this code is encountered, the safe version will be executed instead

4.1.3 Hardware assistance for type 1

Comply with the Popek/Goldberg theorem, where all sensitive operations must be privileged, and cause a trap which can be caught. Then we support virtualisation in the processor hardware, so there is a new privileged mode (unofficially known as ring -1), and now the VM (guest kernel) can run in native ring 0. Therefore, this can now trap to the hypervisor for every sensitive operation.

4.1.4 Virtual memory problems

Even with hardware virtualisation support, virtualising the MMU is a challenge. The guest OS thinks that it owns the physical memory, and page table, and so will modify the page to frame mapping freely, try and context switch (load new page tables), and so on. If we do not virtualise the MMU, then the guest OS will cause many problems in the read RAM.

4.1.5 Shadow tables

We can virtualise the MMU by pointing it to a real page table (shadow table) while making the guest OS think it’s still managing it via the guest page table. The shadow table will map the guest OS pages to real memory frames, managed by the hypervisor, and the mapping is based on a virtual frame to real frame mapping, maintained by the hypervisor. This has the same page table structure (pages) as the guest page table, but pages containing the guest page table are read only.

These come with the pros of hardware speed memory access, as long as there are no page faults, however, it will be incredibly when when there are page faults, since we must trap every tie the guest page table is updated (since an access violation = page fault), which is to say we must trap all page faults. Thus, every page fault is now a VM exit event (which is really very slow)

4.1.6 Nested paging

So we see that there are also problems with shadow tables. Let us instead introduce "second level address translation" or **SLAT**, also known as nested paging. This is also known on Intel as the Extended Page Table (EPT), and in AMD as Rapid Virtualisation Indexing (RVI). The concept is to offload the virtual frame, to real frame mapping from the hypervisor (which is in software), to the CPU (i.e. in hardware).

So we have no shadow page table, the page table is managed by the guest OS, and page table translation ends in a virtual frame. Now we add another (2nd) page table like translation from a virtual frame, to a physical frame (visible only to the hypervisor), and the second page table is managed by the hypervisor.

The original (and hierarchical) page table is based on frame numbers, pointing down the table hierarchy. So, this method requires a second level translation for all of them, which is of course slower. Here is a graphical demonstration from `this` blog, which demonstrates that each attempt to read from the memory in the guest OS, results in a full address read, with the entire page table hierarchy in the host OS.

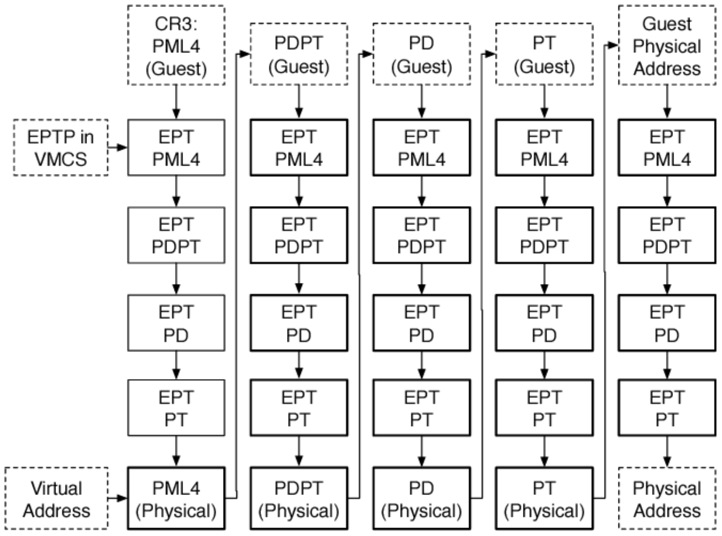

Figure 1: EPT full translation

However, it is not that bad in reality. We do need a second level translation for each virtual to real translation, but this is at the speed of the hardware, and we have the TLB to help us in this.

5 Containers

Containers are a middle ground between processes and VMs. For example, where processes share file system, containers do not. Containers mostly have applications, dependencies, libraries, binaries, and configurations files.
Here we virtualise the OS, and no the hardware. Each container has its own file-system, memory, and OS resources, as if it runs by itself, however, the hardware is shared. Containers are a middle ground between processes and VMs. For example, processes share file systems, and containers do not.

However, we have said that **processes** are virtual machines, as the applications see it, exactly like containers. We may take another view on containers, which is that they are packaging a set of processes, having their own OS resources, which are defined by the concept of **namespaces**, and controlled by the concept of **cgroups**.

5.1 Namespaces

A **namespace** wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of a global resource. We define 7 namespaces:

- PID
- User
- Mount
- Network
- IPC
- UTS

- cgroups

Each process has its own address space, and restricted privileges. Containers can be defined as each having its own namespaces. So when Container A has its own PID namespace, implying it sees only processes in the container, and container B is in a separate namespace, they both have processes within with an ID of 1, which refer to different processes.

	Process	Container	VM
Definition	A representation of a running program	Isolated group of processes managed by a shared kernel	A full OS that shares host hardware via a hypervisor
Use case	Abstraction to store state about a running process	Creates isolated environments to run many apps	Creates isolated environments to run many apps
Type of OS	Same OS and distribution as the host	Same kernel, but different distribution	Multiple different operating systems
OS isolation	Memory space and user privileges	Namespaces and cgroups	Full OS isolation
Size	Whatever user's application uses	Images measured in MB + user's application	Images measured in GB + user's applications
Lifecycle	Created by forking, can be long or short lived, more often short	Runs directly on the kernel, with no boot process, is often short lived	Has a boot process, and is typically long lived

Table 1: Concepts comparison

	Virtual machines	Containers
Require hardware support	Yes (production grade)	No
Can run in a guest VM (e.g. in the cloud)	No, we cannot run a VM in a VM	Yes
Isolation	Hardware based	Software (kernel) based
Kernel instances	Multiple different	Single
Root access	Full	Limited
Runtime overhead	Workload dependent	Negligible, but increases with use of extra features, such as NAT

Table 2: Technical comparison

5.2 Real world uses

Many applications that are run in the cloud today are run inside containers. We have migrated from a system of monolithic services, to microservices, which may be implemented as containers.

6 Virtualisation

6.1 IO virtualisation

We have a few options here that can be carried out. We may firstly trap and emulate the devices in the hypervisor. Secondly, we can split in the hypervisor, such that the guest OS has a virtual driver, which interfaces with the device driver in the hypervisor. This is Paravirtualisation, and is a very simple interface, but we need to port all drivers to this. On top of that, the drivers need to be installed on the hypervisor itself, requiring special drivers for each hypervisor. Finally we have pass through, where the device driver is in the guest OS, and it is passed straight through to the hardware device.

7 Type 2 hypervisors

Until now we have spoken just of type 1, which duplicates much OS functionality. Type 2 avoids duplication, and uses OS services, where the OS schedules the VMs, and allocates memory. Files are used to emulate disks, and VMs can be killed with signals. This requires a hypervisor related kernel module.

8 Paravirtualisation

Here we modify the guest OS, so that all calls to privileged instructions are changed to hypervisor calls (hypercalls), which reduces the number of traps, and removes sensitive instructions, but requires recompiling the guest OS specifically for each host OS. It is much easier, and more efficient, to modify source code, than to emulate hardware instructions (as in binary translation), but also means that one cannot run any guest OS as is, but rather need to access the source code.

9 Products

There are a few different systems. Microsoft have have Hyper-V, which is type 1 ish. QEMU, which is a system emulation mode (complete binary translation) and is type 1. Oracle own VirtualBox, which is a type 2 hypervisor, and VMware provide the ESXi server, which is type 1, with optional paravirtualisation. VMware also provide Workstation Pro, which is type 2. Finally (in this list) there is Xen, which is type 1, with optional paravirtualisation.

10 Summary

In conclusion, virtualisation provides a way to consolidate OS installations onto fewer hardware platforms, and there are 4 basic approaches, type 1 hypervisors, type 2 hypervisors, paravirtualisation, and containers. These must also account for virtual access to shared resources, such as memory, and IO.