1 Virtualisation

Consider a company that has an e-mail server, a web server, an FTP server, etc. All these servers usually run on **separate machines** because of the load and/or primarily separate machines. Sysadmins simply do not trust the OS to run forever with **no failure**. In separate machines, if one of these servers fail, at least the other ones are **not affected**. Is this a good policy?

1.1 Virtual machines

- **Abstract hardware** of a single computer into several **different execution environments**. An old idea, actually, from 60s and 70s.
- Similar to layered approach, but layer creates **virtual system** (virtual machine, or VM) on which operation systems or applications can run several components.

1.2 Motivation

A failure in a particular VM does not result in **bringing down** any others. Of course, a failure at the server level would result in an even more **catastrophic** situation. Nevertheless, most service outages do not come from faulty hardware, but due to **unreliable software** (especially OSs). We have already seen that an OS virtualises :

- Virtual memory
- Virtual file systems

However, a VM virtualises an **entire physical machine**: Providing the illusion that software has full control over the hardware. As implication, you can run **multiple instances** of an OS (or different OS) on the **same physical machine**.

1.3 Components

- Host - underlying **hardware system**
- **Virtual machine manager** (VMM) or hypervisor - **creates and runs** virtual machines by providing an *interface* that is identical to the host (Except in the case of paravirtualization)
- Guest - process provided with **virtual copy** of the host (Usually an operating system)

1.4 Main properties

- Isolation: each VM is **independent**, so failures do not affect the host.
- Encapsulation: state can be **captured** into a file. Check pointing, migration. It is easier than migrating processes. We merely have to move the memory image that contains OS tables.
- Interposition: All guest actions go through the monitor (**VMM**), which can inspect, modify, deny operations.
- Fewer physical machines saves money on hardware and electricity.
- Run legacy applications.

1.5 Requirements

- Safety: the hypervisor should have **full control** of the virtualised resources (Resources sharing).
- Fidelity: the behaviour of a program on a VM should be **identical** to that of the same program running on bare hardware. What if we run privileged instructions? (Virtualisation Technology (VT) - Hardware support).
- Efficiency: much of the code in the VM should run **without intervention** by the hypervisor (Overheads). For instance, with VMware:
  - CPU-intensive apps: `2-10%` overhead
  - I/O intensive apps: `25-60%` overhead

1.6 Approaches

- Full virtualisation: It tries to trick the guest into believing that it **has the entire system**.
- Paravirtualisation: VM does not simulate hardware. It offers a set of *hypercalls* which allows the guest to send explicit requests to the hypervisor (as a system call offers kernel services to applications).
- Process-level virtualisation: The aim is to simply allow a process that was **written for a different OS** to run. For instance, Wine in Linux to run Windows applications, Cygwin to run Linux shell on Windows.

2 Virtual Machines

2.1 Types of hypervisors

- **Natives** (Type 1): Technically, it is like an OS, since it is the only program running in the most privileged mode. Its job is to **support multiple copies** of the actual hardware.
- **Hosted** (Type 2): It relies on a OS to **allocate** and **schedule** resources, very much like a **regular process**.

2.2 Naive virtual machines

Hypervisor installs **directly on hardware**. The hypervisor is the **real kernel**. (Unmodified) **OS runs in user mode**.

- It seems to be in kernel model: virtual kernel mode.
- Privileged instructions need to be processed by the **Hypervisor**.
- Hardware VT technology will be **necessary**.

Acknowledged as **preferred architecture** for high-end servers. Paravirtualisation-based VMs are typically based on type 1 hypervisors.
Examples: VMware ESX Server, Xen, Microsoft Viridian (2008)

2.3 Hosted virtual machines

- Installs and runs VMs as an **application** on an existing OS.
- Relies on **host scheduling**. Therefore, it may not be suitable for **intensive** VM workloads.
- I/O path is **slow** because it requires world switch.
- Process-level virtualisation will rely on **type 2** hypervisors.
- It needs an OS.

Examples: VMware Player/Workstation/Server, Microsoft Virtual PC/Server, Parallels Desktop

3 Requirements for virtualisation

A hypervisor must **virtualise**:

- Privileged instructions (Exceptions and interruptions)
- CPU
- Memory
- I/O devices

Approaches will be similar to what we did with OSs. A bit **simpler** in functionality and implementing a different abstraction: Hardware interface vs. OS interface.

3.1 Virtualise priveledged instructions

It is not safe to let **guest kernel** run in **kernel mode**. So a VM needs two modes: *virtual user mode* and *virtual kernel mode*. Both of which run in **real user** mode! What happens when the guest OS executes an instruction that is allowed only when the **CPU really is in kernel mode**? (e.g. map virtual pages to physical pages)

- Type-1 hypervisors: In CPUs without Virtual Technology (VT), the instruction fails, and the OS crashes.
- Type-2 hypervisors could work without VT - privileged instructions are emulated (**binary translation**)

How does switch from virtual **user mode** to virtual **kernel mode** occur?

- Attempting a privileged instruction in user mode causes an **error** it is trapped
- VMM gains control, analyses error, executes operation as **attempted by guest**
- Returns control to guest in user mode
- Known as *trap-and-emulate*

3.2 CPU

- VMMs need to *multiplex* VMs on CPU. But, how can we do that?
- **Time-slice the VMs**
- Each VM will **timeslice** its OS/Applications during its quantum.
- For type-1: we will need a relatively simple scheduler (E.g. round robin, work-conserving (give unused quantum to other VMs))

3.3 Memory

The hypervisor **partitions** memory among VMs. It assigns **hardware pages** to VMs. It needs to **control** mappings for isolation. In each VM, the OS creates and manages page tables. But these tables **are not used** by the MMU. For each VM, the hypervisor creates a **shadow page table** that maps virtual pages used by the VM onto the actual pages the hypervisor gave it.

3.4 Events and IO

Guest OSs **cannot** interact directly with I/O devices - but the guest OS **thinks** it owns the device (e.g. disk). VMM **receives interrupts** and exceptions.

- Type I hypervisors **run the drivers**.
- Type II hypervisors: Driver knows about VMM and **cooperates** to pass the buck to a **real device driver** on the underlying host OS