Chapter 22

Virtualization

We’ve seen lots of examples of virtualization. This is another: a virtual machine monitor. A VMM virtualizes an entire hardware machine.

Remarks:

• We can contrast this OS processes and address spaces. In a sense, processes and address spaces present user programs with a virtual machine, but one with a rather different execution environment to raw hardware: system calls instead of hardware devices, signals instead of interrupts, etc.

• The term “virtual machine” is also used to describe highly abstract execution environments, such as the byte code executed by the Java Virtual Machine. In practice, there is a continuum: what these definitions all have in common is that they are interpreters.

• The virtual machines we deal with here are at one end of this spectrum: the execution environment they provide is a simulation of raw machine hardware. This means that, in most cases, the applications than run in this environment are, themselves, operating systems.

Definition 22.1 (Guest operating system). A guest operating system is an OS, plus associated applications, etc. which is running inside a virtual machine.

Definition 22.2 (Hypervisor). Many people draw a distinction between a VMM, and a hypervisor. A VMM is the functionality required to create the illusion of real hardware for a single guest OS – that is, it creates a single virtual machine. A hypervisor is the software that runs on real, physical hardware and supports multiple virtual machines (each with its associated virtual machine monitor).

Remarks:

• In this distinction (which we’ll use), the hypervisor is, for all intents and purposes, an operating system itself.

• In fact, an OS can be extended to act as a hypervisor.
22.1. THE USES OF VIRTUAL MACHINES

Definition 22.3 (Type 1 and type 2 hypervisors). A type 1 hypervisor runs “on the metal”, that is to say, it functions as an OS kernel. In contrast, a type 2 hypervisor runs on top of, or as a part of, a conventional OS like Linux or Windows.

Remarks:
- IBM VM/CMS, VMware ESX, and Xen are all type 1 hypervisors.
- VMware Workstation, kvm, and VirtualBox are all type 2 hypervisors.

These days, there is considerable interest in containers, which share some characteristics with virtual machines, but are actually an example of something different, sometimes known as OS-level virtualization.

Definition 22.4 (OS-level virtualization). Operating system-level virtualization uses a single OS to provide the illusion of multiple instances or containers of that OS. Code running in a container have the same system call interface as the underlying OS, but cannot access any devices.

Remarks:
- System-level virtualization is achieved by limiting the file system namespace (by changing the root for each container), and the process namespace (so processes can only “see” processes which share their container).
- In addition, the OS may provide more sophisticated scheduling and memory allocation policies to allocate to containers rather than simply processes.
- Containers are somewhat limited in functionality compared with virtual machines, but are more efficient (at least at present; this claim is disputed by some researchers who have produced extremely efficient hypervisors). We don’t cover them further in this course.

22.1 The uses of virtual machines

Virtual machines are an old idea, dating back to IBM’s VM/CMS for System/370 in the 1960s. They were revived about 15 years ago, by systems (and companies) like VMware, Xen, Hyper-V, kvm, etc. Why would you want one?

Definition 22.5 (Server Consolidation). The industry (marketing) term server consolidation refers to taking a set of services, each running on a dedicated server, and consolidating them onto a single (probably larger) physical machine so that each one runs in a virtual machine.
Remarks:

- You might ask why these were all running on dedicated machines in the first place. It’s typically for reasons of security and performance isolation, and sometimes because the applications can only run one-per-machine (for a variety of reasons), or can only run on an old hardware configuration.

- Server consolidation is not the first use for VMMs that many people think about, but it was this that really drive the market in the early 2000’s.

**Definition 22.6** (Backward compatibility). **Backward compatibility** is the ability of a new machine to run programs (including operating systems) written for an old machine.

**Example 22.7.** Programs written for Windows XP or earlier versions of Windows can still run on Windows 10, but frequently this is achieved by Windows 10 creating a new virtual machine running XP behind the scenes to run the application.

**Definition 22.8** (Cloud computing). **Cloud computing** is, broadly speaking, the business of renting computing resources as a utility to paying customers, rather than selling hardware.

Remarks:

- Hypervisors decouple allocation of resources (VMs), from provisioning of infrastructure (physical machines)

**Example 22.9.** Amazon Web Services (AWS) is, probably, the world’s largest cloud computing business, and it is primarily based on renting computing resources in the form of virtual machines running customers’ code.

**Definition 22.10** (Resource isolation). When multiple applications contend for resources (CPU time, physical memory, etc.), the performance of one or more may degrade in ways outside the control of the OS. **Resource isolation** is the property of an OS guaranteeing to one application that its performance will not be impacted by others.

Remarks:

- One might think that resource isolation would be a fundamental function of any OS. Performance isolation can be critical in many enterprises (e.g. in cloud computing where real money is at stake).

- In some ways, this is true, but OS mechanism miss the target: UNIX, for example, can sometimes perform resource isolation between processes, but the problem is that an “application” is not a process: many applications are made up of multiple processes, and many processes are servers shared by multiple applications.

- This problem has been recognized for some time (one comment is that UNIX lacks resource containers [BDM99]; however, it was virtual machine monitors that first provided this (in the form of virtual machines).
The above uses are the most common, and the most commercially important, cases where virtual machines are used. There are many more, it turns out:

- OS development and testing
- Recording and replaying the entire state of a machine, for debugging, auditing, etc.
- Sandboxing for security
- Lock-step replication of arbitrary code
- Speculative execution and rollback.

and others.

22.2 Virtualizing the CPU

How can we build an efficient virtual machine monitor? Let’s go through all the resources that need to be virtualized, starting with the CPU.

In a sense, threads or processes virtualize the processor, but only in “user mode”. To run an OS inside a VM, we need completely virtualize the processor including kernel mode.

What happens when the processor executing code in a virtual machine executes a privileged instruction? Obviously it can’t execute it “for real”, but it has to do something.

By default, if the processor tries to execute a privileged operation in user space, the result is a trap or fault. We can use this to catch the attempt to do something privileged and simulate its effect.

**Definition 22.11 (Trap-and-emulate).** Trap-and-emulate is a technique for virtualization which runs privileged code (such as the guest OS kernel) in non-privileged mode. Any privileged instruction causes a trap to the VMM, which then emulates the instruction and returns to the VM guest code.

**Remarks:**

- This might be enough to fully virtualize the CPU over a conventional OS (i.e. the host OS is functioning as the hypervisor): we can run the guest kernel in a regular process in user mode, and use trap-and-emulate to allow the host OS to emulate the effects of privileged operations in the VMM.

- This is enough, if the instruction set is strictly virtualizable.

**Definition 22.12 (Strict virtualizability).** An instruction set architecture (ISA) is strictly virtualizable if it can be perfectly emulated over itself, with all non-privileged instructions executed natively, and all privileged instructions emulated via traps.
Remarks:

- IBM S/390, DEC Alpha, and IBM PowerPC are all strictly virtualizable ISAs.
- Basic Intel x86 and ARM are not.

What goes wrong? Instructions which do the same thing in kernel and user space will work, and instructions which can only be executed in kernel mode will trap and be emulated. The problem is instructions which work in both user space and kernel mode, but do something different depending on the mode.

**Example 22.13.** The *PUSHF* and *POPF* instructions are among 20 or so in the x86 ISA that cannot be virtualized. The push and pop the condition code register, which includes the includes interrupt enable flag (IF). In kernel mode, this really can enable and disable interrupts, but not in user space. In this case, the VMM cannot determine if Guest OS wants interrupts disabled. We can’t cause a trap on a (privileged) *POPF*.

What can we do? There are several solutions, including:

1. Full software emulation
2. Paravirtualization
3. Binary rewriting
4. Change the hardware architecture

**Definition 22.14** (Software emulation). A *software emulator* creates a virtual machine by interpreting all kernel-mode code in software.

Remarks:

- This is, unsurprisingly, very slow - particularly for I/O intensive workloads.
- It is used by, e.g. SoftPC, DosBox, MAME, and other emulators.

**Definition 22.15** (Paravirtualization). A *paravirtualized guest OS is one which has been specially modified to run inside a virtual machine. Critical calls are replaced with explicit trap instruction to VMM.*

Remarks:

- This was used for the first version of Xen, for example [? ]: the authors modified any problematic bits of Linux to explicitly talk to Xen.
- It prevents you from running arbitrary OS binaries in your VMM, because the VMM simply can’t copy with, e.g., non-virtualizable instructions.
- However, it’s fast: almost all commercial VMMs use paravirtualizing to punch holes in the strict VM/VMM interface for performance reasons, usually in special kernel modules loaded at boot time, or custom device drivers.
Definition 22.16 (Hypercall). A hypercall is the virtual machine equivalent of a system call: it explicitly causes the VM to trap into the hypervisor. Paravirtualized VMs use this to ask the hypervisor to do something for them; see below.

Definition 22.17 (Binary rewriting). Virtualization using binary rewriting scans kernel code for unvirtualizable instructions, and rewrites them – essentially patching the kernel on the fly.

Remarks:
- This is generally done on demand: all guest kernel instruction pages are first protected (no-execute). When the VMM takes a trap on the first instruction fetch to the page, it is scanned for all possible instructions that might need rewriting. After patching the code, the page is marked executable and the faulting instruction restarted in the Guest OS.
- VMware uses this approach for x86 virtualization [BDR+12] quite effectively.

Definition 22.18 (Virtualization extensions). An instruction set architecture which cannot be strictly virtualized can be converted into one that is by adding virtualization extensions. This typically takes the form of a new processor mode.

Remarks:
- Both x86 and ARM processor architectures now have additional processor modes which change the behavior of non-virtualizable instructions so that they all trap in guest kernel mode.
- x86 actually has two different ways of doing this, depending on whether you are on an Intel (VT-x) or AMD (AMD-V) processor.
- Hardware support for virtualization often goes beyond merely making the instruction set virtualizable, as we show see below.

22.3 Virtualizing the MMU

So much for the processor, what about the MMU? The guest OS kernel is going to create page tables and install them in the MMU. How do we virtualize this, that is to say, how does the VMM let the guest OS do this and create a result which is, from the point of view of the guest kernel, correct, given that we only have one MMU per core? First, we need some (revised) definitions of addresses.

Definition 22.19 (Virtual address (virtualized)). We define virtual address now to mean an address in a virtual address space created by the Guest OS.

Definition 22.20 (Physical address (virtualized)). We define physical address to mean an address that the guest OS thinks is a physical address. In practice, this is likely to be in virtual memory as seen by the VMM. We let the guest OS create arbitrary mappings between virtual and physical addresses inside its virtual machine.
Definition 22.21 (Machine address). We define a **machine address** to be a “real” physical address, that is, a physical address as seen by the hypervisor. Guest physical addresses are translated into machine addresses, but the guest OS is typically unaware of this extra layer of translation.

What’s happening under the cover is that the hypervisor is allocating machine memory to the VM, and somehow ensuring that the MMU translates a guest virtual address not to a guest physical address but instead to a machine address. The efficiency of this is critical for VM performance.

There are basically three ways to achieve this:

1. Direct writable page tables
2. Shadow page tables
3. Hardware-assisted nested paging

Definition 22.22 (Directly writeable page tables). *In the first approach, the guest OS creates the page tables that the hardware uses to directly translate guest virtual to machine addresses.*

**Remarks:**

- Clearly, this requires paravirtualization: the guest must be modified to do this.
- The VM has to enforce two conditions on each update to a PTE:
  1. The guest may only map pages that it owns
  2. Page table pages may only be mapped RO
- The VMM needs to validate all updates to page tables, to ensure that the guest is not trying to “escape” its VM by installing a rogue mapping.
- In fact, we need more than that: the VMM needs to check **all** writes to any PTE in the system.
- In practice, all page table pages are marked read-only to the guest kernel, and a hypercall is used to modify them.
- We also need to watch for race conditions: the VMM may have to “unhook” whole chunks of the page table in order to be able to apply a set of updates.
- A further hypercall is needed to change the page table base.
- This is naturally expensive (hypercalls can be slow), but many updates to the page tables can be batched up and requested in a single hypercall, which helps somewhat.

Definition 22.23 (Shadow page tables). A **shadow page table** is a page table maintained by the hypervisor which contains the result of translating virtual addresses through first the guest OS’s page tables, and then the VMM’s physical-to-machine page table.
Remarks:

- With shadow page tables, the guest OS sets up its own page tables but these are never used by the hardware.

- Instead, the VMM maintains shadow page tables which map directly from guest VAs to machine addresses.

- A new shadow page table is installed whenever the guest OS attempts to reload the Page Table Base Register (which does cause a trap to the VMM).

- The VMM must keep the shadow table consistent with both the guest’s page tables and the hypervisor’s own physical-to-machine table. It does this by write-protecting all the guest OS page tables, and trapping writes to them. When this happens, it applies the update to the shadow table as well.

- As with direct page tables, this can incur significant overhead, but many clever optimizations can be applied [BDF+03].

**Definition 22.24** (Nested paging). **Nested paging**, also known as **second level page translation** or (on Intel processors) **extended page tables**, is an enhancement to the MMU hardware that allows it to translate through two page tables (guest-virtual to guest-physical, and guest-physical to machine), caching the end result (guest-virtual to machine) in the TLB.

Remarks:

- Most modern processors which support virtualization offer nested paging. It can be significantly faster than shadow page tables (but was not always such).

- Nested paging reduces TLB coverage, the TLB tends to hold both guest-virtual to machine and host-virtual to machine translations (in particular, the ones needed for the guest OS page tables).

- TLB miss costs are correspondingly higher, and a TLB fill (and table talk) can itself miss in the TLB.

- Prior to adding nested paging to x86 processors, neither AMD nor Intel provided tagged TLBs (with address space identifiers). However, the context switch overhead became so high (since the TLB had to be completely flushed even on an intra-guest context switch) that they finally introduced TLB tags to the architecture.

- Whatever its performance trade-offs, nested paging gives hardware designers a performance target, to is likely to improve in the future.

- Nested paging is also much, much easier to write the VMM for. It is the main reason that *kvm* is so small.
22.4 Virtualizing physical memory

That takes care of the page tables and MMU, but what about allocating memory to a virtual machine? A VM guest OS is, typically, expecting a fixed area of physical memory (since that’s what a real computer looks like). It is certainly not expecting it’s allocation of “physical” memory to change dynamically.

In practice, of course, the VM’s “physical” memory is not actually real, but “virtual” memory as seen by the underlying VMM. Not only that, as we saw with virtual memory and processes, the amount of physical memory allocated to a VM should be able to change over time. This leads to two problems:

1. How can the hypervisor “overcommit” RAM, as an OS does with regular processes, and obtain the same dramatic increase in efficiency as a result?
2. How can the hypervisor reallocate (machine) memory between VMs without them crashing?

In theory, this is just demand paging: if the hypervisor demand pages guest-physical memory to disk, it can reallocate machine memory between VMs exactly how an OS reallocates physical memory among processes. The problem is:

Definition 22.25 (Double Paging). Double paging is the following sequence of events:

1. The hypervisor pages out a guest physical page $P$ to storage.
2. A guest OS decides to page out the virtual page associated with $P$, and touches it.
3. This triggers a page fault in the hypervisor, which pages $P$ back into memory.
4. The page is immediate written out to disk and discarded by the guest OS.

Again, this might be fixable using paravirtualization to move paging code out of the guest and into the hypervisor, at the cost of considerable complexity in both the (modified) guest OS and hypervisor. A more elegant solution was created by VMware: ballooning [Wal02].

Definition 22.26 (Memory ballooning). Memory ballooning is a technique to allow hypervisors to reallocate machine memory between VMs without incurring the overhead of double paging. A loadable device driver (the balloon driver) is installed in the guest OS kernel. This driver is “VM-aware”: it can make hypercalls, and also receive messages from the underlying VMM.

Ballooning allows memory to be reclaimed from a guest OS thus (this is called “inflating the balloon”):

1. The VMM asks the balloon driver to return $n$ physical pages from the guest OS to the hypervisor.
2. The balloon driver uses the guest OS memory allocator to allocate $n$ pages of kernel memory for its own private use.
22.5. **VIRTUALIZING DEVICES**

3. It then communicates the guest-physical addresses of these frames to the VMM using a hypercall.

4. The VMM then unmaps these pages from the guest OS kernel, and reallocates them elsewhere.

Deflating the balloon, i.e. reallocating machine memory back to a VM, is similar:

1. The VMM maps the newly-allocate machine pages into guest-physical pages inside the balloon, i.e. page numbers previous handed by the balloon driver to the VMM.

2. The VMM then notifies the balloon driver that these pages are now returned.

3. The balloon driver returns these guest-physical pages to the rest of the guest OS, which can now use them for any purpose.

### 22.5 Virtualizing devices

How do we virtualize devices? That is to say, how do we give each guest OS a set of devices to access?

Recall that, to software, a device is something that the kernel (or the driver at least) communicates with using:

- Memory-mapped I/O register access from the CPU
- Interrupts from the device to the CPU
- DMA access by the device to and from main memory

**Definition 22.27 (Device model).** A device model is a software model of a device that can be used to emulate a hardware device inside a virtual machine, using trap-and-emulate to catch CPU writes to device registers.

**Remarks:**

- Device models emulate real, commonly-found hardware devices, so that the guest OS is already likely to have a driver for.
- “Interrupts” from the emulated device are simulated using upcalls from the hypervisor into the guest OS kernel at its interrupt vector.
- There’s a minimal number of device models that a virtual machine has to support, and it’s not trivial: you need device models for all the interrupt controllers, memory controllers, PCIe bridges, basic console, in short, everything the OS needs in order to minimally boot.
- The best hardware devices to emulate in software are not always the ones you would want “in real life”: a highly sophisticated network card, for instance, would be difficult and slow to emulate, while a very basic card (such as the notorious RTL8139, or the much-loved DECchip 21140 Tulip, emulated in Microsoft Hyper-V) can transfer data to and from the virtual machine more efficiently.
The next logical step after emulated “real” devices, is to create fictitious devices which are only designed to be emulated in a VMM, and write optimized drivers for them for the most popular OSes.

**Definition 22.28** (Paravirtualized devices). A *paravirtualized device* is a hardware device design which only exists as an emulated piece of hardware. The driver of the device in the guest OS is aware that it’s running in a virtual machine, and can communicate efficiently with the hypervisor using shared memory buffers and hypercalls instead of trap-and-emulate.

**Remarks:**

- When using a desktop hypervisor (such as VMware workstation or VirtualBox), it’s common to install the guest OS (particularly if it’s an old one like Windows XP), and then subsequently install a set of “tools” which make its performance improve dramatically. Most of these “tools” are actually paravirtualized drivers.

- It sounds like a good idea to standardize the interface to paravirtualized drivers so that they can be supported by as many hypervisors and guest OSes as possible. It’s such a good idea that almost every hypervisor has standardized them, unfortunately in different ways. The Linux/kvm way of doing this is called *virtio*.

What about the real device drivers, that talk to the real devices?

One option is to put them in the hypervisor kernel (as in, say, VMware ESX). This is fast, but requires the hypervisor itself to natively provide drivers for any device that it supports (much like any other OS).

Alternatively, they can be in a virtual machine, using “device passthrough.”

**Definition 22.29** (Device passthrough). *Device passthrough* maps a real hardware device into the physical address space of a guest OS, allowing it exclusive access to the hardware device as if it were running on real hardware.

**Remarks:**

- Device passthrough is rather more difficult than simply mapping the memory-mapped I/O regions of the device into the virtual machine’s physical address space. For one thing, you need to make sure that the physical addresses the driver hands to its DMA engine are translated into machine addresses before it actually transfers data.

- If you know something about PCI configuration, you’ll also have realized that you need to emulate a whole PCI or PCIe tree just to have a single device direct-mapped into it.

- Nevertheless, it can be done (in the past, three ETH students did this for Barrelfish’s VMM as a semester lab project!).

- Modern hardware has made this a lot easier: IOMMUs (mostly) solve the memory translation problem, and processor support for virtual machines now includes the ability to deliver selected hardware interrupts directly to the virtual machine.
• Device passthrough doesn’t solve the problem of how to share a real device among multiple virtualized guest OSes – all it does is allow a single guest to use the device directly.

**Definition 22.30** (Driver domain). A **driver domain** is a virtual machine whose purpose is not to run user applications, but instead to provide drivers for devices mapped into its physical address space using device passthrough.

**Remarks:**

• The solution to device sharing used by, say, Xen, uses driver domains. A driver domain runs some version of a popular, well-supported kernel (such as Linux), which has drivers for all the devices you might need to support on the real machine.

• The driver domain then runs user-space code which talks to the kernel device drivers, and re-exports a different interface to these devices using inter-VM communication channels.

• For example, the driver domain might export a physical disk by running a file server which other virtual machines might access over “the network”, or alternatively it could export a block interface to virtual disk volumes over another channel. In the client virtual machines, this channel appears as a paravirtualized disk driver.

• Driver domains are great for compatibility, but they are slow for the same reason that poorly-implemented microkernels can be slow: communication between a guest OS kernel and the physical device involves a lot of boundary crossings: from the guest kernel to the hypervisor, into the driver domain kernel, up into user space in the driver domain, and all the way back again.

The commercial importance of virtual machines is demonstrated by the fact that, fairly early on, hardware vendors started working on how to make devices which solved these problems on their own.

**Definition 22.31** (Self-virtualizing devices). A **self-virtualizing device** is a hardware device which is designed to be shared between different virtual machines on the same physical machine by having different parts of the device mapped into each virtual machine’s physical address space.

The most common form of self-virtualizing devices today is SR-IOV.

**Definition 22.32** (Single-Root I/O Virtualization). **Single-Root I/O Virtualization (SR-IOV)** is a extension to the PCI Express standard which is designed to give virtual machines fast, direct, but safe access to real hardware.

An SR-IOV-capable device appears initially as a single PCI device (or “function”, in PCI jargon), known as the **physical function** or PF. This device, however, can be configured to make further virtual functions (VFs) to appear in the PCI device space: each of this is restricted version of the PF, but otherwise looks like a complete different, new device.
Remarks:

• The way this works is that you run a driver for the PF in a driver domain. When a new VM is created, the system asks the driver domain to create a new VF for the VM, and this is then direct-mapped in the new guest’s physical address space.

• You can have quite a few of these. It’s not unusual for a SR-IOV device to support to 4096 VF on a single PF (4096 is the architectural limit). That’s a lot of ethernet cards.

22.6 Virtualizing the network

Networking is a particularly interesting case for virtualization, as it is often the main interface between the virtual machine and the “real” world.

Definition 22.33 (Soft switch). A **soft switch** is a network switch implemented inside a hypervisor, which switches network packets sent from paravirtualized network interfaces in virtual machines to other VMs and/or one or more physical network interfaces.

Remarks:

• In effect, a soft switch extends the network deep into the edge system.

• The soft switch can be quite powerful (including switching on IP headers, etc.) but it needs to be fast: implementations like OpenVSwitch are highly optimized at parsing packet headers.

• There are many different ways of addressing network interfaces inside virtual machines. The most common is to give each virtual network interface a new MAC address, and let DHCP do the rest. However, it’s also common for packets to and from a particular VM to go through a GRE tunnel, so that if the virtual machine is migrated between physical machines the network traffic can follow it.

• With SR-IOV-capable network interfaces, this gets a bit complicated. Many SR-IOV NICs actually have their own hardware switch on board, which switches packets between all the VFs, the PF, and the physical network ports.

Bibliography

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP ’03, pages 164–177, New York, NY, USA, 2003. ACM.

