Computer Systems - Notes Week 7

Ruben Schenk, ruben.schenk@inf.ethz.ch

January 6, 2022

Chapter 13: Virtualization

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

A **guest operating system** is an OS, plus associated applications, etc. which is running inside a virtual machine.

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.

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. For example, IBM VM and VMWare ESX are type 1 hypervisors, and VMWare Workstation and VirtualBox are all type 2 hypervisors.

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.

13.1 The Uses Of Virtual Machines

The industry (marketing) term **server consolidation** refers to a set of services, each running on a dedicated server, and *consolidating* them onto a single physical machine so that each one runs in a virtual machine. **Backward compatibility** is the ability of a new machine to run programs (including operating systems) written for an old machine. **Cloud computing** is, broadly speaking, the business of renting computing resources as a utility to paying customers, rather than selling hardware.

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.

The above uses are the most common, and the most commercially important, cases where virtual machines are used. There are many more as it turns out:

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

13.2 Virtualizing 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 to completely virtualize the processor *including kernel mode.* 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.

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.

An instruction set architecture (ISA) is **strictly virtualizable** IFF it can be perfectly emulated over itself, with all non-privileged instructions executed natively, and all privileged instructions emulated via traps.

Example: The `PUSHF` and `POPF` instructions are among 20 or so in the x86 ISA that cannot be virtualized. They push and pop the condition register, which includes the interrupt enable flag (`IF`). In kernel mode, this really can enable and disable interrupts, but not in user space. In this case, the VMM can't 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:

- Full software emulation: A **software emulator** creates a virtual machine by interpreting all kernel-mode code in software.
- 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 instructions to the VMM. 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.
- Binary rewriting: Virtualization using **binary rewriting** scans kernel code for unvirtualizable instructions, and rewrites them – essentially patching the kernel on the fly.
- Change the hardware architecture: 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.

13.3 Virtualizing The MMU

So much for processor, but 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?

We define **virtual address** now to mean an address in a virtual address space created by the guest OS. 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 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. There are basically three ways to achieve this:

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

In the first approach, the guest OS creates the page tables that the hardware uses to directly translate guest virtual to machine addresses.

- 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:
  - The guest may only map pages that it owns
  - 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

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.

- 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

- The VMM must keep the shadow table consistent with both the guest's page tables and the hypervisors own physical-to-machine table. It does this by write-protecting all the guest OS page tables, and trapping writes to them
- As with direct page tables, this can incur significant overhead, but many clever optimizations can be applied

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

- Most modern processors which support virtualization offer nested paging
- Nested paging reduces TLB coverage, the TLB tends to hold both guest-virtual to machine and host-virtual to machine translations.
- TLB miss costs are correspondingly higher, and TLB fill can itself miss in the TLB
- Nested paging is also much, much easier to write the VMM for

13.4 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. It is certainly not expecting its allocation of "physical" memory to change dynamically. 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 process. The problem is:

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 by using paravirtualization. A more elegant solution was created by VMWare: *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. Thus, ballooning allows memory to be reclaimed from a guest OS:

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.
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-allocated 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.

13.5 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 communicates by 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

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

Remarks:

- Device models emulate real, commonly-found hardware devices, that the guest OS is already likely to have drivers for.
- "Interrupts" from the emulated device are simulated using upcalls from the hypervisor into the guest OS kernel at its interrupt vector.

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.

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 a real hardware.

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.

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 from of self-virtualizing devices today is SR-IOV.

Single-Root I/O Virtualization (SR-IOV) is an 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 appear initially as a single PCI device, known as the *physical function (PF).* This device, however, can be configured to make further *virtual functions (VFs)* to appear in the PCI device space: each of this is a restricted version of the PF, but otherwise looks like a completely different, new device.

13.6 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.

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.