Virtual Machines

- Virtual machine technology, often just called **virtualization**, makes one computer behave as several computers by sharing the resources of a single computer between multiple **virtual machines**.
- Each of the virtual machines is able to run a complete operating system.
- This is an old idea originally used in the IBM VM370 system, released in 1972.
- The VM370 system was running on big mainframe computers that easily could support multiple virtual machines.
- As expensive mainframes were replaced by cheap microprocessors, barely capable of running one application at a time, the interest in virtualization vanished.
- For many years the trend was to replace big expensive computers with many small inexpensive computers.
- Today single microprocessors have at least two cores each and is more powerful than the supercomputers of yesterday, thus most programs only need a fraction of the CPU-cycles available in a single microprocessor.
- Together with the discovery that small machines are expensive to administrate, if you have too many of them, this has created a renewed interest in virtualization.

Uses for Virtual Machines

There are several uses for virtual machines:

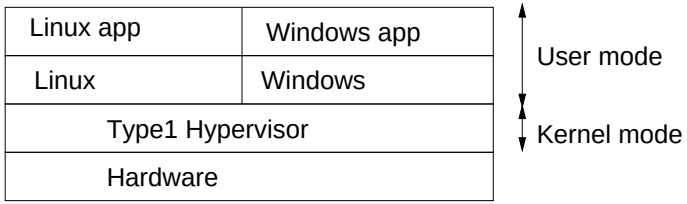
- Running several operating systems simultaneously on the same desktop.
  - → May need to run both Unix and Windows programs.
  - → Useful in program development for testing programs on different platforms.
  - → Some legacy applications my not work with the standard system libraries.
- Running several virtual servers on the same hardware server.
  - → One big server running many virtual machines is less expensive than many small servers.
  - → With virtual machines it is possible to hand out complete administration rights of a specific virtual machine to a customer.

Types of Virtual Machines

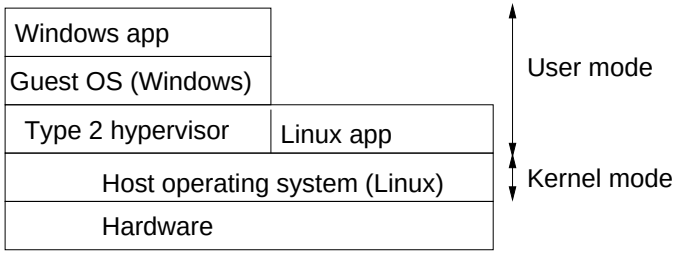
- Type 1 hypervisor (virtual machine monitor or VMM)
  - → The type 1 hypervisor runs at the physical hardware and is the real operating system.
  - → Normal unmodified operating systems, like Linux or Windows runs atop of the hypervisor.
- Type 2 hypervisor
  - → A normal unmodified host operating system like Linux or Windows runs on the physical hardware.
  - → A type 2 hypervisor like VMware Workstation runs on the host operating system.
  - → A normal unmodified guest operating system is booted in the type 2 hypervisor.

Virtual Machine Types

Type 1 hypervisor



Type 2 hypervisor



Requirements for Virtualization

The requirements for virtualization were formally described by Popek and Goldberg in 1974.

“Formal requirements for Virtualizable Third Generation Architectures”, Communications of ACM 17 (7), 1974.

Popek and Goldberg formulated three requirements for a virtual machine monitor (VMM):

Isolation (safety) The VMM manages all hardware resources. The client system is limited to its own virtual space and is unable to determine that it is virtualized.

Equivalence (fidelity) Programs running on the VMM executes identically to its execution on hardware with exception for timing effects.

Performance A majority of guest instructions are executed by the hardware without intervention of the VMM.

Requirements for Virtualization

Definitions used by Popek and Goldberg:

- **Sensitive instructions** are instructions that need to be executed in kernel mode because they do I/O or for example change the MMU settings.
- **Privileged instructions** are instructions that *trap* if executed in user mode.

Popek and Goldberg found that for a processor architecture to meet the requirements for virtualization:

1. A processor architecture must have both *user* and *kernel* mode.
2. The set of *sensitive instructions* is a subset of the *privileged instructions*.

- Expressed in a simpler way, if an instruction is executed in user mode which is not allowed to execute in user mode it should trap to kernel mode.
- Today the term **classically virtualizable** if often used for an architecture that has this property.
- The IBM/370 processor had this property but the Intel 80386 did not have it.

Virtual Machine Implementation

Type 1 hypervisor

- The real operating system is the hypervisor, which runs in kernel mode.
- The guest operating system runs in user mode (sometimes called virtual kernel mode).
- All *non sensitive instructions* executed by the guest execute in the normal way (but in user mode).
- When the guest operating system executes a *sensitive instruction*, a trap occurs and the instruction is emulated by the hypervisor.
  - → If the machine have *sensitive instructions* that do not trap in user mode, the emulation will not work making virtualization impossible.

Virtual Machine Implementation

Type 2 Hypervisor

- Actually, it is possible to virtualize a machine that do not meet the Popek and Goldberg requirements if the guest executes on an interpreter instead of directly on the hardware.
- Interpretation, however , would not meet the performance criteria.
- Combining interpretation with a technique called **binary translation** will however meet all requirements.
- This is how the x86 architecture which is **not classically virtualizable** was actually virtualized by VMware using a type 2 hypervisor.

Type 2 Hypervisor - VMware

- VMware runs as a normal user process atop of an operating system such as Linux or Windows.
- The first time VMware is started, it acts like a newly started uninstalled computer, expecting to find an installation CD in the CD-ROM drive.
- The guest operating system is installed in the normal way on a **virtual disk** (just a file in Linux or Windows).
- Once the guest operating system is installed, it can be booted and run.

The working of VMware

- When VMware executes binary code from a guest system, it first scans the code to find **basic blocks**.
- A **basic block** is a sequence of instructions terminated by a jump, call, trap or other instruction that changes the control flow.
- The basic block is inspected for sensitive instructions and all such instructions are replaced by VMware procedure calls that emulate the instruction.
- When these steps have been taken the basic block is cached inside VMware and executed.
- This technique is called **binary translation**.
- The basic block ends with a call to VMware, that locates the next basic block.
- All basic blocks that do not contain sensitive instructions will run at the same speed as on a bare machine, because they do run on a bare machine.

Paravirtualization

- Both type 1 and type 2 hypervisors works with unmodified guest operating systems.
- An alternative strategy is to modify the guest operating system, so that instead of executing sensitive instructions it makes a procedure call to the hypervisor.
- The method using modified guest operating systems is called **paravirtualization**.
- Paravirtualization can be used to solve the problem with architectures that are not in compliance with Popek and Goldberg but can also be used to improve performance on architectures that are classically virtualizable.
- On architectures such as x86 with many operating systems and many hypervisors, paravirtualization will require a standardized API.
- A proposal from VMware is **VMI (Virtual Machine Interface).**
  - → A VMI library is implemented for each hypervisor. All implementations present the same interface to the guest OS but do different calls to the hypervisor.
- Another such interface is **paravirt_ops** supported by the Linux kernel developers.

X86 Hardware Virtualization

- In late 2005 Intel started to ship some of its x86 processors with VT (Virtualization Technology) extensions.
- The primary goal with the VT extension was to bring x86 into compliance with the Popek and Goldberg criteria, making classical virtualization possible.
- The VT-x extensions introduce a new VMX mode of operation with two new operating states: VMX *root* and VMX *guest*.
- The *root* state is intended for the hypervisor and the *guest* state is for a guest OS.
- Both *root* and *guest* state support all the original privilege levels, making it possible to run a guest OS at its intended privilege level.
- A new *Virtual Machine Control Structure* VMCS specifies the behavior of some instructions in VMX mode.
  - → In VMX mode all sensitive instructions will trap if executed in guest state.
- The processor boots with VMX mode disabled. To activate VMX mode a new *vmxon* instruction is executed and a *vmxoff* instruction will terminate VMX mode.
- AMD have added similar virtualization extensions to their processors, called AMD-V.

Memory Virtualization

- Virtualized operating systems not only share the CPU; they also have to share virtual memory and I/O devices.
- Unfortunately virtualization greatly complicates management of virtual memory.

Example:

- Assume that a guest OS want to mmap virtual pages 3, 4 and 5 to physical pages 10, 11 and 12.
- It will build page tables for this mapping and point a register to the top level page table.
- This instruction is sensitive and will trap on a VT enabled CPU.
- The hypervisor can now allocate physical pages 10, 11 and 12 and set up the page tables to map the guest's pages 3, 4 and 5 to them.
- Now suppose that another guest OS starts and maps its virtual pages 6, 7 and 8 to physical pages 10, 11 and 12.
- Now the hypervisor cannot use this mapping because physical pages 10, 11 and 12 are already in use. It have to find some other free pages and modify the page tables to use them.
- In general, for each guest the hypervisor will have to manage a **shadow page table**.

Memory Virtualization, cont.

- Every time a guest OS changes a page table the hypervisor must change the **shadow page tables.**
- The trouble is that the guest page tables reside in normal memory. No sensitive instruction is needed to write in these pages and no trap to the hypervisor will occur.

A trick is needed to solve this problem:

- When the hypervisor creates the page tables for the guest, it will mark all memory that contain the page tables as read-only.
- Now any attempt from the guest to write in its page tables will generate a *protection fault* and give control to the hypervisor.
- Doing some complicated analysis, the hypervisor will be able to determine that this was an attempt to update a page table and it will update the *shadow page tables* accordingly.
- A paravirtualized guest is in a much better position here. It will know that it is virtualized and will do a procedure call to inform the hypervisor that it has changed its page table. This is more efficient than generating lots of protection faults an have the hypervisor guessing what is going on.

Hardware support for Memory Virtualization

- For a few years AMD have shipped processors with hardware support for memory virtualization.
- The AMD virtual memory support for virtual machines is called NPT (Nested Page Tables).
- Also Intel has announced virtual memory hardware support called EPT (extended page tables).
- In processors with NPT another level of translation is added in the hardware MMU.
  - → The guest page table converts between *guest virtual address* and *guest physical address*.
  - → A new nested page table under control of the hypervisor converts between *guest physical address* and *system physical address*.
- Using NPT the guest OS will be in full control of its page tables.
- The price to pay for this is that the translation takes more time than with the original hardware and that the caching of the translations in the TLB is less efficient.

I/O Virtualization

When virtualizing I/O devices, at least the following interactions between software and devices must be handled:

Device discovery: A method must be provided for the guest OS to find out which I/O devices are present.

Device control: Special I/O instructions or memory mapped registers can be used. I/O instructions are sensitive and will trap but memory mapped registers are not sensitive unless read/write protected.

Data transfers: Most devices use DMA which in most cases uses absolute physical addresses. The guest OS will however use guest physical addresses which need to be translated by the hypervisor for DMA to work.

I/O interrupts: A method is needed to relay interrupts for guest initiated data transfers to the guest.

I/O Virtualization

- A common way for a hypervisor to handle I/O devices is by **emulation**.
- The hypervisor implements a software model of the I/O device including all the control registers.
- If the device use memory mapped I/O, the memory addresses for the emulated registers must be protected by the hypervisor to make operations on them trap.
- The guest OS will believe that it is talking to a hardware device, when in fact it is communicating with a software model.
- The hypervisor may be accessing a real hardware when emulating the device, but the emulated device may be different from the hardware device. For example an IDE disk may be emulated while the real hardware is a SATA disk.
- Using emulation, the hypervisor can present many more devices to the guest systems than are physically present on the machine.
- A type 2 hypervisor will use the device drivers in the host OS to access the real hardware devices while a type 1 hypervisor will need to develop its own device drivers for all hardware that are present on the machine.
- The disadvantage with emulation is that performance suffers when every operation on an I/O register will generate a timeconsuming trap to the hypervisor. Here **paravirtualization** can give much better performance.

Hardware support for I/O Virtualization

- The use of DMA is a problem with virtual machines because a DMA controller is able to write to the entire physical memory, not just the memory assigned do a single guest OS.
- For this reason a guest OS is not allowed to handle the DMA controller itself, but must call for the hypervisor to do it.
- To solve this problem both AMD and Intel have added IOMMU:s (I/O Memory Management Unit) to some recent processors.
- An IOMMU will use a translation technique similar to a virtual memory page table to translate between *guest physical addresses* and *absolute physical addresses*.
- The IOMMU will also restrict which physical addresses a device may access.
- Using an IOMMU a guest OS may be allowed to initialize a DMA transfer itself, however a call to the hypervisor is needed to set up the IOMMU mapping.
- Like a virtual memory system needs a TLB (Translation Lookaside Buffer) to get reasonable performance, an IOMMU will need an IOTLB.

Some Existing Virtual Machine Softwares

- VMware Workstation, WMware Player, WMware server
  - → Type 2 hypervisor using binary translation
  - → VMware Workstation 6.0 also supports paravirtualization (VMI)
  - → Proprietary, Free download for Player and server
  - → Host and guest CPUs: X86, AMD64
  - → Host OS: Windows and Linux
- QEMU
  - → Emulator using dynamic recompilation
  - → Open source
  - → Many different host and guest CPUs
  - → Host OS: Linux, Mac OS X, Solaris, FreeBSD, OpenBSD, Windows
- KQEMU
  - → Linux kernel module to accelerate QEMU
  - → Type 2 hypervisor using dynamic recompilation
  - → Host and guest CPUs: x86 and x86-64
  - → Host OS: Linux (experimental versions for FreeBSD and Windows XP)

Some Existing Virtual Machine Softwares, Cont.

- Virtualbox (Sun)
  - → Type 2 hypervisor
  - → Partially based on QEMU
  - → Both open source and proprietary versions (free download)
  - → Host and guest CPUs: x86 and x86-64
  - → Host OS: Windows, Linux, Mac OS X, Solaris
- Xen
  - → Type 1 hypervisor
  - → Open source
  - → Host CPUs: x86 with Intel VT, AMD64 with AMD-V
  - → Host OS: Linux. FreeBSD, Solaris
  - → Runs on any x86 processor with Linux and FreeBSD using paravirtualization