Notes on Virtual Machines
15-440, Fall 2011
Carnegie Mellon University
Randal E. Bryant

References:

Tannenbaum, 3.2

Barham, et al., "Xen and the art of virtualization," SOSP ’03

Rosenblum & Garfinkel, "Virtual machine monitors: current technology and future trends," IEEE Computer, May 2005

Adams & Agesen, "A comparison of software and hardware techniques for x86 virtualization," ASPLOS ’06

Concept of Virtualization:

Make it possible for multiple users to share a set of physical resources in way that each appears to have complete control over these resources.

Example: Operating systems provide a virtualization of a computer system to multiple application programs.

Virtualized resources:

CPU: When application runs, it has complete control over the CPU. Update to register by one process doesn’t affect value of that register for other processes. This is done by context switching---CPU state for each active process saved in memory, loaded when schedule process and stored when switch to different process. (Multicore processors allow multiple physical CPUs to be active simultaneously, but not generally visible to application how these physical resources are managed.)

Memory: Virtual memory makes it look like each process has its own complete linear address space. Write by one process does not affect value read by another, unless we have set them up to have shared state. This is managed by software (setting up page tables), the MMU (translating memory references based on page tables), and the TLB (serving as a cache for address translations.)

Network: Only partially virtualized. All processes share a common IP address and port mapping. E.g., can only have one process listening on port 80.

Disk: Not generally virtualized. All process make use of common file system.

Other I/O devices: Not generally virtualized.

Note: the one resource that can never be fully virtualized is time. Measure different "time" when using wall clock vs. execution time for process.

Virtual Machines:

* Take idea one step further. Allow virtualizing at the OS level. Want to run multiple OS’s on single machine, each with its own set of applications.

* Idealized image: Virtual machine manager (VMM), a.k.a, "hypervisor" runs on host machine, either directly ("Type I") or on top of host operating system ("Type II"---common case today). Allows multiple unmodified "guest" OSs to run on top. Each guest OS operates exactly the same as if it were executing directly on the host hardware.

* Motivation:

  - Expanded capabilities. Want to use one machine to support multiple OS's simultaneously. E.g., Parallels program makes it possible to run Windows applcations on Macintosh. Does this by creating a VMM with MacOS as host, and Windows (or Linux) OS as guest. Can be done efficiently partly because both MacOS & Windows run on x86, so can directly execute most of the Windows & application code.

  - Resource sharing. Want to schedule many different programs onto a collection of machines, while providing high degree of isolation between them. Example: Amazon Web Services. Companies can rent virtualized machines by the hour. Use to run large data centers. Advantage of renting vs. buying: can dynamically change number of machines according to load, and let others deal with system administration. Much cheaper than buying & operating enough machines to meet peak load: rely on averages, and cost / CPU less than purchase & operate cost.

  (This was the original motivation for virtualization. IBM had big mainframe computers. Wanted to have multiple clients, each using OS of its choice, on single machine.)

  - Security. Example: Autolab system used by multiple courses for student grading. Runs user program within virtual machine that allows incoming network connection, but no outgoing one. Virtualization makes it impossible for student program to modify any files or initiate network connections.

  - Flexibility. Can migrate complete "system" from one machine to another, including the OS and all applications running on it.

  - Testing / debugging. E.g., Jiri Simsa's dBug program. Runs many tests on threaded program to look for concurrency bug. Each time, reloads image of machine's starting point and then traces through new test sequence. VM provides simple way to checkpoint entire machine state and restart from saved state.

How it works.

Brief overview of non-virtualized machines:

1. OS consists of two parts: the kernel, which must be resident in memory at all times & a set of libraries, which provide useful services on top of the kernel. Example, Linux read & write functions execute at the kernel level. I/O routines such as fprintf, are implemented by libraries that do not execute at kernel level. Instead, they call read & write.

2. Machine state & instructions are either protected or generally accessible (for update.) Examples of protected resources:

* Page tables
* State of other processes
* Some of the registers (e.g., time stamp counter. Status registers)

Typically, only OS kernel can access the protected state & instructions. Kernel + OS libraries + applications make extensive use of user-accessible state & instructions.

Interfaces:

* Hardware: instructions, special registers, physical memory.

* System call and traps: interface between kernel and libraries + applications. E.g., x86 has syscall instruction, which passes information via regular registers, including operation to be performed.

* ABI: What compiler sees: general instructions + calls to library

functions.

Our challenge: Present the hardware interface to a guest OS via a VMM.

One easy way: have VMM implement an instruction-set interpreter. Simulates what would happen if instructions were really running on hardware. E.g., this is what the SIMICS system in 15-410 does. But, this runs ~100X slower than native execution of instructions.

Desired properties of virtualization (Popek & Goldberg, 1974):

1. Fidelity: Sofware running on VMM behaves exactly as it would on hardware (except for time effects.)

2. Performance: Very large majority of guest instructions are executed by the hardware without intervention by the VMM. This is referred to as "direct execution." Ideal is to have same performance as if operating directly on hardware.

3. Safety: The VMM manages all hardware resources.

Classic implementation. Hardware support via "trap and emulate"

Run with privilege levels:

VMM > Guest OS > Application

What this means:

* Can mark memory pages with privilege level & permission (e.g., read/write/execute).

* Memory access or instruction (e.g., syscall) that requires higher privilege will "trap": like a procedure call to the (host) kernel. Kernel code determines what resource was being accessed by whom, and passes control to appropriate handler. E.g., if application invoked syscall, then pass control to guest kernel. If guest kernel attempted to update host page table, then pass control to VMM. When done, return control to original process. A.k.a. "trap and emulate."

  As example, consider syscall to disable interrupts. Don't want guest to be able to actually turn off all hardware interrupts. Instead, VMM makes entries in a table it manages to disable all interrupts from that VM.

  Virtual memory is a bit trickier. Each guest OS maintains the set of page tables that it would use if it were managing the memory. These are referred to as "shadow page tables." Want to make sure that host OS tracks any changes to these shadow page tables in order to maintain the true page tables for the actual system. VMM marks the page table entries in the shadow page tables so that pages are read-only. When guest OS tries to modify them, this will cause a fault, giving control to VMM. VMM then looks at shadow table updates and updates its own.

  VMM time slices the VMs using simple scheme, such as round robin.

* Requires hardware designed specifically to support virtualization. Commonplace for traditional, high-end machines, but not for x86.

x86 virtualization challenges:

* User processes can read status register that indicates privilege level.

* fpopc (pop flags) instruction behaves differently when executing in user mode than privileged mode, but never traps. Instruction designed to read collection of flags from top of stack and set CPU status accordingly. One flag causes interrupts to be disabled. When executed in user mode, this flag is ignored.

* Hardware MMU. Difficult to maintain a set of "shadow pages" for each VM, and keep synchronized with actual pages of running process.

* Untagged TLB. Must do complete flush every time do context switch.

How to deal with this:

1. Paravirtualization. E.g., used by Xen. Requires source code modifications of guest OS. Replace non-virtualizable instruction by function call, e.g., for fpopc. Only works when have access to source code.

2. Binary translation. E.g., VMWare. Transform executable code into new form, with problematic instructions replaced by either function calls or new code sequences.

This is really hard to do, especially with x86 code.

Rough steps:

1. When first encounter code entry point, start interpreting instructions until hit branch. Call this a "basic block."

2. Set up new copy of sequence, replacing any problematic instructions with function calls. Terminate with two-way branch: one to jump target, other to fall-through block.

3. Must do this dynamically, since cannot determine entry points in advance, & since x86 code can get generated dynamically (e.g., from buffer overflow).

4. Certain parts are tricky, since layout of code different from before. E.g., PC-relative addresses.

5. Once have this capability, can fix code to interoperate properly with VMM.

Has advantage that it works with any guest OS. Don’t need to have access to source code. Performance (as reported by Dave O’Hallaron: < 5% penalty for CPU-intensive programs, ~ 15% for network-intensive ones.)

Recently: both AMD & Intel have added hardware support for virtualization.

1. Introduce different privilege levels: host, guest, user.

2. Define guest VM control state, called "virtual machine control block" (VMCB).

3. Provide instructions vmrun & exit. Vmrun loads state from VMCB and sets privilege level to guest. Exit stores state in VMCB and returns back to host VMM.

This has made virtualization much easier to implement. E.g., Parallels, Linux KVM. VMWare claims that it is less efficient, since changing privilege level is still very costly.

A few fine points:

1. Networking. Want to truly virtualize. Each VM gets its own MAC address. Can invoke DHCP to assign own IP address. Host OS + VMM must emulate a router: route incoming packets to appropriate VM.

2. Time: What time should we report to the guest OS? Find out that need both real time & virtual time:

* E.g., to judge timeout or RTT from network connection, need real time. To manage scheduling of jobs, want virtual time.

Conclusion:

We have seen 3 approaches to virtualization:

1. Hardware support. Make it possible to support trap & emulate

2. Paravirtualization. Modify guest OS to make calls to VMM rather than attempt to access resource directly.

3. Binary translation. Like paravirtualization, but with modified guest OS code generated automatically from original binary, rather than through manual rewrite. Useful when don’t have source code access. Also enables more adaptive translations to improve performance.