Week 3: Virtual machines & Modules

CSC 469 / CSC 2208
Fall 2019
Angela Demke Brown
Overview

• Virtual Machines

• Kernel Extensions

NOTE: VIRTUALIZATION SLIDES ORIGINALLY APPEARED IN LECTURE 2, BUT HAVE BEEN MOVED HERE FOR CONSISTENCY WITH THE LECTURE SCHEDULE.
ADDRESS TRANSLATION DIAGRAMS (SLIDES 47 & 49) HAVE BEEN ADDED.
Wednesday, Sept 18

• Announcements
  • Assignment 1
    • Will be posted tomorrow morning or evening
    • Due date extended to Tuesday, Oct. 1
VM Examples

• Original – IBM’s VM/CMS (1970’s)

• Now hot again:
  • Disco (Stanford research, 1997) ➔ VMWare
  • Xen (Cambridge, 2003)
  • Linux KVM (kernel virtual machine, as of 2.6.20, 2007)
  • VirtualBox (Innotek GmbH, 2007 ➔ Sun ➔ Oracle)
  • Hyper-V (2008, Microsoft)

• What’s the big deal about virtual machines?
What is a virtual machine?

• An efficient, isolated, duplicate of the real machine
  • Popek & Goldberg, 1974 “Formal Requirements for Virtualizable Third Generation Architectures”
  • Provided by “virtual machine monitor” with three essential characteristics:
    • Transparency: Essentially identical execution environment (as real machine)
    • Efficiency: Minor performance penalty for programs in VM
    • Resource Control: VMM has complete control over system resources

• Software added to the execution platform to give the appearance of a different platform or multiple platforms
  • Smith & Nair, 2004 “Virtual Machines”
Why virtual machines?

• Original motivation in 1960’s
  • Large, expensive computers shared by many users
  • Different groups wanted or needed different operating systems
  • Convenient timesharing mechanism (each user gets own virtual machine)

• Today’s motivation?
  • Large scale servers - similar as original motivation
  • Security
  • Reliability/fault tolerance
  • Portability/compatibility
  • Avoid dealing with multiprocessor issues in OS
  • Migration
  • Performance
  • Innovation
Types of virtual machines

- Many uses of the term “virtual machine”
  - A matter of perspective (process, OS)
- Conventional software is developed/compiled for a specific OS and ISA
  - Application binary interface (ABI): interface between a process and the machine
  - Instruction set architecture (ISA): instructions that can be executed; real machine vs. virtual machine

Source: Smith & Nair – Virtual Machines: Versatile Platforms for Systems and Processes
Types of virtual machines

• Virtualization – 2 parts:
  • Map virtual resources (registers, mem, files, etc.) to real resources
  • Use real machine instructions or syscalls on the host OS, to carry out instructions or syscalls specified by the VM
  • Virtualization software must emulate the virtual machine ABI or ISA

• Distinguish virtual machines based on whether they virtualize the ABI or the ISA

• **Process virtual machines** provide virtual ABI
  • Created and destroyed along with the process they run

• **System virtual machines** provide a complete system environment
  • Multiple user processes, file system, I/O, GUI, etc.
Virtualization Software

Process Virtual Machine

Virtualization software = Runtime

System Virtual Machine

Virtualization software = VMM
System VMs

• “classic” VMM (type I)
  • VMM runs on bare hardware, everything else runs on top
    • VMM is most privileged software, everything else less privileged
  • E.g. VMware ESX, Xen

• “hosted” VMM (type II)
  • Virtualizing software installed on top of existing OS
    • E.g. VMWare Workstation, Linux KVM

Requirements for Virtualizability

• Architecture requirements
  • Dual mode operation
  • A way to call privileged operations from non-privileged mode
  • Memory relocation / protection hardware
  • Asynchronous interrupts for I/O to communicate with CPU
    • Goldberg, 1972

• VMM must provide 3 primary functions
  • Interpreter (“Virtualizing the computer”)
  • Dispatcher component
  • Allocator
Instruction Requirements

• Virtualizing is easy if all instructions are virtualizable.

• **Privileged instructions**: required to trap if not executed in supervisor mode

• **Sensitive instructions**: affect the operation of the system in some way

• THEOREM: An efficient VMM may be constructed if the set of sensitive instructions is a subset of the set of privileged instructions

• Intel Pentium: 17 instructions are sensitive but not privileged (Robin & Irvine, USENIX Security 2000)
  - VMware used binary rewriting to deal with this
  - Xen required changes to the OS → *paravirtualization*
  - Intel VT, AMD-V (Pacifica) fixed this
Disco

• “Disco: Running Commodity Operating Systems on Scalable Multiprocessors”, Edouard Bugnion, Scott Devine, and Mendel Rosenblum, SOSP’97

• Goals
  • Extend modern OS to run efficiently on shared memory multiprocessors without large changes to the OS
  • VMM can run multiple copies of Silicon Graphics IRIX operating system on a Stanford Flash shared memory multiprocessor

• Revitalized interest in virtual machine concept
Problem

• Commodity OS's not well-suited for ccNUMA (1997)
  • Do not scale: Lock contention, memory architecture
  • Do not isolate/contain faults: more processors => more failures

• Customized operating systems
  • Take time to build, lag hardware
  • Cost a lot of money

• ➔ Reduce the gap between H/W innovation and release of adapted system S/W
Solution

• Add a virtual machine monitor (VMM)
  • Commodity OSes run in their own virtual machines (VMs)
  • Communicate through distributed protocols
• VMM uses global policies to manage resources
  • Moves memory between VMs to avoid paging
  • Schedules virtual processors to balance load
Advantages

• Scalability
• Flexibility
• Hide NUMA effect
• Fault Containment
• Compatibility with legacy applications
VM challenges

- Overheads
  - Instruction execution, exception processing, I/O
- Memory
  - Code and data of hosted operating systems
  - Replicated buffer caches
- Resource management
  - Lack of information
    - Idle loop, lock busy-waiting
    - Page usage
- Communication and sharing
  - Not a problem → use distributed protocols
Disco interface

- VCPUs provide abstraction of a MIPS R10000 processor
  - Emulates all instructions, the MMU, trap architecture
  - Enabling/disabling interrupts, accessing privileged registers -> Memory-based interface to VMM

- Physical memory
  - Contiguous address space, starting at address 0
  - Physical-to-machine address translation, second (software) TLB
Guest attempts to load TLB with VPN->PPN translation, privileged operation traps to Disco.

Disco loads real TLB with VPN->MPN translation.
Disco interface (cont’d)

- I/O devices
  - Virtual devices exclusive to VM
  - Physical devices multiplexed between virtual ones
  - Special interface to SCSI disks and network devices
  - Interpose on DMA calls

- Disk:
  - Set of virtualized disks to be mounted by VMs
  - Copy-on-write disks; for persistent disks, uses NFS

- Network:
  - Virtual subnet across all virtual machines
  - Uses copy-on-write mappings => reduces copying, allows sharing
Virtualizing x86 Address Translation

- Shadow page tables track changes to guest page tables, mapping guest VPN to MPN. How?
- Guest changes to virtual CR3 translated to VMM changes to real CR3
Xen virtualization

• Technically, two kinds
• Paravirtualization
  • Guests run a modified OS
    • Introduced direct paging:
      • guest OS can read PPN to MPN map, maintains VPN to MPN translation in page tables
      • Xen still controls actual writes to page tables
    • High performance on x86
• Hardware-assisted virtualization
  • CPUs that support virtualization
  • Unmodified guest OSes
Xen infrastructure

- Control Plane Software
- User Software
- User Software
- User Software

- GuestOS (XenoLinux)
  - Xeno-Aware Device Drivers

- GuestOS (XenoLinux)
  - Xeno-Aware Device Drivers

- GuestOS (XenoBSD)
  - Xeno-Aware Device Drivers

- GuestOS (XenoXP)
  - Xeno-Aware Device Drivers

Domain0 control interface
- virtual x86 CPU
- virtual phy mem
- virtual network
- virtual blockdev

H/W (SMP x86, phy mem, enet, SCSI/IDE)
Figure 3: Relative performance of native Linux (L), Xen/Linux (X), VMware Workstation 3.2 (V) and User-Mode Linux (U).

From: “Xen and the art of virtualization” Barham et al.
How does it compare to Disco?

- Three main differences
  - Less complete virtualization
  - *Domain0* to initialize/manage VMs, incl. to set policies
  - Strong performance isolation

- Other
  - Interface is pretty close to hardware and enables low-overhead high-performance virtualization
  - Need to change more OS code than in Disco

- All the cool details are in the paper!
  - “Xen and the art of virtualization” – SOSP’03
Hypervisors for servers

• Type 1 or Type 2?

• Hyper-V: “MicroKernelized” Hypervisor Design

• VMWare ESX Server: “Monolithic” hypervisor architecture
Monolithic Hypervisor

- VM1 (Admin)
- VM2
- VMn

- Virtualization Stack
- Hypervisor
- Drivers

Hardware

Microkernel Hypervisor

- VM1 (Admin)
- VM2
- VMn

- Virt Stack
- Drivers

- Drivers

Hardware

- Both true Type 1 hypervisors – no host OS
- The hardware is the physical machine; OSs are all virtual
Friday: OS Extensions

• Adding new function to OS “on the fly”

• Why?
  • Fixing mistakes
  • Supporting new features or hardware
  • Efficiency / Custom implementations

• How?
  • Give everyone their own machine (VMs)
  • Allow some OS function to run outside (ukernel)
  • Allow users to modify the OS (e.g., modules)
Loadable Kernel Modules

• Giving everyone a virtual machine doesn’t entirely solve the extension problem
  • You can run what you want on your VM, but do you really want to write a custom OS?
• Often just want to modify/replace small part
• Solution: Allow parts of the kernel to be dynamically loaded / unloaded
  • Requires dynamic relocation and linking
• Common strategy in monolithic kernels for device drivers (FreeBSD, Windows NT/2K/XP, Linux)
Linux Loadable Kernel Modules

- Module writer must define (at least) two functions
  - `init_module` – code executed when module loads
  - `cleanup_module` – code executed when module unloads
- Module functions can refer to any exported kernel symbols
- Module is compiled into relocatable `.ko` file (since 2.6)
- `insmod` command loads module into running kernel
  - 2.4 – `insmod` (at user level) resolves references to kernel symbols
  - 2.6 – invokes syscall, kernel does the linking
- `rmmod` command removes module from kernel
- `lsmod` command lists currently-installed modules
- `modprobe` is a library wrapper that checks module dependencies and loads additional required modules
Tracking Modules

- Kernel has a linked list of module objects
- struct contained in the module memory itself

```c
enum module_state state;
struct list_head list;
char name[MODULE_NAME_LEN];
...
/* What modules depend on me? */
struct list_head source_list;
/* What modules do I depend on? */
struct list_head target_list;
atomic_t refcount;
...
```
rmmod

• Unlinks module from kernel
• Needs to ensure no one is using module first!
  • Reference count incremented whenever module is used
  • source_list identifies other modules that depend on this one
• Invokes module-provided exit / cleanup function
• Frees memory
Problems with module approach

- Requires stable interfaces
  - Linux uses version numbers to check if module is compiled for correct version of kernel, but it is easy to get this wrong
- Unsafe
  - Module code can do *anything* because it runs privileged
    - E.g. VMWare Workstation driver
      - “hijacks” machine by changing *interrupt descriptor table (IDT)* base register and then jumps to code in the VM application!
Alternate kernel-level schemes

- Trusted compiler (or certification authority) + digital signatures
  - Allows verification of source of code added to kernel
  - You still have to decide if you trust that source
  - Code can still do anything
- Proof-carrying code
  - Code Consumer (OS) supplies a specification for what extensions are allowed to do
  - Code Producer (the extension) must supply a proof that it is safe to execute according to specification
  - OS validates proof
  - Proof should be easy to check, but may be hard to generate (e.g. maze example)
• G. Necula - Safe Kernel Extensions Without Run-Time Checking, OSDI’96
• A maze is “safe” if there’s a path through it. => Easy to check a path, but hard to generate.
Alternates (2)

- **Sandboxing (software fault isolation)**
  - Limit memory references to per-module segments
  - Check for certain unsafe instructions

- **Examples:**
  - SPIN (U. of Washington)
    - Modula-3 + trusted compiler
    - Safety properties provided by language
    - Problems with dynamic behavior (e.g. “while(1)”)
  - Vino (Harvard)
    - Sandboxed C/C++ code called “grafts”
    - Timeouts to guard against misbehaved grafts
    - Resource limits + transactional “undo”
  - Byte-Granularity Isolation (Microsoft) - BGI
eBPF

- “extended Berkeley Packet Filters”
- Language-level VM within Linux kernel
  - Register-based VM
  - Custom 64-bit RISC instruction set
  - Bytecode verifier
- Restrictions are placed on eBPF programs for safety
  - Limited number of instructions
  - Controlled memory referencing
  - Originally, no loops allowed

  • Bounded loops are available starting with Linux 5.3!