Virtualization

Mike Kasick
Glenn Willen
Mike Cui

15-410: Operating System Design & Implementation

April 16, 2007

Outline

Outline

What is Virtualization?

- Virtualization:
  Process of presenting and partitioning computing resources in a *logical* way rather than what is dictated by their *physical* reality

- Virtual Machine:
  An execution environment identical to a physical machine, with the ability to execute a full operating system

Advantages of the Process Abstraction

- Each process is a pseudo-machine
- Processes have their own registers, address space, file descriptors (sometimes)
- Protection from other processes

Disadvantages of the Process Abstraction

- Processes share the filesystem
- Difficult to simultaneously use different versions of:
  - Programs, libraries, configurations
- Single machine owner:
  - root *is* the superuser
  - Which “domain” does a machine belong to?

Disadvantages of the Process Abstraction

- Processes share the same kernel
  - Kernel/OS specific software
  - Kernels are *huge*, lots of possibly unstable code
    - AFS client fallover?
- Processes have limited degree of protection, even from each other
  - OOM killer?

Why Use Virtualization?

- Process abstraction at the kernel layer
  - Separate filesystem
  - Different machine owners
- Offers much better protection (in theory)
  - Secure hypervisor, fair scheduler
  - Interdomain DoS? Thrashing?

Why Use Virtualization?

- Run two operating systems on the same machine!
- Huge impact on enterprise hosting
  - No longer have to sell whole machines
  - Sell machine slices
  - Can put competitors on the same physical hardware

Why Use Virtualization?

- With NAS, can separate instance of VM from instance of hardware
- Live migration of VM from machine to machine
  - No more maintenance downtime
- VM replication to provide fault-tolerance
  - Why bother doing it at the application level?

Disadvantages of Virtual Machines

- Attempt to solve what really is an abstraction issue somewhere else
  - Monolithic kernels
  - Not enough partitioning of global identifiers
    - pids, uids, etc
- Draws a box around “the problem”:
  - Still hard to solve “the problem”
  - Relatively easy to manipulate the box (the VM)

Disadvantages of Virtual Machines

- Feasibility issues
  - Hardware support? OS support?
  - Admin support?
  - VMware ESX seems to be doing the job well
- Performance issues
  - Is a 10-20% performance hit tolerable?
  - Can your NIC or disk keep up with the load?

Outline

Full Virtualization

- IBM CP-40 (later CP/CMS & VM/CMS) (1967)
  - Supported 14 simultaneous S/360 virtual machines.
- Popek & Goldberg: Formal Requirements for Virtualizable Third Generation Architectures (1974)
  - Defines characteristics of a *Virtual Machine Monitor*
  - Describes a set of architecture features sufficient to support virtualization

Virtual Machine Monitor

1. Equivalence:
   Provides an environment essentially identical with the original machine
2. Efficiency:
   Programs running under a VMM should exhibit only minor decreases in speed
3. Resource Control:
   VMM is in complete control of system resources

Popek & Goldberg Instruction Classification

1. Privileged instructions:
   - Trap if the processor is in user mode
   - Do not trap if in supervisor mode
2. Sensitive instructions:
   - Attempt to change configuration of system resources
   - Illustrate different behaviors depending on system configuration

Popek & Goldberg Theorem

"... a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions."

- All instructions must either:
  - Exhibit the same result in user and supervisor modes
  - Or, they must trap if executed in user mode
- Architectures that meet this requirement:
  - IBM S/370, Motorola 68010+, PowerPC, others.

Outline

x86 Virtualization

- x86 ISA does not meet the Popek & Goldberg requirements for virtualization
- ISA contains 17+ sensitive, unprivileged instructions:
  - `SGDT`, `SIDT`, `SLDT`, `SMSW`, `PUSHF`, `POPF`, `LAR`, `LSL`, `VERR`, `VERW`, `POP`, `PUSH`, `CALL`, `JMP`, `INT`, `RET`, `STR`, `MOV`
  - Most simply reveal the processor's CPL
- Virtualization is still possible, requires a workaround

VMware (1998)

- Runs guest operating system in ring 3
  - Maintains the illusion of running the guest in ring 0
- Insensitive instructions execute as is:
  - `addl %ecx, %eax`
- Privileged instructions trap to the VMM:
  - `cli`
- Performs binary translation on guest code to work around sensitive, unprivileged instructions:
  - `popf` $\Rightarrow$ `int $99`

VMware (1998)

Privileged instructions trap to the VMM:

```
cli
```

actually results in:

`int $13` (General Protection Fault)

which gets handled:

```
void gpf_exception(int vm_num, regs_t *regs)
{
    switch (vmm_get_faulting_opcode(regs->eip))
    {
        ...
        case CLI_OP:
            vmm_defer_interrupts(vm_num);
            break;
        ...
    }
}
```

VMware (1998)

A sensitive, unprivileged instruction:

`popf` (restore `%EFLAGS` from the stack)

we would like to result in:

`int $13` (General Protection Fault)

but actually results in:

`%EFLAGS` $\leftarrow$ all bits from stack except IOPL

VMware (1998)

So, VMware performs *binary translation* on guest code:

```
popf
```

VMware translates to:

`int $99` (popf handler)

which gets handled:

```
void popf_handler(int vm_num, regs_t *regs)
{
    regs->eflags = regs->esp;
    regs->esp++;
}
```

Hardware Assisted Virtualization

- Recent variants of the x86 ISA that meet Popek & Goldberg requirements
  - Intel VT-x (2005), AMD-V (2006)
- VT-x introduces two new operating modes:
  - VMX root operation & VMX non-root operation
  - VMM runs in VMX root, guest OS runs in non-root
  - Both modes support all privilege rings
  - Guest OS runs in (non-root) ring 0, no illusions necessary

Hardware Assisted Virtualization

- VT-x defines two new processor transitions:
  - VM entry: root $\Rightarrow$ non-root
  - VM exit: non-root $\Rightarrow$ root
  - Guest instructions & interrupts that result in a VM exit are specified by the virtual-machine control structure (VMCS)
  - `movl %eax, %cr0` $\Rightarrow$ VM exit $\Rightarrow$ VMM sets `%CR0` $\Rightarrow$ VM entry

VT-x in the Real World

- Supports virtualization of all of x86 protected mode
  - All rings, descriptor tables, page tables, etc
- Requires paging
  - Real mode & protected mode without paging is unsupported and must be emulated by the VMM
- Most OSes only use a subset of x86 features
  - Two rings, a few segments, etc
- Binary translation necessary to support x86 feature subset actually used by OSes is faster than the full-blown hardware solution

Paravirtualization (Denali 2002, Xen 2003)

- First observation:
  - Most commodity OSes are open source[1]
  - OSes can be modified at the source level to supported limited virtualization
- Paravirtualizing VMMs (hypervisors) virtualize only a subset of the x86 execution environment
- Run guest OS in rings 1–3
  - No illusion about running in a virtual environment
  - Guests may not use sensitive, unprivileged instructions and expect a privileged result
- Requires source modification only to guest kernels
  - No modifications to user level code and applications

[1] One notable exception

Paravirtualization (Denali 2002, Xen 2003)

- Second observation:
  - Regular VMMs must emulate hardware for devices
    - Disk, ethernet, etc
    - Performance is poor due to constrained device API
    - Emulated hardware, x86 ISA, inb/outb, PICs
  - Already modifying guest kernel, why not provide virtual device drivers?
    - Faster API?
  - Hypercall interface:
    - syscall:kernel :: hypercall:hypervisor

VMware vs. Paravirtualization

Kernel's device communication with VMware (emulated):

```
void nic_write_buffer(char *buf, int size)
{
    for (; size > 0; size--) {
        nic_poll_ready();
        outb(NIC_TX_BUF, *buf++);
    }
}
```

Kernel's device communication with hypervisor (hypercall):

```
void nic_write_buffer(char *buf, int size)
{
    vmm_write(NIC_TX_BUF, buf, size);
}
```

Xen (2003)

- Popular hypervisor supporting paravirtualization
- Hypervisor runs on hardware
- Runs two kinds of kernels
- Host kernel runs in domain 0 (dom0)
  - Required by Xen to boot
  - Hypervisor contains no peripheral device drivers
  - dom0 needed to communicate with devices
  - Supports all peripherals that Linux or NetBSD do!
- Guest kernels run in unprivileged domains (domUs)

Xen (2003)

- Provides virtual devices to guest kernels
  - Virtual block device, virtual ethernet device
  - Devices communicate with hypercalls & ring buffers
  - Can also assign PCI devices to specific domUs
    - Video card
- Also supports hardware assisted virtualization (HVM)
  - Allows Xen to run unmodified domUs
  - Useful for bootstrapping
  - Also used for “the OS” that can’t be source modified
- Supports Linux & NetBSD as dom0 kernels
- Linux, FreeBSD, NetBSD, and Solaris as domUs

Outline

chroot

- Runs a Unix process with a different root directory
  - Almost like having a separate filesystem
- Share the same kernel & non-filesystem “things”
  - Networking, process control
- Only a minimal sandbox
- Can be escaped! (chroot+fchdir)

User-mode Linux

- Runs a guest Linux kernel as a user space process under a regular Linux kernel
- Requires highly modified Linux kernel
  - No modification to application code
- Used to be popular among hosting providers
- More mature than Xen, but much slower

Container-based OS Virtualization

- Allows multiple instances of an OS to run in isolated containers under the same kernel
- VServer, FBSD Jails, OpenVZ, Solaris Containers
- Aims for VM-like isolation with higher efficiency
  - Hypervisor isolates at the physical resource level
  - Container isolates at the logical resource level
  - Global ids live in separate namespaces for each VM
    - pids, uids, etc
- Total isolation between container userlands
- Kernel resources are well partitioned
  - Makes kernel version migration feasible (VServer)

Outline

Full System Simulation (Simics 1998)

- Software simulates hardware components that make up a target machine
- Interpreter executes each instruction & updates the software representation of the hardware state
- Approach is very accurate but very slow
- Great for OS development & debugging
  - Break on triple fault is better than a reset

System Emulation (Bochs, DOSBox, QEMU)

- Seeks to emulate just enough of system hardware components to create an accurate “user experience”
- Typically CPU & memory subsystems are emulated
  - Buses are not
  - Devices communicate with CPU & memory directly
- Many shortcuts taken to achieve better performance
  - Reduces overall system accuracy
  - Code designed to run correctly on real hardware executes “pretty well”
  - Code not designed to run correctly on real hardware exhibits wildly divergent behavior

System Emulation Techniques

- Pure interpretation:
  - Interpret each guest instruction as they execute
  - Perform a semantically equivalent operation on host
- Static translation:
  - Translate each guest instruction to host once
  - Happens at startup
  - Limited applicability, no self-modifying code

System Emulation Techniques

- Dynamic translation:
  - Translate a block of guest instructions to host instructions just prior to execution of that block
  - Cache translated blocks for better performance
- Dynamic recompilation & adaptive optimization:
  - Discover what algorithm the guest code implements
  - Substitute with an optimized version on the host
  - Hard

QEMU (2005)

- Open source fast processor/machine emulator
- Run an i386, amd64, arm, sparc, powerpc, or mips OS on your i386, amd64, powerpc, alpha, sparc, arm, or s390 computer
- Can run any i386 (or other) OS as a user application
  - Complete with graphics, sound, and network support
  - Don't even need to be root!
- Tolerable performance for real world OSes
  - Orders of magnitude faster than Simics

QEMU's Portable Dynamic Translator

- Cute hack: uses GCC to pregenerate translated code
- Code executing on host is generated by GCC
  - Not hand written
- Makes QEMU easily portable to architectures that GCC supports
  - "The overall porting complexity of QEMU is estimated to be the same as the one of a dynamic linker."

QEMU's Portable Dynamic Translator

Instructions for a given architecture are divided into micro-operations. For example:

```
addl $42, %eax # eax += 42
```

divides into:

```
movl_T0_EAX # T0   = eax
addl_T0_im  # T0  += 42
movl_EAX_T0 # eax  = T0
```

QEMU's Portable Dynamic Translator

- At (QEMU) compile time, each micro-op is compiled from C into an object file for the host architecture
  - *dyngen* copies the machine code from object files
  - Object code used as input data for code generator
- At runtime, code generator reads a stream of micro-ops and emits a stream of machine code
  - By convention, code executes properly as emitted

QEMU's Portable Dynamic Translator

Micro-operations are coded as individual C functions:

```
void OPPROTO op_movl_T0_EAX(void) { T0  = EAX }
void OPPROTO op_addl_T0_im(void)  { T0 += PARAM1 }
void OPPROTO op_movl_EAX_T0(void) { EAX = T0 }
```

which are compiled by GCC to machine code:

```
op_movl_T0_EAX:
    movl    0(%ebp), %ebx
    ret

op_addl_T0_im:
    addl    $42, %ebx
    ret

op_movl_EAX_T0:
    movl    %ebx, 0(%ebp)
    ret
```

QEMU's Portable Dynamic Translator

dyngen strips away function prologue and epilogue:

```
op_movl_T0_EAX:
    movl    0(%ebp), %ebx

op_addl_T0_im:
    addl    $42, %ebx

op_movl_EAX_T0:
    movl    %ebx, 0(%ebp)
```

QEMU's Portable Dynamic Translator

At runtime, QEMU translate the instruction:

```
addl $42, %eax
```

into the micro-op sequence:

```
op_movl_T0_EAX
op_addl_T0_im
op_movl_EAX_T0
```

and then into machine code:

```
movl    0(%ebp), %ebx
addl    $42, %ebx
movl    %ebx, 0(%ebp)
```

QEMU's Portable Dynamic Translator

- When QEMU encounters untranslated code, it translates each instruction until the next branch
  - Forms a single *translation block*
- After each code block is executed, the next block is located in the block hash table
  - Indexed by CPU state
  - Or, block is translated if not found
- Write protects guest code pages after translation
  - Write attempt indicates self modifying code
  - Translations are invalidated on write attempt

Outline

Summary

- Virtualization is big in enterprise hosting
- {Full, hardware assisted, para-}virtualization
- Containers: VM-like abstraction with high efficiency
- Emulation is a slower alternative, more flexibility

Further Reading

Gerald J. Popek and Robert P. Goldberg.

Formal requirements for virtualizable third generation architectures.
Communications of the ACM, 17(7):412–421, July 1974.

John Scott Robin and Cynthia E. Irvine.

Analysis of the intel pentium's ability to support a secure virtual machine monitor.
In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig.

Intel Virtualization Technology: Hardware support for efficient processor virtualization.
Intel Technology Journal, 10(3):167–177, August 2006.

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 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, October 2003.

Yaozu Dong, Shaofan Li, Asit Mallick, Jun Nakajima, Kun Tian, Xuefei Xu, Fred Yang, and Wilfred Yu.

Extending Xen with Intel Virtualization Technology.
Intel Technology Journal, 10(3):193–203, August 2006.

Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson.

Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors.
In *Proceedings of the 2007 EuroSys conference*, Lisbon, Portugal, March 2007.

Fabrice Bellard.

QEMU, a fast and portable dynamic translator.
In *Proceedings of the 2005 USENIX Annual Technical Conference*, Anaheim, CA, April 2005.