Virtual Machines

questions/logistics

office hours posted on website

VM setup due Friday

virtual machines

illusion of a dedicated machine

could or could not behave like real machine

virtual machine types

language — designed for programming language

process — designed for shared system

system — designed to emulate “real” hardware

virtual machine types

language — designed for programming language

process — designed for shared system

system — designed to emulate “real” hardware

language VMs

programming languages have a ‘virtual machine’

e.g. the Java virtual machine

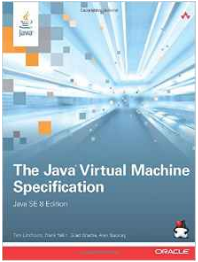
compiler targets virtual machine

virtual machine designed for language

easier than real machine to compile to

reasonably fast to simulate on real machine

JVM specializations



"assembly" of virtual machine
knows about objects, methods

ISA designed for Java programs
with some adaptations for other languages

all stack-based instructions (no registers)
(thought to be) easier to implement in software

safe: can't leak memory; can't segfault

virtual machine types

language — designed for programming language

process — designed for shared system

system — designed to emulate “real” hardware

OSs are virtual machines

process virtual machines

different interface than physical HW

system calls instead of I/O instructions

system calls/signals instead of interrupts

process versus system

more complicated:

- files
- network connections
- communicating with other processes
- ...

but simpler to program

- more flexible
- no hardware details (disk sizes, etc.)

virtual machine types

language — designed for programming language

process — designed for shared system

system — designed to emulate “real” hardware

system virtual machines

acts (more) like real hardware

not files, but a hard drive

not network connections, but an ethernet device

not memory allocation calls, but page tables

…

system virtual machines run operating systems

modern system VM software

VMWare — 1998 startup

VirtualBox (open source; Oracle, formally Sun)

Parallels (targets OS X)

Xen

QEMU

Hyper-V (Microsoft)

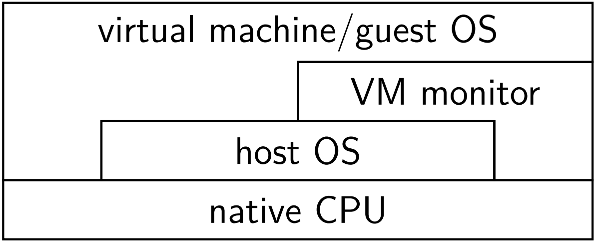
hosts and guests

guest OS — what's inside the virtual machine

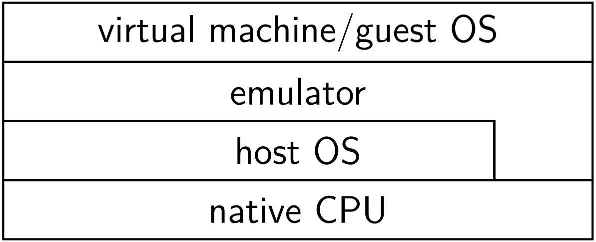
host OS — what's outside the virtual machine

VM implementation strategies

traditional VM

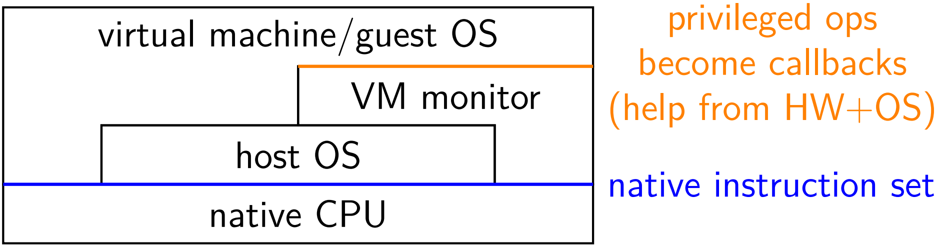


emulator



VM implementation strategies

traditional VM



virtual machine/guest OS

VM monitor

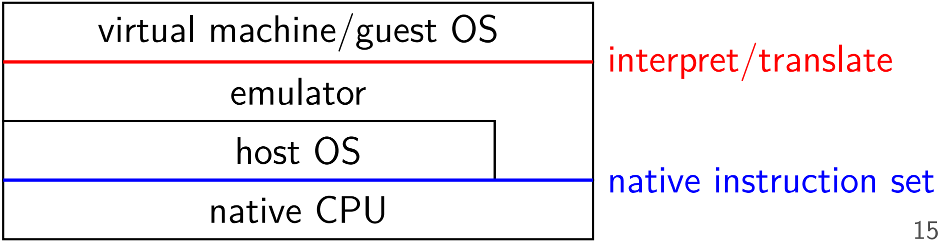
host OS

native CPU

privileged ops
become callbacks
(help from HW+OS)

native instruction set

emulator



virtual machine/guest OS

emulator

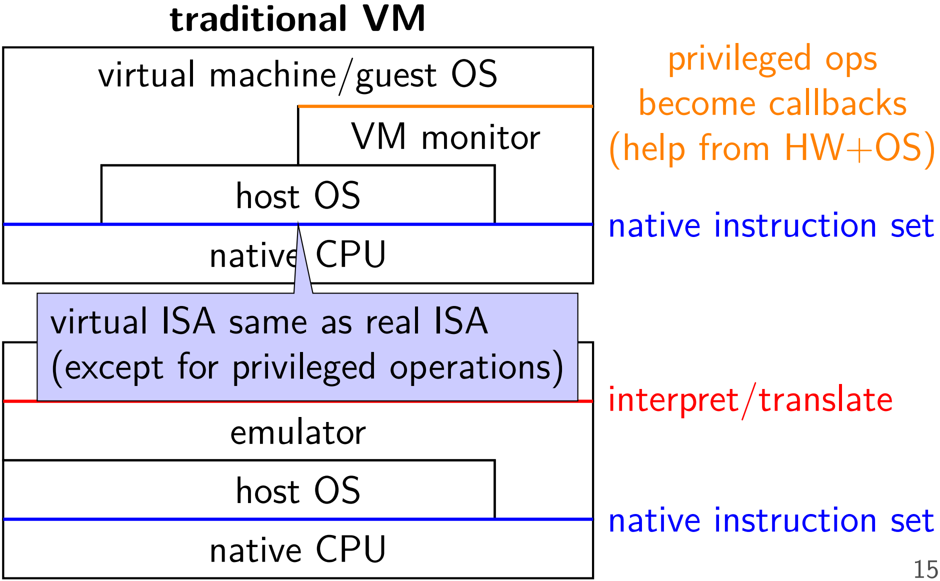
host OS

native CPU

interpret/translate

native instruction set

VM implementation strategies



traditional VM

virtual machine/guest OS

VM monitor

host OS

native CPU

privileged ops become callbacks (help from HW+OS)

native instruction set

virtual ISA same as real ISA (except for privileged operations)

emulator

host OS

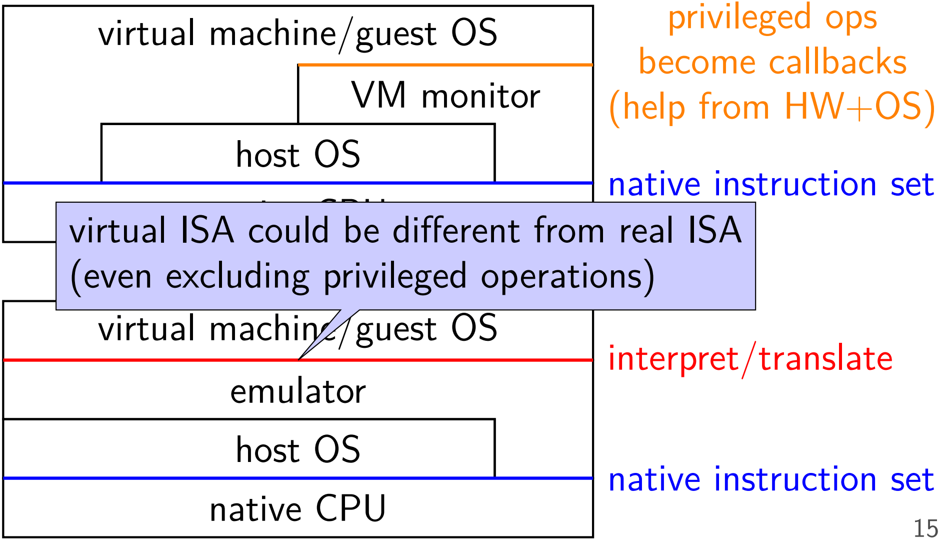
native CPU

interpret/translate

native instruction set

VM implementation strategies

traditional VM



virtual machine/guest OS

VM monitor

host OS

privileged ops become callbacks (help from HW+OS)

native instruction set

virtual ISA could be different from real ISA (even excluding privileged operations)

virtual machine/guest OS

emulator

host OS

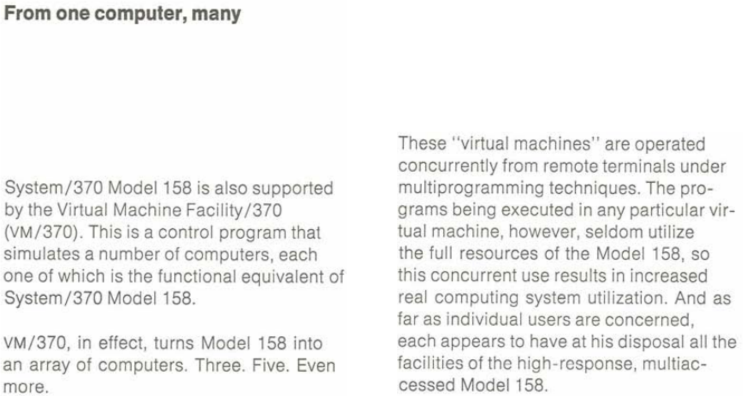
native CPU

interpret/translate

native instruction set

VMs are old

IBM/370 Model 158 (announced 1972) marketing:



From one computer, many

System/370 Model 158 is also supported by the Virtual Machine Facility/370 (VM/370). This is a control program that simulates a number of computers, each one of which is the functional equivalent of System/370 Model 158.

VM/370, in effect, turns Model 158 into an array of computers. Three. Five. Even more.

These "virtual machines" are operated concurrently from remote terminals under multiprogramming techniques. The programs being executed in any particular virtual machine, however, seldom utilize the full resources of the Model 158, so this concurrent use results in increased real computing system utilization. And as far as individual users are concerned, each appears to have at his disposal all the facilities of the high-response, multiaccessed Model 158.

Excerpt from: Computer History Museum catalog number 102646258
`http://www.computerhistory.org/collections/catalog/102646258`

VMs as consolidation

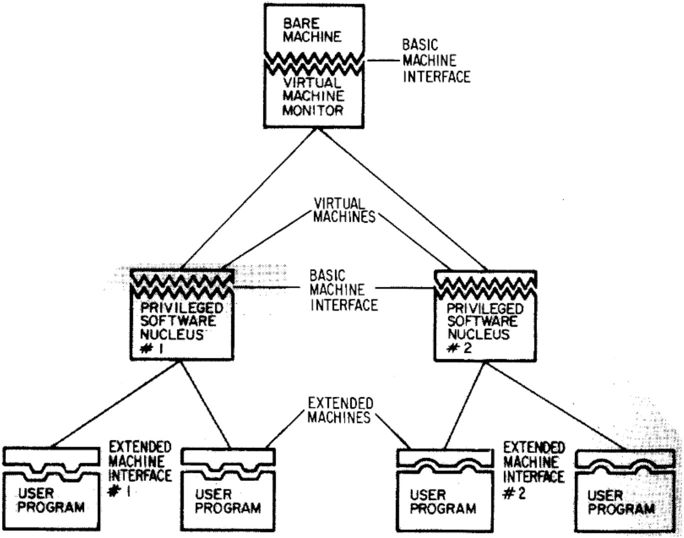


Figure: Goldberg, "Survey of Virtual Machine Research", IEEE Computer, September 1974

the consolidation case

compatibility — customize “whole” machine

efficiency —

- two+ CPUs/hard drives for the work/data of one?
- two+ CPUs for the work/data of one?

2011 public ‘cloud’ server CPU utilization: <10%

- after consolidation

utilization %s: Liu, “A Measurement Study of Server Utilization in Public Clouds”

VM death and resurgence

VMs started with mainframes

one computer for an entire company

…

then the personal computer happened

resurgence of VMs

consolidation again (still a good idea)

compatibility
- Windows on Mac
- Unix on Windows
- Windows 98 on Windows NT, etc.

…

1998 startup: VMWare
- bought by EMC which was bought by Dell

VM implementation

hardware support —

- originally, only viable way
- IBM/370, VirtualBox, modern VMware, etc.

binary translation —

- historic VMware

paravirtualization — Xen

emulation — Bochs

on kernel mode

hardware has two modes:

- user mode and kernel mode

typically, only OS can run in kernel mode

privileged operations require kernel mode

exceptions and VMs

privileged operations need to run in kernel mode

guest OS is run in user mode

guest OS tries to do a privileged operation?

  exception gives control to host OS

I/O device (e.g. keyboard) tries to signal OS

  exception gives control to host OS

exception handlers are part of virtual machine monitor

VMs and kernel mode

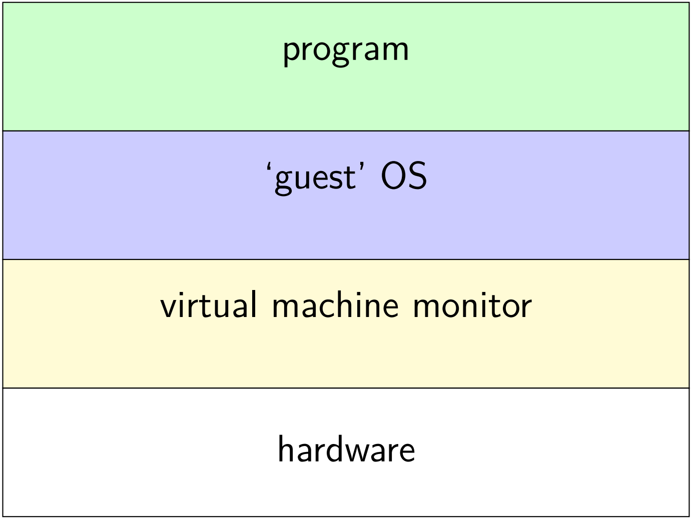
basic idea: run guest OS in user mode

virtual machine monitor (VMM) runs in kernel mode

on exception: virtual machine monitor forwards to guest OS

“mirrors” what hardware did for VMM

system call flow



conceptual layering

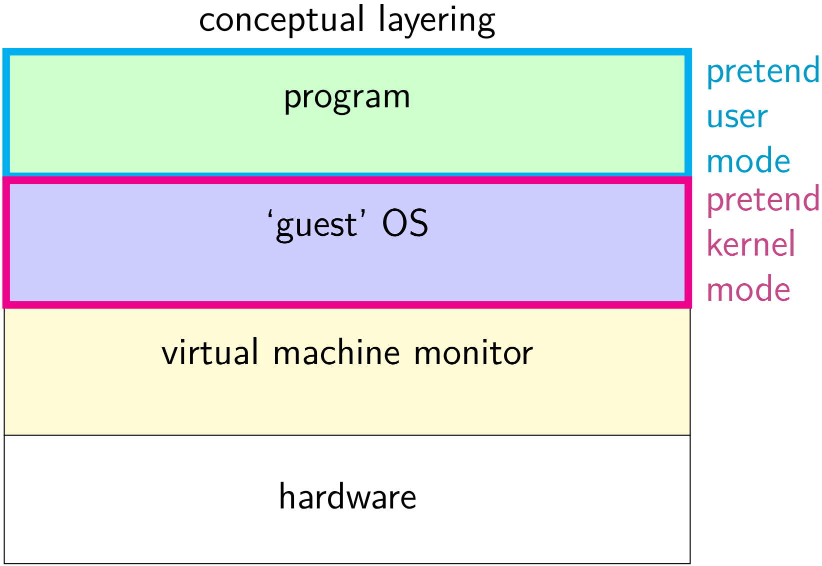
program

'guest' OS

virtual machine monitor

hardware

system call flow



conceptual layering

program

'guest' OS

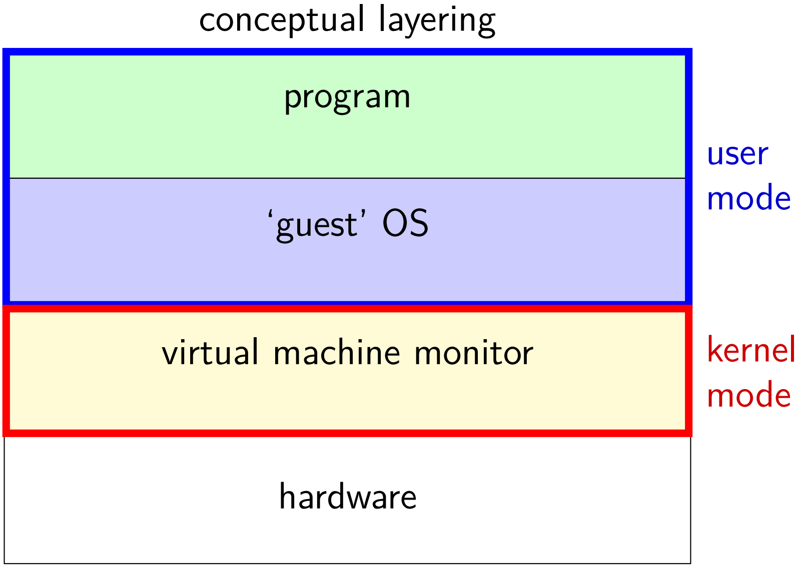
virtual machine monitor

hardware

pretend user mode

pretend kernel mode

system call flow



conceptual layering

program

'guest' OS

virtual machine monitor

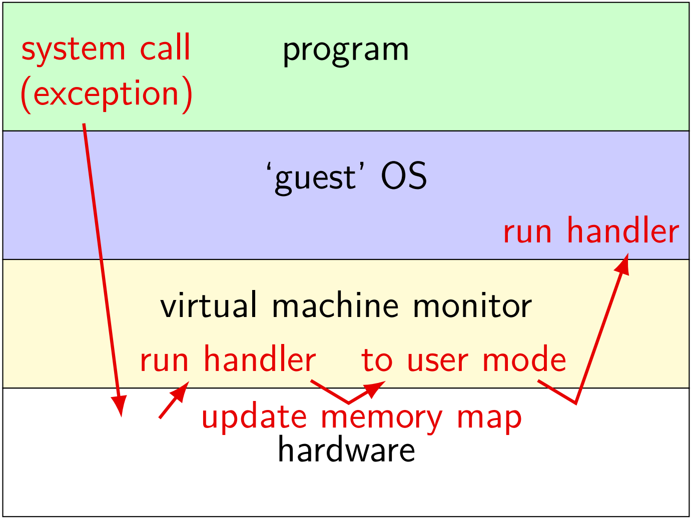
hardware

user mode

kernel mode

system call flow

conceptual layering



program

system call (exception)

'guest' OS

run handler

virtual machine monitor

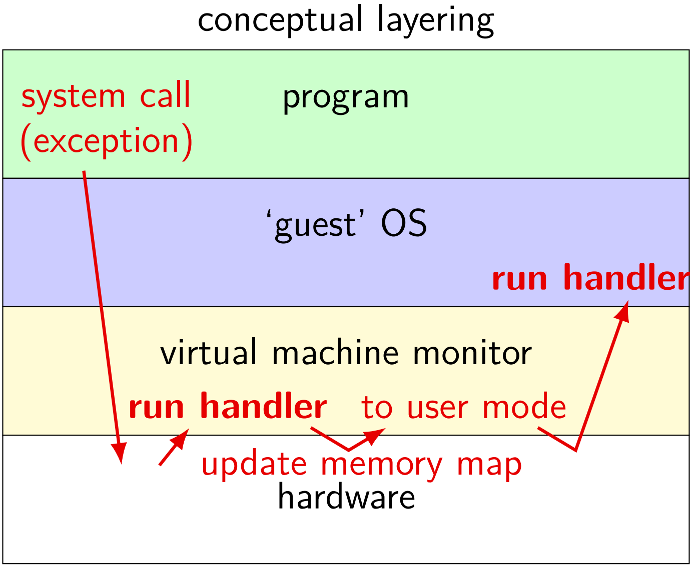
run handler

to user mode

update memory map

hardware

system call flow



conceptual layering

system call (exception)

program

'guest' OS

run handler

virtual machine monitor

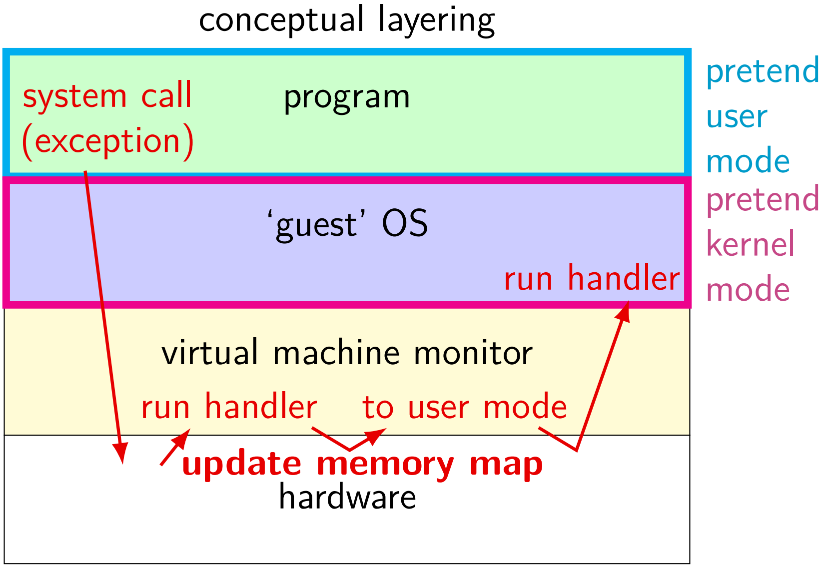
run handler

to user mode

update memory map

hardware

system call flow



extra hardware support

privileged operations becoming exceptions: minimal

hardware can do more:

nested page table lookup

- makes memory mapping changes much faster/simpler

handling of read-only privileged instructions

- e.g. reading “interrupt enable” flag

forwarding of some exceptions

- e.g. flag to make syscalls run guest OS

binary translation

compile assembly to new assembly

works without instruction set support

early versions of VMWare on x86 (before x86 added virtualisation support)

can be used to run one platform on another

binary translation idea

```
0x40FE00: addq %rax, %rbx
movq 14(%r14,4), %rdx
addss %xmm0, (%rdx)
...
0x40FE3A: jne 0x40F404
subss %xmm0, 4(%rdx)
...
je 0x40F543
ret
```

binary translation idea

```
0x40FE00: addq %rax, %rbx
movq 14(%r14,4), %rdx
addss %xmm0, (%rdx)
...
0x40FE3A: jne 0x40F404
```

```
subss %xmm0, 4(%rdx)
...
je 0x40F543
```

```
ret
```

divide machine code into *basic blocks*
(= “straight-line” code)
(= code till jump/call/etc.)

binary translation idea

```
0x40FE00: addq %rax, %rbx
movq 14(%r14,4), %rdx
addss %xmm0, (%rdx)
...
0x40FE3A: jne 0x40F404
subss %xmm0, 4(%rdx)
...
je 0x40F543
ret
```

generated code:

```
// addq %rax, %rbx
movq rax_location, %rdi
movq rbx_location, %rsi
call checked_addq
movq %rax, rax_location
...
// jne 0x40F404
... // get CCs
je do_jne
movq $0x40FE3F, %rdi
jmp translate_and_run
do_jne:
movq $0x40F404, %rdi
jmp translate_and_run
```

a binary translation idea

convert whole *basic blocks*

- code upto branch/jump/call

end with call to `translate_and_run`

- compute new simulated PC address to pass to call

making binary translation fast

cache converted code

- `translate_and_run` checks cache first

patch calls to `translate_and_run` to refer directly to cached code

do something more clever than
`movq rax_location, ...`

- map (some) registers to registers, not memory

ends up being “just-in-time” compiler

binary translation? really?

early VMWare: focused on little pieces of OS code that couldn't be emulated

- a few instructions that behaved differently to fix

used by Apple to handle changing CPU designs

- not a system VM — used the native OS mostly

Rosetta: run Power PC on Intel (2005–2011)

Mac 68k emulator: Run Motorola 680x0 on Power PC (1994–2005)

why binary translation?: POPF

x86 has an instruction called POPF

pop flags from stack

- condition codes — `CF`, `ZF`, `PF`, `SF`, `OF`, etc.
- direction flag (`DF`) — used by string instructions
- I/O privilege level (`IOPL`)
- interrupt enable flag (`IF`)
- …

why binary translation?: POPF

x86 has an instruction called POPF

pop flags from stack
  - condition codes — `CF`, `ZF`, `PF`, `SF`, `OF`, etc.
  - direction flag (`DF`) — used by string instructions
  - **I/O privilege level** (`IOPL`)
  - **interrupt enable flag** (`IF`)
  - …

some flags are privileged!

popf **silently** doesn't change them in user mode

more binary translation problems

PUSHF also bad — want to pretend interrupts are disabled, e.g.

several more x86 instructions

processor extensions to change these to be virtualizable

mechanism: flag to make them trigger interrupt to virtual machine monitor

other binary translation utility

enables other software analysis on unmodified binaries

example: valgrind, debugging tools:

- memory errors
- synchronization bugs
- …

paravirtualization

only a few pieces of the OS use things like POPF

instead: modify OS

called paravirtualization

OS makes explicit calls to virtual machine monitor

very small OS patch

more efficient?

other virtualisation support nits

hardware support for nested page tables

alternatives work, but are complex/slower

hardware support for limiting I/O devices

- can safely give VMs I/O device access

emulation

read instruction, do what it says, repeat

slowest technique, but easiest to implement

easiest to provide detailed debugging information for

Why do we care about VMs?

isolation

run dangerous stuff safely!

analyze dangerous stuff without disrupting it!

isolation: network

virtual machines have a “virtual” network device

easy to make disconnected

provide network of other VMs, not connected to internet

setup custom firewall without extra hardware

isolation: disk

virtual machines have “virtual” hard drives — just a file!

virus infects files? not anything that matters on the machine

easy to identify what to backup

- even if virus modifies “hidden” files

…

snapshots

virtual disks, virtual memory, ...

make copy of disk/memory/etc.

e.g. see what damage malware does

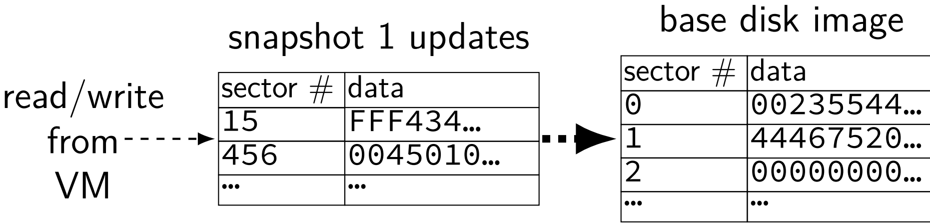
go back to before damage happens

snapshot efficiency

but aren't snapshots slow???

- copy all of disk, memory

can be done faster:



read/write from VM

snapshot 1 updates

sector #	data
15	FFF434…
456	0045010…
…	…

base disk image

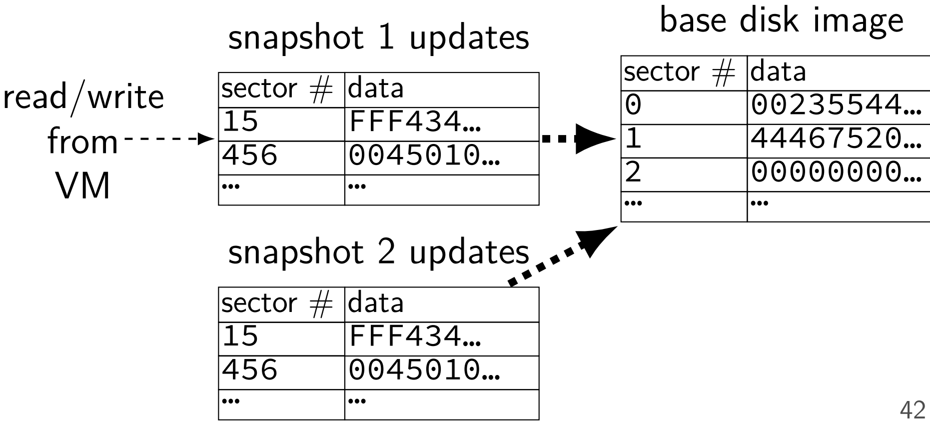
sector #	data
0	00235544…
1	44467520…
2	00000000…
…	…

snapshot efficiency

but aren't snapshots slow???

- copy all of disk, memory

can be done faster:



read/write from VM

snapshot 1 updates

sector #	data
15	FFF434…
456	0045010…
…	…

snapshot 2 updates

sector #	data
15	FFF434…
456	0045010…
…	…

base disk image

sector #	data
0	00235544…
1	44467520…
2	00000000…
…	…

debugging support

hardware has support for debuggers…

but there are ways of interfering/detecting

virtual machines can "hide" these changes

- e.g. slow down in debugger? — virtual clock

(might require slower implementation technique)

also easy to do whole-machine debugging on VMs

- attach GDB to entire VM

VM replay

virtual machines can support replay

rerun something exactly the same

good for debugging

not trivial to implement — why?

VM replay challenges

timing and I/O

VM replay challenges

timing and I/O

need to remember exactly when I/O happens

need to have virtual clock

how?

VM replay challenges

timing and I/O

need to remember exactly when I/O happens

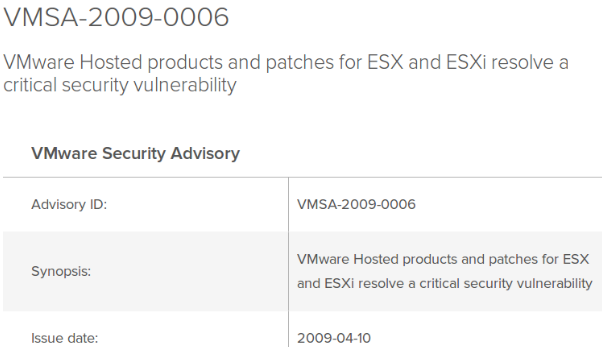
need to have virtual clock

how?

- log all I/O, timer readings
- read log on replay

```
at instruction 100043243: keypress 'a'
at instruction 100483782: time = 100333.3456
at instruction 100688445: network packet '024A...
```

virtual machine escape

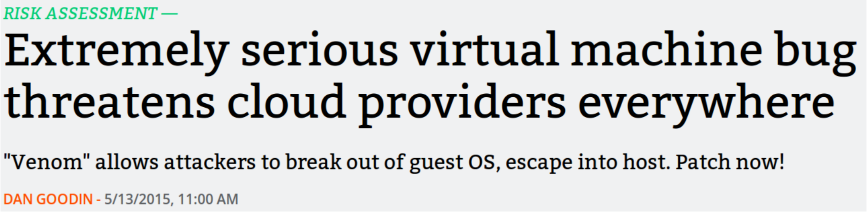


VMSA-2009-0006

VMware Hosted products and patches for ESX and ESXi resolve a critical security vulnerability

VMware Security Advisory

Advisory ID:	VMSA-2009-0006
Synopsis:	VMware Hosted products and patches for ESX and ESXi resolve a critical security vulnerability
Issue date:	2009-04-10



RISK ASSESSMENT —

Extremely serious virtual machine bug threatens cloud providers everywhere

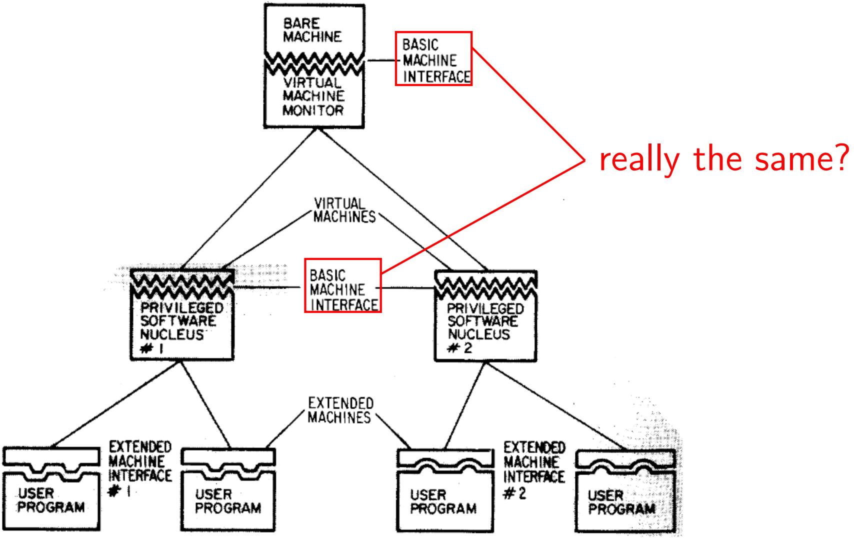
"Venom" allows attackers to break out of guest OS, escape into host. Patch now!

DAN GOODIN - 5/13/2015, 11:00 AM

virtual machine escape

bug in virtual machine monitor that lets virtual machines run code that's not isolated

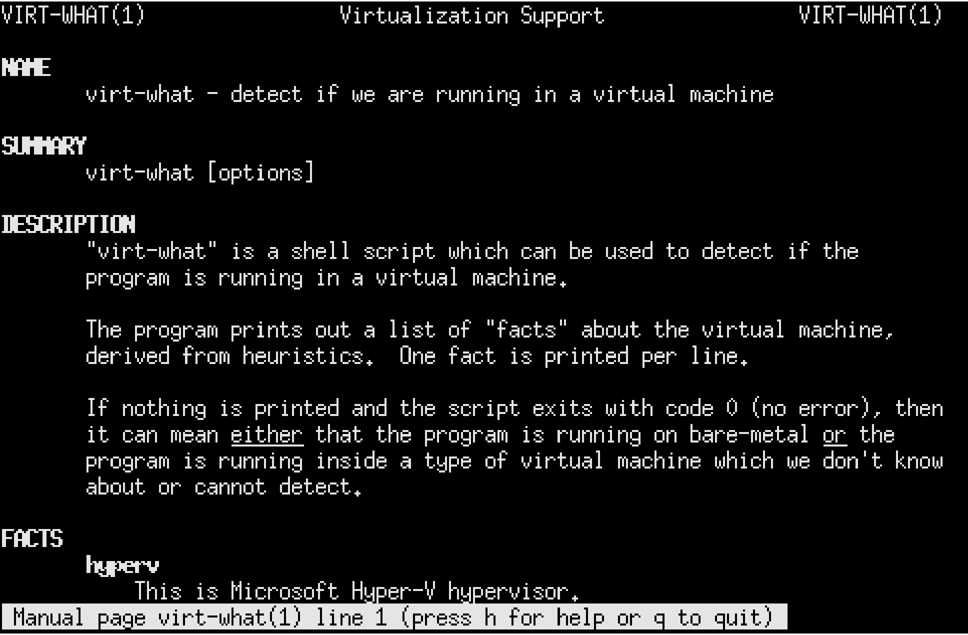
VM detection



VM detection

no reason why detectable, but…

normal system VMs are not not stealthy



```
VIRT-WHAT(1)            Virtualization Support            VIRT-WHAT(1)

NAME
       virt-what - detect if we are running in a virtual machine

SUMMARY
       virt-what [options]

DESCRIPTION
       "virt-what" is a shell script which can be used to detect if the
       program is running in a virtual machine.

       The program prints out a list of "facts" about the virtual machine,
       derived from heuristics.  One fact is printed per line.

       If nothing is printed and the script exits with code 0 (no error), then
       it can mean either that the program is running on bare-metal or the
       program is running inside a type of virtual machine which we don't know
       about or cannot detect.

FACTS
       hyperv
           This is Microsoft Hyper-V hypervisor.
 Manual page virt-what(1) line 1 (press h for help or q to quit)
```

without specialized tools

```
ubuntu@ubuntu-xenial:~$ sudo dmidecode | head
# dmidecode 3.0
Getting SMBIOS data from sysfs.
SMBIOS 2.5 present.
10 structures occupying 450 bytes.
Table at 0x000E1000.

Handle 0x0000, DMI type 0, 20 bytes
BIOS Information
        Vendor: innotek Gmbcp
        Version: VirtualBox
```

DMI — BIOS (system startup) table

VM detection: case study

search for devices with “VMWARE” in their names

search for VM-only device drivers

check if processor is suspiciously slow
  - ideally things that are easier in HW than SW
  - e.g. speed of syscalls, address space changes
  - unimplemented features?
  - might need external source of time

s://www.fireeye.com/blog/threat-research/2011/01/the-dead-giveaways-of-vm-aware-malware.html

VMs for anti-malware

does SW do something bad?

run it in a VM/“sandbox”

check if things change that shouldn’t

actual antivirus software technique

VMs as antimalware limitations

completeness

- emulate entire filesystem?
- emulate all system calls?
- emulate network?
- provide real network?

user input, etc.

- can't easily automate keypresses, etc.

speed

- how long until you say "it's safe"

lightweight sandboxing

(system) VMs are resource-intensive

two OSes — lots of extra memory

worse performance

- more code needed for I/O

more efficient alternative: operating system isolation

- e.g. on lab machines, users can't interfere with each other
- e.g. browsers do this for web page code

OS interface size

OS interfaces are complicated

Linux:

- 100s of system calls
- ... including some to talk to hundreds of device drivers

hard to tell which program needs

hard to tell which are safe

OS sandboxing support

OS-level isolation of filesystem, memory, CPU

extra code for each kind of resource/system call

lots of obscure system resources to exhaust, etc.:

- list of pending signals
- network buffers
- buffers for interprocess pipes
- process control data structures in the OS
- etc.

need to limit each of them

sandboxing on Linux (1)

one mechanism: secccomp

system call filter

example: video decoder:

- reads encoded video
- writes decoded images

only needs read/write — easy to sandbox

sandboxing on Linux (2)

another mechanism: cgroups

set limits for CPU, memory, networks, process IDs, etc.

extra kernel code for each kind of resource

only expose subset of filesystem (`chroot`)

/ (root directory) changedA

much more complex to configure securely than VM

not used by major rental computing providers

the real sandboxing problem

policy

VMs in this course

consistent environment!

our attacks may depend on exact memory addresses

our attacks may depend on exact versions of system libraries

do real attackers do that?

if exploits are so sensitive...

fragile, not always broken

exploits can be made less fragile

Slapper worm: exploit variants for 23 architectures

exploits: avoiding fragility

some exploits cause a jump to attacker-controlled code

fragile because need to encode exact address

partial fix: choose exploit code to give leeway

nop sled

```
nop  /* ← jumping to here */
nop
nop
nop
nop
nop  /* ← same as jumping to here */
nop
nop
...
/* exploit code here */
```

next topic: x86-64 assembly

you've seen this before

in theory

x86-64 assembly

history: AMD constructed 64-bit extension to x86 first

- marketing term: AMD64

Intel first tried a new ISA (Itanium), which failed

Then Intel copied AMD64

- marketing term: EM64T (Extended Memory 64 Technology)
- later marketing term: Intel 64

both Intel and AMD have manuals — definitive reference

Intel® 64 and IA-32 Architectures Software Developer's Manual

Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D

x86-64 manuals

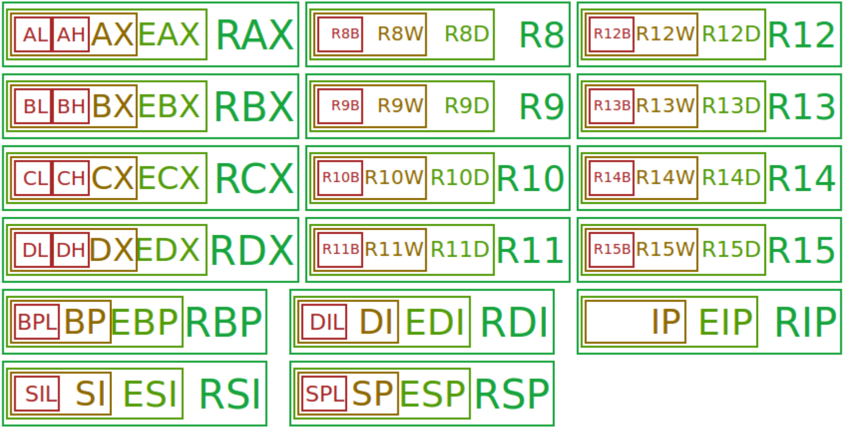
Intel manuals:

`https://software.intel.com/en-us/articles/intel-sdm`
24 MB, 4684 pages
Volume 2: instruction set reference (2190 pages)

AMD manuals:

`https://support.amd.com/en-us/search/tech-docs`
"AMD64 Architecture Programmer's Manual"

recall: x86-64 general purpose registers



Immae via Wikipedia

overlapping registers (1)

setting 32-bit registers sets whole 64-bit register

extra bits are always zeroes

```
movq $0x123456789abcdef, %rax
xor %eax, %eax
// %rax is 0, not 0x1234567800000000
movl $−1, %ebx
// %rbx is 0xFFFFFFFF, not −1 (0xFFFFFFFFFFFFFFFF
```

overlapping registers (2)

setting 8/16-bit registers doesn't change rest of 64-bit register:

```
movq $0x12345789abcdef, %rax
movw $0xaaaa, %ax
// %rax is 0x123456789abaaaa
```

AT&T versus Intel syntax

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

AT&T syntax (1)

`movq $42, 100(%rbx,%rcx,4)`

destination last

constants start with $

registers start with %

AT&T syntax (2)

`movq $42, 100(%rbx,%rcx,4)`

operand length: `q`

    `l` = 4; `w` = 2; `b` = 1

`100(%rbx,%rcx,4)`:
`memory[100 + rbx + rcx * 4]`

`sub %rax, %rbx`: `rbx` ← `rbx - rax`

Intel syntax

destination first

`[...]` indicates location in memory

`QWORD PTR [...]` for 8 bytes in memory

- DWORD for 4
- WORD for 2
- BYTE for 1

On LEA

LEA = Load Effective Address

uses the syntax of a memory access, but…

just computes the address and uses it:

- `leaq 4(%rax), %rax` has same result as `addq $4, %rax`
  - almost — doesn't set condition codes
- `leaq (%rax,%rax,4), %rax` multiplies `%rax` by 5
  - `address-of(memory[rax + rax * 4])`

question

```
.data
string:
    .asciz "abcdefgh"
.text
    movq $string, %rax
    movq string, %rdx
    movb (%rax), %bl
    leal 1(%rbx), %ebx
    movb %bl, (%rax)
    movq %rdx, 4(%rax)
```

What is the final value of string?

a. "abcdabcd"

b. "bbcdefgh"