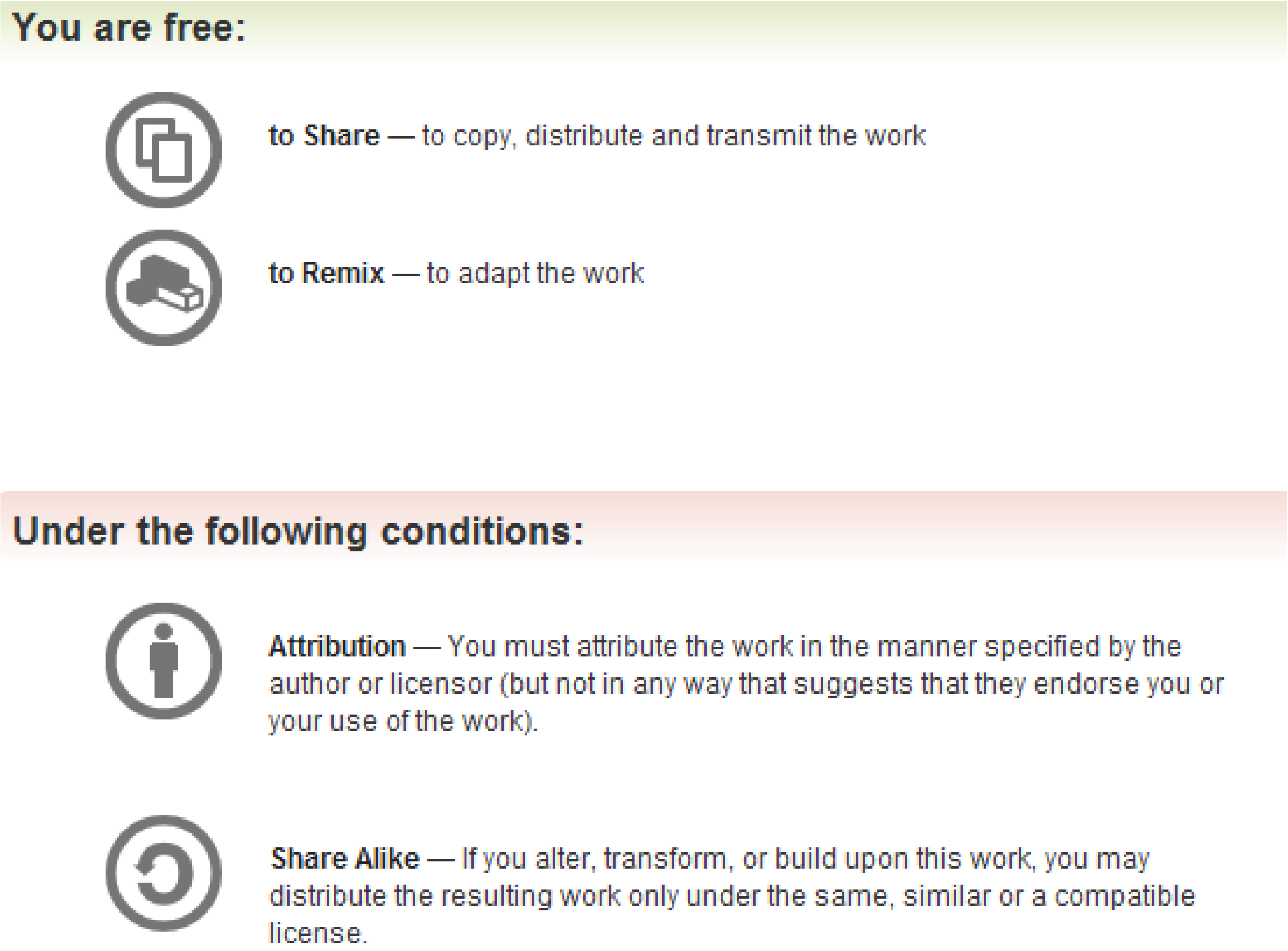
Advanced x86: Virtualization with VT-x Part 1

David Weinstein
dweinst@insitusec.com

All materials are licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Acknowledgements

- <Could be you!> : Tell me something that I didn’t know that ends up in the course material
- Thanks to Xeno Kovah for pushing me to create this material and for reviewing it periodically as it was created.
- Thanks to Corey Kallenberg for device driver signing info for Windows 7

Introductions

- Name
- Department
- Work interests
  - Projects, sponsor, etc.

Prerequisites

- Intro/Intermediate x86 (or equivalent) required
- Rootkits class will probably help

Agenda

- Introduction
- Lightning x86_64 review
- VT-x
- VMM detection
- Relevant hypervisor projects
- Time permitting
    - Discussion: writing “undetectable” bot for SC2/Diablo 3?

Questions

- Stolen from Xeno…
- Questions: Ask ‘em if you got ‘em
  - If you fall behind and get lost and try to tough it out until you understand, it’s more likely that you will stay lost, so ask questions ASAP.
- Browsing the web and/or checking email during class is a great way to get lost ;)
- 2 hours, 10 min break, 2 hours, 1 hour lunch, 2 hours 10 min break, 1.5 hours, done
- Adjusted depending on whether I’m running fast or slow (or whether people are napping after lunch :P)

Scope

- While advanced, still introductory
- Fundamentals, challenges, techniques
- Open source virtualization technologies and implementations
- Primarily Intel® specific discussions, 64 bit host/guests
- All indications to sections in Intel® manual correspond to December 2011 edition (Order Number: 325384-041US) which should be provided with these slides

Goals

- Identify/understand/implement various hypervisor concepts, integrate by parts
- Blue Pill/Hyperjack
  - post-boot (hosted) hypervisor shim technique
- Highly curated tour of Intel Manual with labs

Introduction

- The goal is to get the core virtualization concepts out of the way and clear up the semantics first.
- We'll cover some 64-bit concepts
- Then move into specifics for Intel VT-x;
  - We will be covering the architecture, instructions, and specifics needed to write real code
  - With Windows focus
  - Series of labs to guide the way

Sqr0…

- Each instance of an OS is called a Virtual Machine (VM), guest, or domU.
- Hypervisor ≡ Virtual Machine Monitor (VMM)
- There are fundamentally different approaches to virtualization; important to understand the differences

Terminology Bootstrap

- Virtual Machine Extensions (VMX)
- Virtual Machine Monitor (VMM)
- VMX Root operation
  - VMM, host VM
- Management VM
  - dom0
- VMX Non-root operation
  - domU, guest VM
- Others we'll pick up along the way

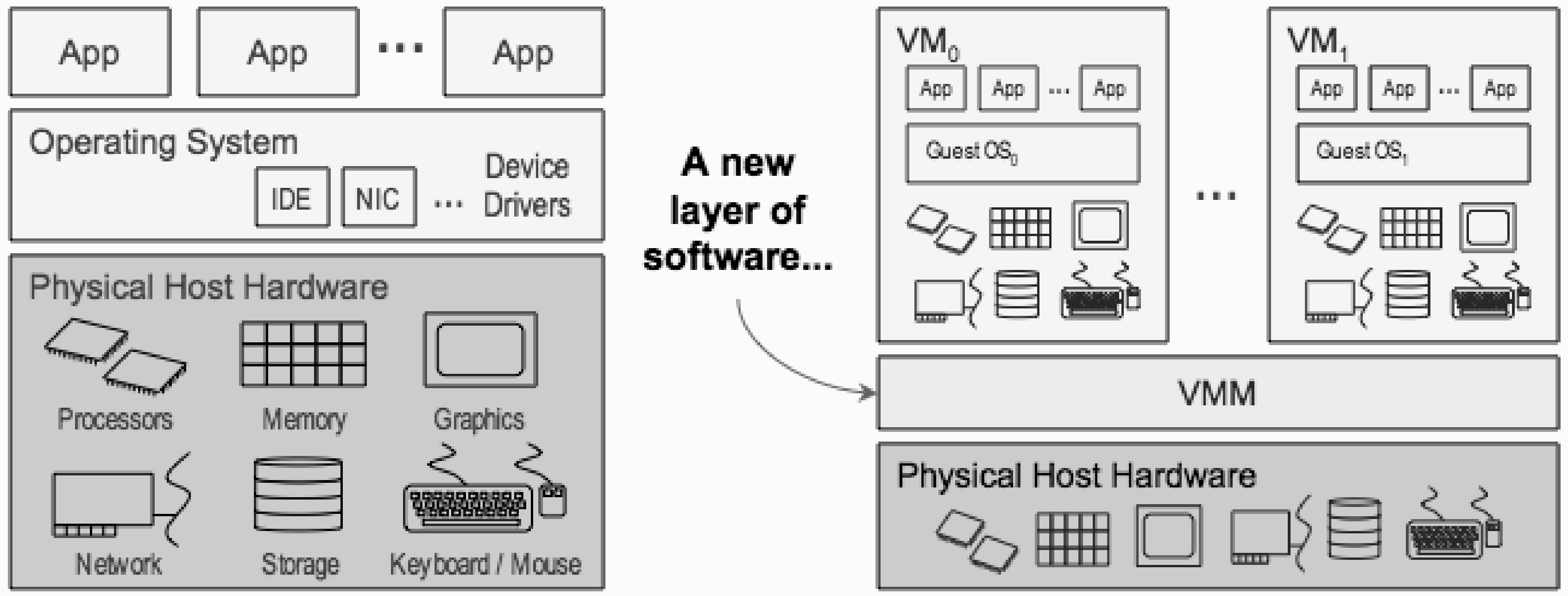
Virtualization is Resource Abstraction yo!

- "The process of hiding the underlying physical hardware in a way that makes it transparently usable and shareable by multiple operating systems." [IBM]
    - Most hardware can be virtualized to the point that a guest doesn't know/care
    - Underlying physical hardware supporting VM may not be dedicated to it

[IBM] http://www.ibm.com/developerworks/linux/library/l-hypervisor/

Abstraction

Grey for effect… 2005 seems so long ago…



Without VMs: Single OS owns all hardware resources

With VMs: Multiple OSes share hardware resources

Ref: August 2005 System Virtual Machines, HotChips 17 Tutorial

Vendor technologies

CPU Flag	Virtual Machine Extensions (VMX)	Secure Virtual Machine (SVM)
Processor emulation	VT-x	AMD-v
Extended page tables	Extended page tables (EPT)	Rapid Virtualization Indexing (RVI)
MMU emulation	VT-d	AMD-Vi
Network emulation	VT-c	
PCI emulation	PCI-SIG I/O Virtualization	PCI-SIG I/O Virtualization

We will be focused on Intel VT-x

VMM Types

Type 1. “bare metal” hypervisors run directly on the host hardware

- guest OS runs at level above the hypervisor

Type 2. “hosted” hypervisors run on top of an OS

- The hypervisor layer exists as distinct second software level
- Guest operating systems run at the third level above the hardware

x86-64 Quick Review

- No more inline assembly (MS compilers) ☆
- New instructions
- New General Purpose (**GP**) registers
- Changes to Segmentation
- Paging, now more fun
- RIP relative addressing
- REX prefixes
- x64 (fastcall) calling convention

Checking The 64-bit feature

- Use CPUID input-value 0x80000001 for obtaining extended feature bits
- Returned in the ECX and EDX registers

```
.data
ext_features  DB 8 DUP(0)          # 8 bytes (zeroed) for extended features bits

.code
mov eax, 80000001h                 # setup input-value in EAX
cpuid                              # then execute CPUID
mov [ext_features+0], edx          # save feature-bits from EDX
mov [ext_features+4], ecx          # save feature-bits from ECX
```

Extended features bits

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ECX =	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	LSF

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EDX =	R	R	IA-3	R	R	R	R	R	R	R	R	XD	R	R	R	R	R	R	R	R	SYSCALL	R	R	R	R	R	R	R	R	R	R	R

Modified from http://cs.usfca.edu/~cruse/cs630f06/lesson05.ppt

IA32e = Intel 64-bit Technology
XD = eXecute Disable paging-bit implemented
SYSCALL = Fast SYSCALL / SYSRET (64-bit mode)
LSF = LAHF / SAHF implemented in 64-bit mode
R = reserved bit

New instructions (x64)

CDQE	Convert doubleword to quadword
CMPSQ	Compare string operands
CMPXCHG16B	Compare RDX:RAX with m128
LODSQ	Load qword at address (R)SI into RAX
MOVSQ	Move qword from address (R)SI to (R)DI
MOVZX (64-bits)	Move doubleword to quadword, zero-extension
STOSQ	Store RAX at address RDI
SWAPGS	Exchanges current GS base register value with value in MSR address C0000102H
SYSCALL	Fast call to privilege level 0 system procedures
SYSRET	Return from fast system call

New GP Registers

- “In 64-bit mode, there are 16 general purpose (GP) registers and the **default operand size is 32 bits**. However, general-purpose registers are able to work with either 32-bit or 64-bit operands.”
- R8-R15 represent **eight** new general-purpose registers. All of these registers can be accessed at the byte (B), word (2 B), dword (4 B), and qword (8 B) level.

Registers

```
typedef struct _GUEST_REGS
{
        ULONG64 rax;
        ULONG64 rcx;
        ULONG64 rdx;
        ULONG64 rbx;
        ULONG64 rsp;
        ULONG64 rbp;
        ULONG64 rsi;
        ULONG64 rdi;
        ULONG64 r8;
        ULONG64 r9;
        ULONG64 r10;
        ULONG64 r11;
        ULONG64 r12;
        ULONG64 r13;
        ULONG64 r14;
        ULONG64 r15;
} GUEST_REGS, *PGUEST_REGS;
```

x86-64 Segmentation

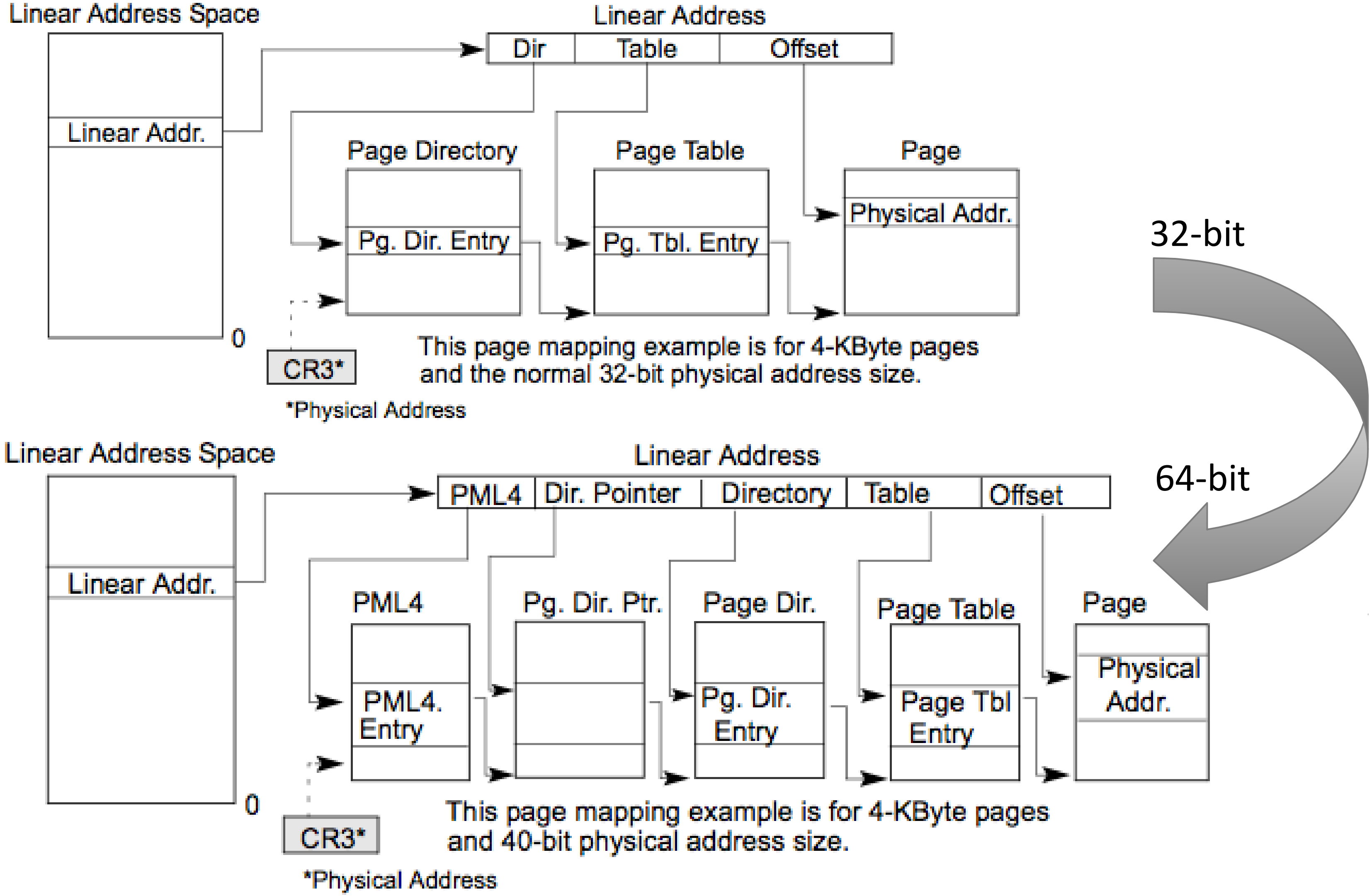
- “Segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space.”
- “Specifically, the processor treats the segment base of CS, DS, ES, and SS as zero in 64-bit mode (this makes a linear address equal an effective address). Segmented and real address modes are not available in 64-bit mode.”

Intel Vol 3 (Section 3.2.4)

x86-64 Segmentation (2)

- "Even though segmentation is generally disabled, segment register loads may cause the processor to perform segment access assists."
- "During these activities, enabled processors will still perform most of the legacy checks on loaded values (even if the checks are not applicable in 64-bit mode)."

Paging structures



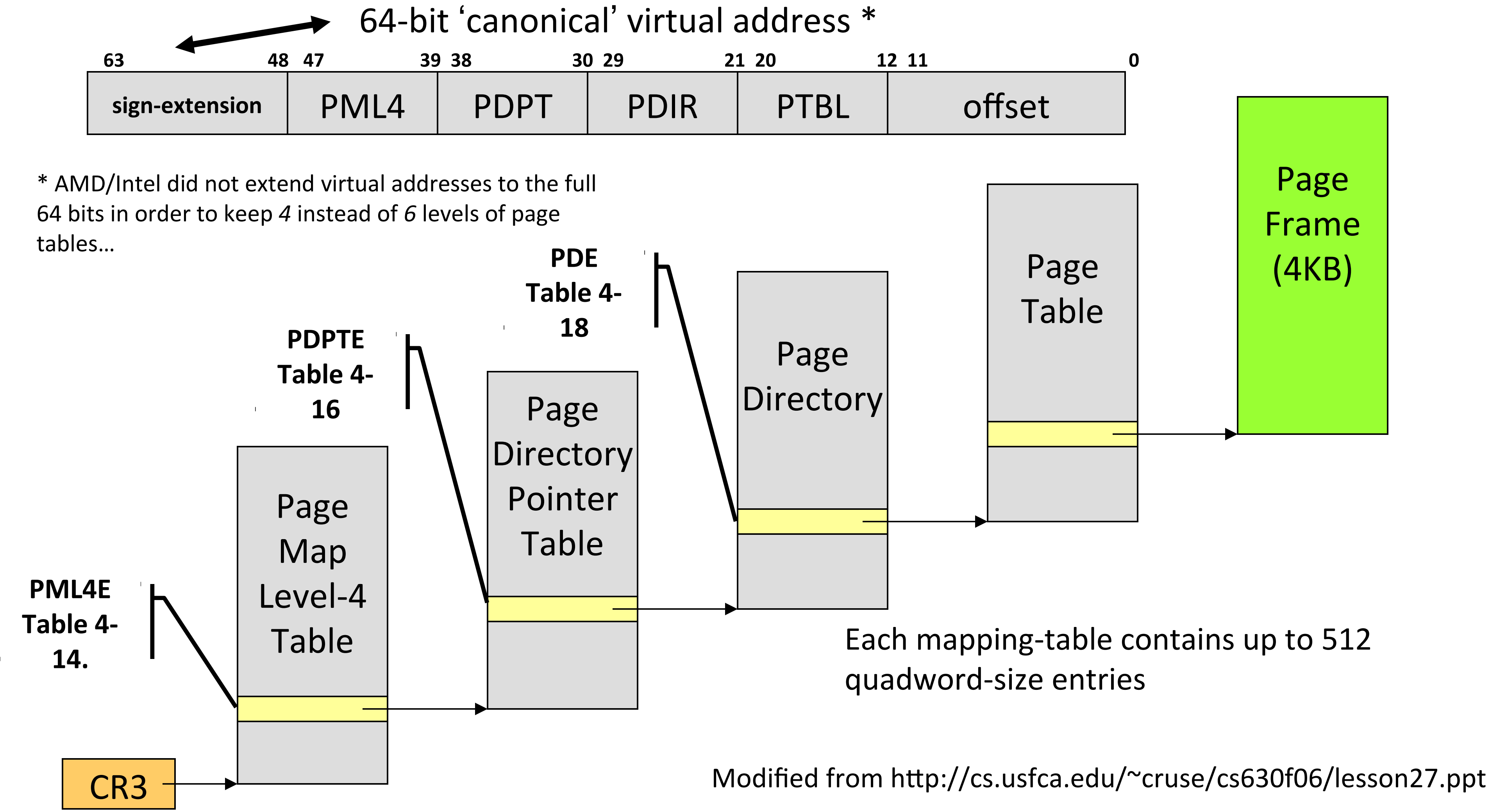
This page mapping example is for 4-KByte pages and the normal 32-bit physical address size.

*Physical Address

This page mapping example is for 4-KByte pages and 40-bit physical address size.

*Physical Address

4-Levels of mapping (4KB pages)



* AMD/Intel did not extend virtual addresses to the full 64 bits in order to keep *4* instead of *6* levels of page tables…

Each mapping-table contains up to 512 quadword-size entries

Modified from http://cs.usfca.edu/~cruse/cs630f06/lesson27.ppt

Page-Table entry format (4KB pages)

63	62 52	51 ☆ M	M-1 32
XD	available	Reserved [51 : M] must be 0	Base Address [(M-1) : 32]

31 12	11 9	8								0
Base Address [31 : 12]	avail	G	PAT	D	A	PCD	PWT	S/U	R/W	P

Legend:

P = present (0=no, 1=yes)
R/W (0=read-only, 1=writable)
S/U (0=supervisor-only, 1=user)
A = accessed (0=no, 1=yes)
D = dirty (0=no, 1=yes)
XD = e(X)ecute (D)isable

PWT = Page Write-Through (0=no, 1=yes)
PCD = Page Caching Disable (0=no, 1=yes)
PAT = Page-Attribute Table-Index
G = Global page (1=yes, 0=no)
M = 12+***MAXPHYADDR*** ☆

Modified from http://cs.usfca.edu/~cruse/cs630f06/lesson27.ppt

MAXPHYADDR (1)

- CPUID.80000008H:EAX[7:0] reports the **physical**-address width supported by the processor.
  - Ours will probably be 36-bits (64 GB)
- For processors that do not support CPUID function 80000008H, the width is generally 36 bits if CPUID.01H:EDX.PAE [bit 6] = 1 and 32 bits otherwise.
- This width is referred to as MAXPHYADDR and is at most 52 bits.

RIP-relative addressing

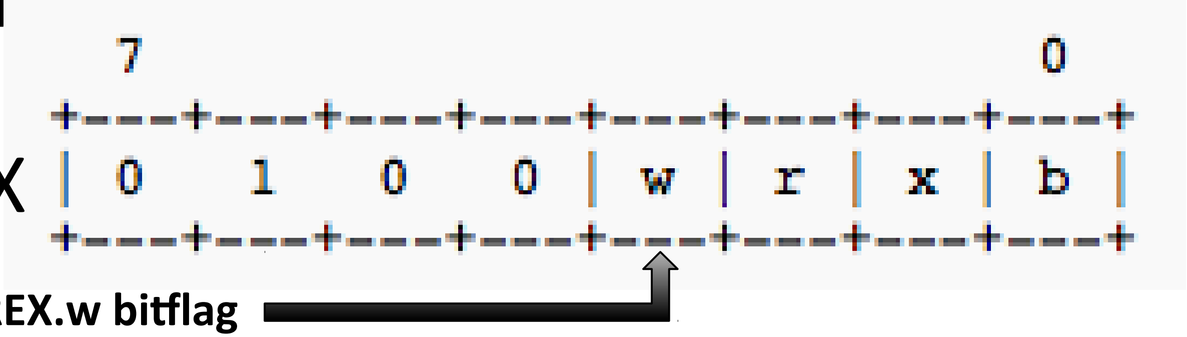
- In 64-bit mode, the RIP register is the instruction pointer.
  - This register holds the 64-bit offset of the next instruction to be executed.
- 64-bit mode also supports a technique called RIP-relative addressing.
  - Using this technique, the effective address is determined by adding a displacement to the RIP of the next instruction.
- ***Some assemblers handle RIP-relative stuff differently*** ☹
  - http://codegurus.be/codegurus/Programming/riprelativeaddressing_en.htm

RIP relative addressing example

```
; New method
mov ah, [rip] ; since RIP points to the next instruction aka NOP, ah now holds 0x90
nop

; Alternative new method
lea rbx, [rip] ; RBX now points to the next instruction
nop
cmp byte ptr [rbx], 90h    ; Should be equal!

; Old method (using 64-bit addressing!)
call $ + 5    ; A 64-bit call instruction is still 5 bytes wide!
pop rbx
add rbx, 5   ; RBX now points to the next instruction aka NOP
nop
mov al, [rbx]

; AH and AL should now be equal :)
cmp ah, al                    ; Ref: http://codegurus.be/codegurus/Programming/riprelativeaddressing_en.htm#Mode64
```

REX Prefix

- REX (byte) prefixes are used to generate 64-bit operand sizes or to reference registers R8-R15.
- If REX.w = 1, a 64-bit operand size is used.
- See Intel Vol. 2 Section 2.2.1.2 for details



REX

```
  7                             0
+---+---+---+---+---+---+---+---+
| 0   1   0   0 | w | r | x | b |
+---+---+---+---+---+---+---+---+
```

REX.w bitflag

Windows x64 calling convention (1)

- Argument passing (use registers)
- 4 register “fast-call” calling convention, with stack-backing for those registers
- The arguments are passed in registers RCX, RDX, R8, and R9.
- RAX, R10, R11 are volatile (caller saved)

Windows x64 calling convention (2)

- All other registers are non-volatile (callee saved)
  - must be preserved
- Caller responsible for allocating space for parameter
  - must always allocate sufficient space for the 4 register parameters,
  - even if the callee doesn’t have that many (or any) parameters

Getting the Brand String with CPUID

- CPUID.EAX = 0x80000002
  - Characters [0:15] in EAX, EBX, ECX, EDX
- CPUID.EAX = 0x80000003
  - Characters [16:31] in EAX, EBX, ECX, EDX
- CPUID.EAX = 0x80000004
  - Characters [32:47] in EAX, EBX, ECX, EDX

VMX Cpuid.S skeleton (for Linux)

```
.section   .rodata
S0:
     .string “VMX available!”
S1:
     .string     “No VMX!”
.text
.global main
.type main, @function
main:
     pushq %rbp
     movq %rsp, %rbp
     <<set appropriate eax value>>
     cpuid
     <<look at VMX bit in appropriate register>>
     jz   <<no_vmx>>
     leaq    S0(%rip), %rdi
     call     puts
     leave
     ret
```

You can lookup appropriate values in the Intel manual: Vol 2a 3-212, Figure 3-6 (pg 269)

Our general cpuid.asm skeleton (for Windows)

```
_CpuId PROC
    push    rbp
    mov     rbp, rsp
    push    rbx
    push    rsi

    mov         [rbp+18h], rdx
    mov         eax, ecx
    cpuid
    mov         rsi, [rbp+18h]
    mov         [rsi], eax
    mov         [r8], ebx
    mov         [r9], ecx
    mov         rsi, [rbp+30h]
    mov         [rsi], edx

    pop         rsi
    pop         rbx
    mov         rsp, rbp
    pop         rbp
    ret
_CpuId ENDP
```

Function prototype:

```
VOID _CpuId (
        ULONG32 leaf,
        OUT PULONG32 ret_eax,
        OUT PULONG32 ret_ebx,
        OUT PULONG32 ret_ecx,
        OUT PULONG32 ret_edx
);
```

Example:
Check for 64-bit using intrinsic cpuid

```
typedef union
_CpuId {
        int i[4];
        struct {
                int eax;
                int ebx;
                int ecx;
                int edx;
        };
} CpuId_t;
```

```
int CheckFor64Bit() {
        int eax;
        CpuId_t regs; // eax, ebx, ecx, edx
        char bitres;

        eax = 0x80000001;
        __cpuid(regs.i, eax);
        bitres = _bittest((long*) &regs.edx, 29);
        return bitres ? 1 : 0;
}
```

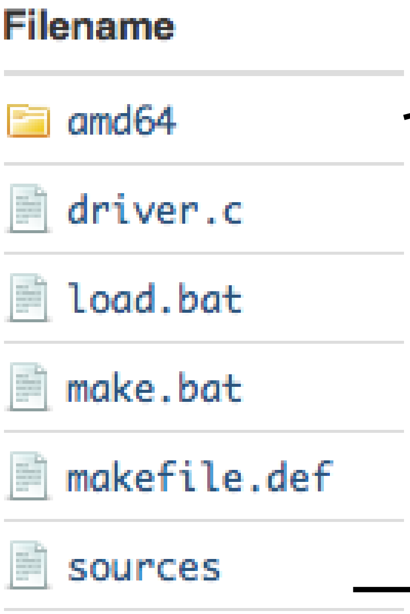
Intrinsic = Visual Studio supplied C macro (i.e. built in)

Lab: CPUID + VMX

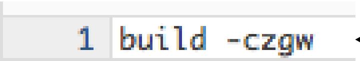
- Purpose
  - x86 Assembly refresher
- Note
  - No more inline assembly on 64-bit Windows
- Steps
  - Setup your coding environment
  - Implement code in assembly to determine whether your CPU supports VMX, and for fun AESNI if you like
  - You can lookup appropriate values in the Intel manual Vol 2a 3-212, Figure 3-6. (Learn to search the manual!)
  - Implement grabbing the brand string—we'll be playing with that later

64-bit driver notes (1)

You will implement here:

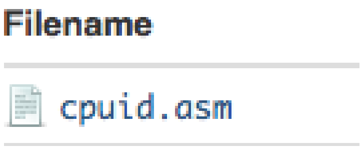


Filename
amd64
driver.c
load.bat
make.bat
makefile.def
sources

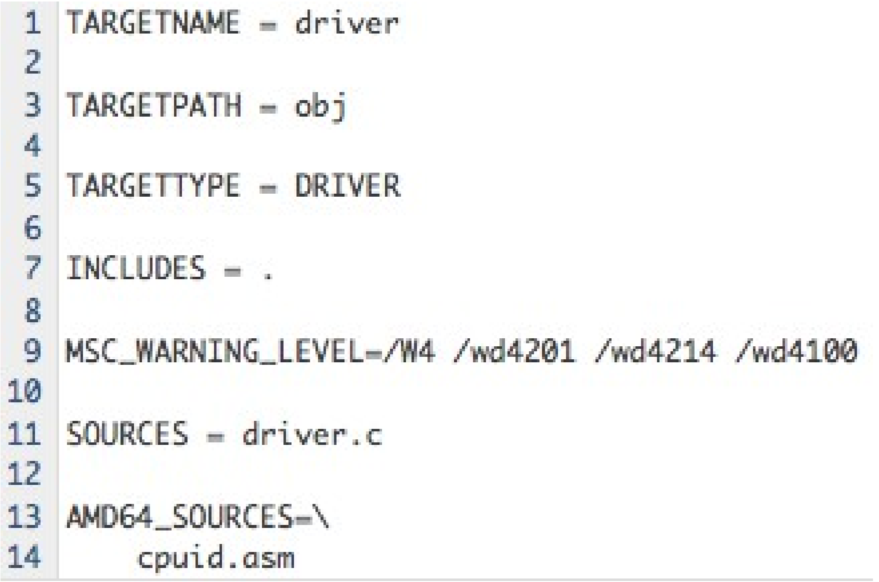


```
build -czgw
```

You will implement here:

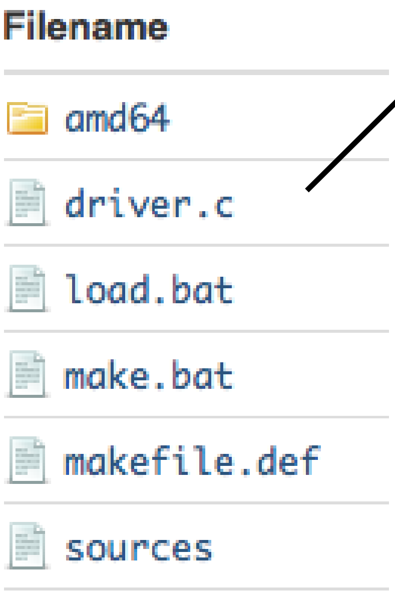


Filename
cpuid.asm

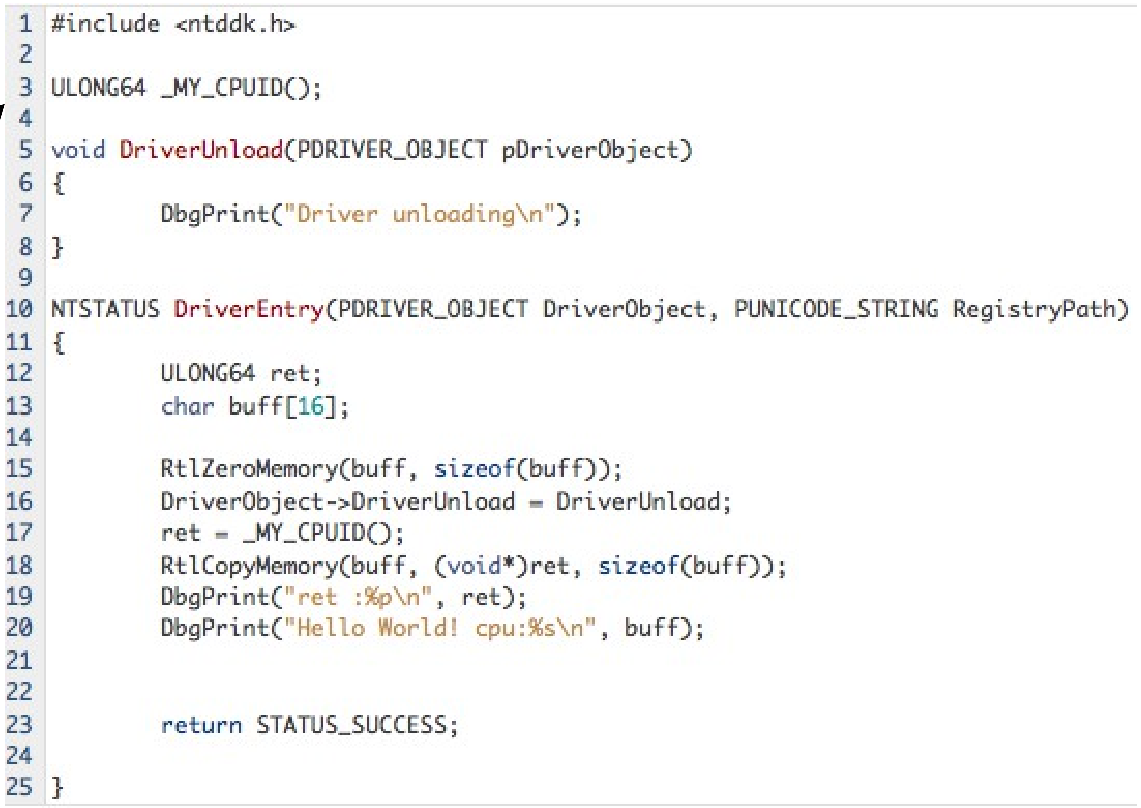


```
TARGETNAME = driver

TARGETPATH = obj

TARGETTYPE = DRIVER

INCLUDES = .

MSC_WARNING_LEVEL=/W4 /wd4201 /wd4214 /wd4100

SOURCES = driver.c

AMD64_SOURCES=\
    cpuid.asm
```

64-bit driver notes (2)

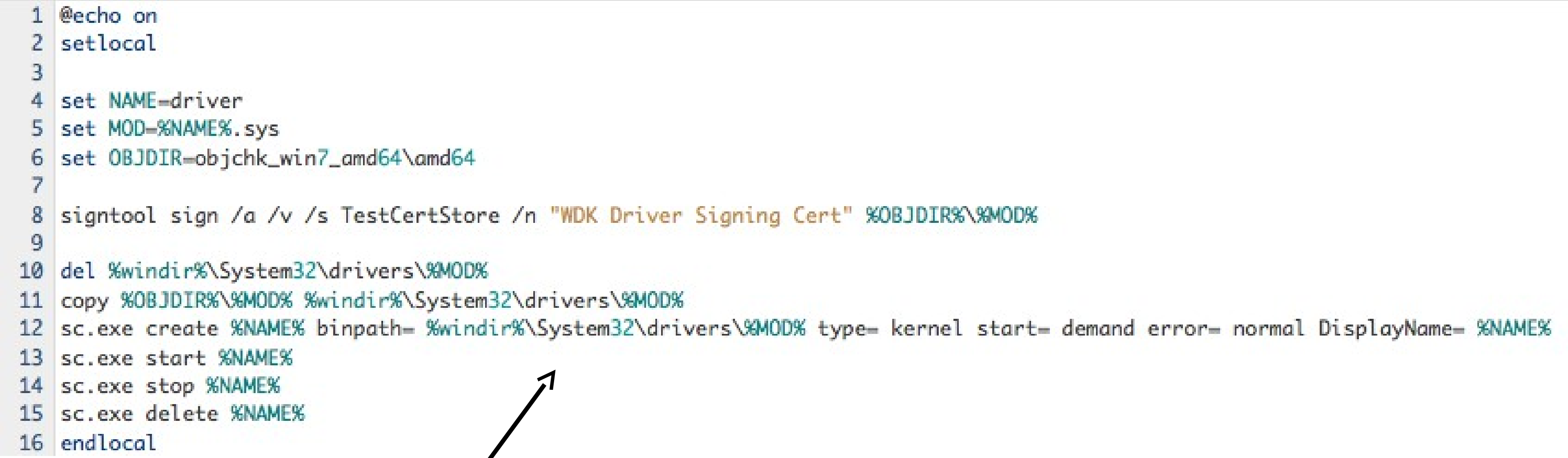


Filename
amd64
driver.c
load.bat
make.bat
makefile.def
sources

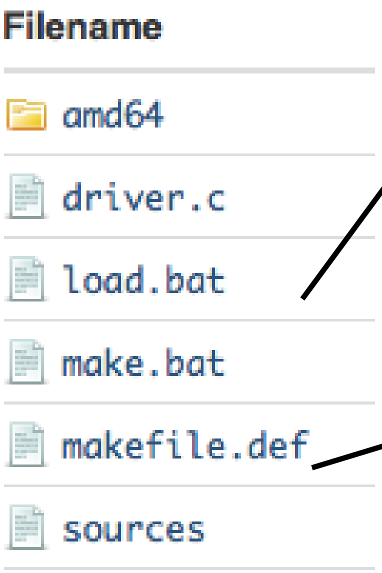


```c
#include <ntddk.h>

ULONG64 _MY_CPUID();

void DriverUnload(PDRIVER_OBJECT pDriverObject)
{
	DbgPrint("Driver unloading\n");
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
	ULONG64 ret;
	char buff[16];

	RtlZeroMemory(buff, sizeof(buff));
	DriverObject->DriverUnload = DriverUnload;
	ret = _MY_CPUID();
	RtlCopyMemory(buff, (void*)ret, sizeof(buff));
	DbgPrint("ret :%p\n", ret);
	DbgPrint("Hello World! cpu:%s\n", buff);


	return STATUS_SUCCESS;

}
```

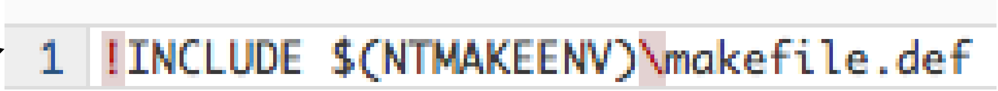
64-bit driver notes (3)



```
@echo on
setlocal

set NAME=driver
set MOD=%NAME%.sys
set OBJDIR=objchk_win7_amd64\amd64

signtool sign /a /v /s TestCertStore /n "WDK Driver Signing Cert" %OBJDIR%\%MOD%

del %windir%\System32\drivers\%MOD%
copy %OBJDIR%\%MOD% %windir%\System32\drivers\%MOD%
sc.exe create %NAME% binpath= %windir%\System32\drivers\%MOD% type= kernel start= demand error= normal DisplayName= %NAME%
sc.exe start %NAME%
sc.exe stop %NAME%
sc.exe delete %NAME%
endlocal
```



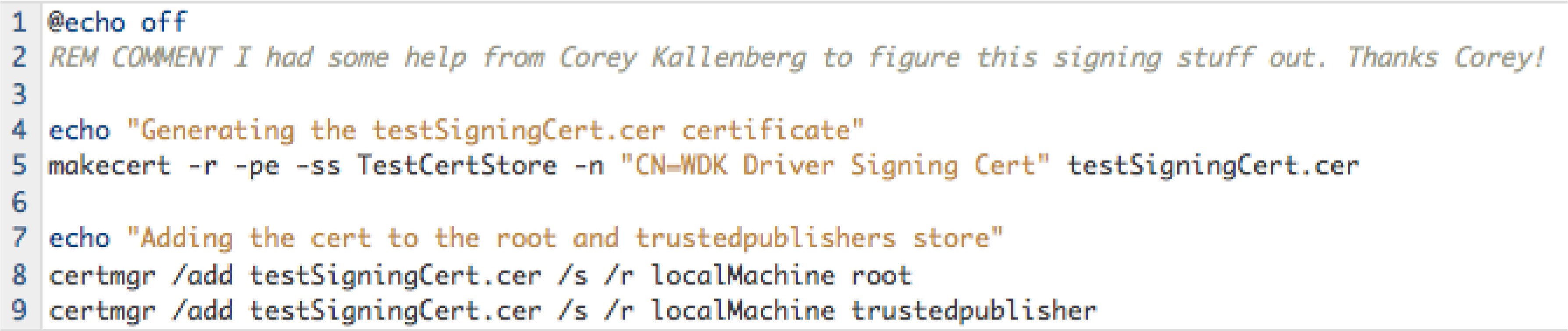
Filename
amd64
driver.c
load.bat
make.bat
makefile.def
sources



```
!INCLUDE $(NTMAKEENV)\makefile.def
```

Generating a certificate

- Windows 7 requires signed drivers
  - We can self-sign if we boot into "Test signing mode"
  - In Admin command prompt:
  - **bcdedit /set testsigning on**
- Self signed drivers:



```
@echo off
REM COMMENT I had some help from Corey Kallenberg to figure this signing stuff out. Thanks Corey!

echo "Generating the testSigningCert.cer certificate"
makecert -r -pe -ss TestCertStore -n "CN=WDK Driver Signing Cert" testSigningCert.cer

echo "Adding the cert to the root and trustedpublishers store"
certmgr /add testSigningCert.cer /s /r localMachine root
certmgr /add testSigningCert.cer /s /r localMachine trustedpublisher
```

Back to virtualization…

- Why is virtualization useful?
- How complex is it to implement?
- What inherent challenges can be expected?
- What techniques have proven successful?

Popek and Goldberg Virtualization Criterion

- POPEK, G. J., GOLDBERG, R. P., "Formal requirements for virtualizable third generation architectures," ACM Communications, July **1974**
- Equivalence / Fidelity
  - A program running under the VMM should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.
- Resource control / Safety
  - The VMM must be in complete control of the virtualized resources.
- Efficiency / Performance
  - A statistically dominant fraction of machine instructions must be executed without VMM intervention.

Different strokes for different folks…

- CPU Simulation
- Binary translation
- Para-virtualization
- Hardware assist +/- some software emulation
  - Emulation required for supporting some x86 guests (i.e., real-mode) even with hardware virtualization

Who is what?

- Full virtualization (aka emulation)
  - Bochs and QEMU
- Paravirtualization
  - Xen, VMware
- Binary Translation
  - VMware, VirtualPC, VirtualBox, QEMU
- Hardware Virtualization
  - Xen, VMware, VirtualPC, VirtualBox, KVM, …

Software Virtualization Challenges

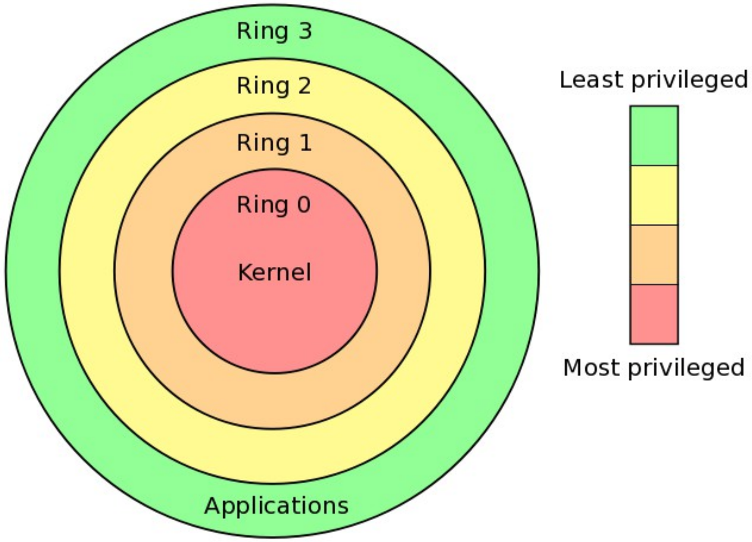
- CPUID instruction
- Ring Aliasing
- Ring Compression
- Memory addressing
- Non-faulting guest access to privileged state
- 17 instructions don't meet Popek and Goldberg criteria [Lawton and Robin] (citation)

CPUID instruction

- Returns processor identification and feature information
- Thought: When employing virtualization, are there certain undesirable features not to be exposed to the guest?
  - Some of these features could make the guest believe it can do things it can't
  - Might want to mask off some features from guest (virtualization)

Ring Aliasing

- Ring 0 is most privileged
- OS kernels assume to be running at ring 0
  - Our guest VM is no different
- VMM and guest cannot share ring 0
  - If guest isn't in ring 0, could use PUSH CS to figure that out (CPL is in the last two bits of CS register)



See Figure 5-3. Protection Rings for Intel's version

Ring Compression

- IA32 supplies two isolation mechanisms, Segmentation and Paging
- Segmentation isn't available in 64-bit
- So paging is only choice for isolating a guest
- But paging doesn't distinguish between rings 0 – 2
  - See Section 5.11.2 "If the processor is currently operating at a CPL of 0, 1, or 2, it is in supervisor mode; if it is operating at a CPL of 3, it is in user mode."
- And our host kernel is in ring 0 and guest software is in ring 3. No more rings = we're compressed.
  - Therefore, our guest cannot be isolated from user-space applications and cannot be reasonably protected from each other.

Faulting instructions

- CLI (clear interrupt flag) and STI (set interrupt flag
- A ring 3 guest that calls CLI or STI raises CPU exception
- Different choices about how to architect your virtualization environment
  - options: turn these interrupts into virtual interrupts, trap to VMM, binary translation.

Non-faulting instructions (1)

- If you wanted to construct a VMM and use fault-then-emulate to virtualize the guest, x86 would turn around and bite you
  - Some things just don't fault, and silently fail instead (i.e., POPF, LAR)
  - The POPF instruction is an example of sensitive instruction which is non-privileged.

AP / Ric Francis

Non-faulting instructions (2)

- Software can also execute the instructions that read (store) from GDT, IDT, LDT, TR using SGDT, SIDT, SLDT, and STR
  - **at any privilege level**
- If the VMM maintains these registers with unexpected values then clearly the guest can figure that out and violate one of our virtualization criteria

Inline assembly refresh

- __asm {
- }

Lab: Non-faulting instruction

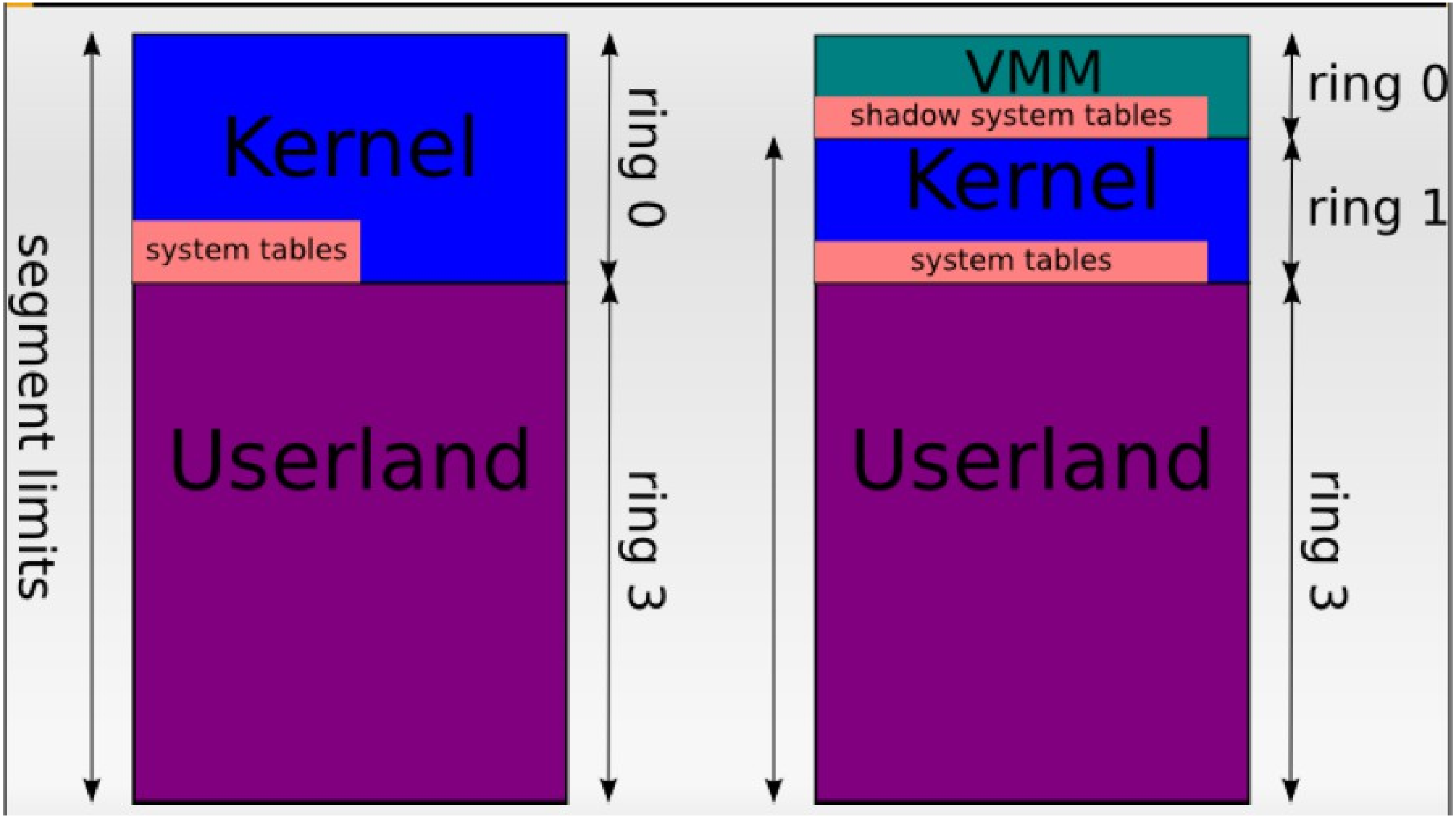
- Go run POPF and confirm that it is bad for virtualization (should be able to do it in ring 3)
- Remember, POPF pops stack into the FLAGS or EFLAGS register
- Use popfd/pushfd
- Get current flags
- Write your own
- Check them

“all will be revealed…”

Memory addressing

- OS kernel expects full (linear) virtual address space. VMM could be in guest address space or mostly in separate address space.
- Why “mostly?”
  - Because there are some data structures to manage transitions from guest to VMM (these structures need to be protected).
- Reminder
  - only protection in 64-bit mode is paging (there is no segmentation)

VMWare-style Virtualization, (pre x64)



PanSec 2009, Tavis Ormandy, Julien Tinnes

Address-space compression

- Refers to the challenges of protecting these portions of the virtual-address space and supporting guest accesses to them
- VMware's older approach could no longer be used on x64 guests because they required segment limits
  - "The virtual machine monitor's trap handler must reside in the guest's address space, because an exception cannot switch address spaces."
  - In theory a task gate in the IDT pointing to a TSS with appropriate CR3 could help, but the performance overhead might have been prohibitive.
  - See http://www.pagetable.com/?p=25 (How retiring segmentation in AMD64 long mode broke VMware)

Access to privileged state

- privileged instructions in the x86 instruction like LGDT, LIDT
- MOV to CR3, CR0, CR4
- For example, contention for IDT between guest and host would result in a crash most likely…

Software based techniques

1. Binary translation
   - Emulation of one instruction set by another for same CPU.
   - When source and target instruction set are the same, it's called instruction set simulation
   - can be done "just in time" (JIT)
   - can do some caching to be more efficient (i.e., hot spot detection)
2. Para-virtualization
   - modification of guest kernel to support being virtualized
   - Can be pretty efficient

Binary Translation

- Can ``defang” privileged instructions such as POPF
- Instruction streams are modified on the fly (think interpreter) to trap offending instruction sequences.
- Two kinds
  - static and dynamic translation

Static Binary Translation

- May not be able to have full code coverage
  - Hidden code in data sections could be reached through an indirect jump or jump into the middle of an instruction
  - A problem if code is specifically trying to thwart the binary translation mechanism

Dynamic Binary Translation

- What do you think?
- Fill it in…

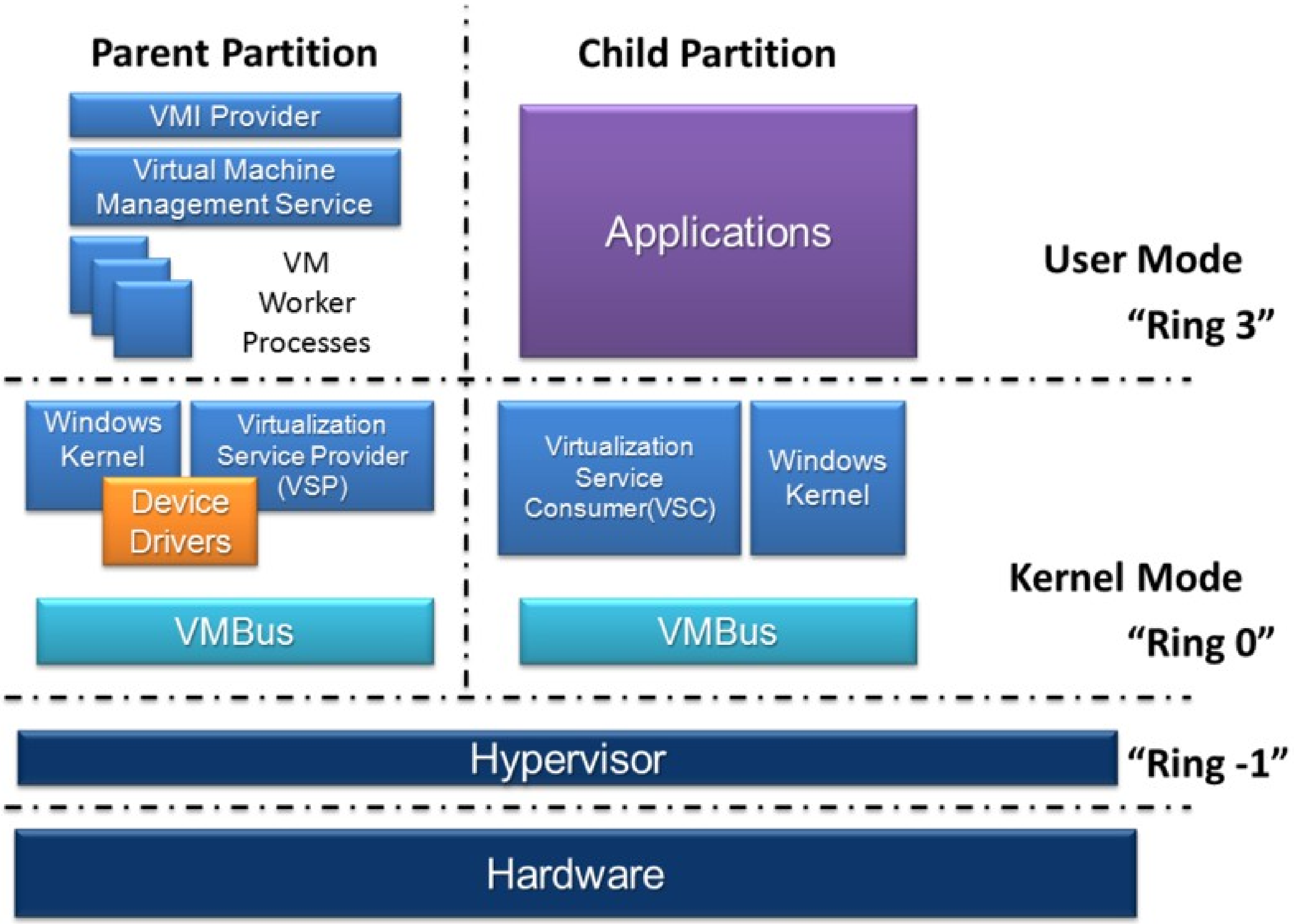
How VMWare done it?

- Early versions of the VMware VMM scanned the instruction stream being executed in the VM and detected the presence of sensitive instructions.
- It then substituted the sensitive instruction with a target instruction and then emulated the action of the original instruction.
- Binary Translation introduced into VMware circa 1999

If interested… read up

- PAYER, M., AND GROSS, T. Requirements for fast binary translation. In 2nd Workshop on Architectural and Microarchitectural Support for Binary Translation (2009).
- PAYER, M., AND GROSS, T. R. Generating low-overhead dynamic binary translators. In SYSTOR'10 (2010).

Microsoft Hyper-V



https://en.wikipedia.org/wiki/File:Hyper-V.png

Microsoft Hyper-V

- A hypervisor instance has to have at least one "parent partition"
- The virtualization stack runs in the parent partition and has direct access to the hardware devices.
- The parent partition then creates the child partitions which host the guest OSs.
- Xen is pretty similar

Lab: Which is it?

- Play with JSLinux (http://bellard.org/jslinux/)
- Run Linux in your web browser…
  - So is it a binary translator or an emulator?
- Read technical notes
  - http://bellard.org/jslinux/tech.html

Review