Ch02 OS Services

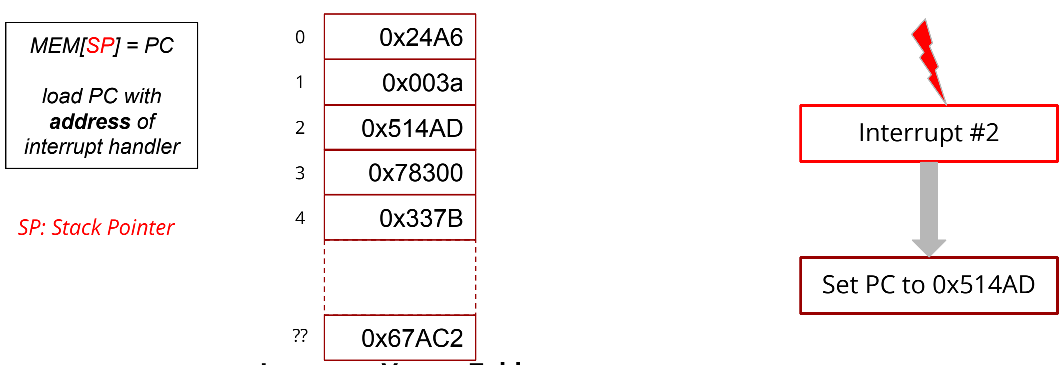
Class Discussion

Services provided by BIOS?

Background Info for Discussion

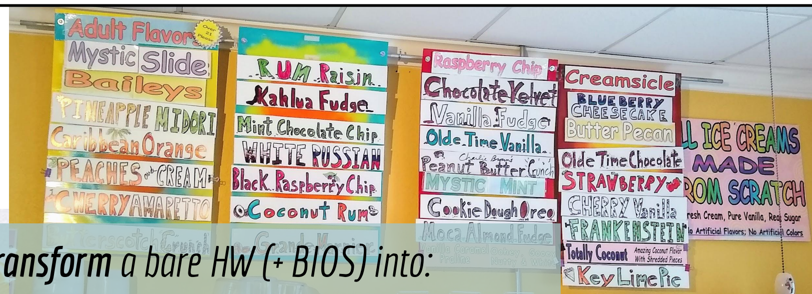
- Bare hardware comes with BIOS (Basic I/O System)
- How do we use these "functions"?
  - BIOS functions are invoked via software interrupt
  - List of BIOS Software Interrupt (Wikiepedia)
- BIOS loads a *small program* (boot loader) from known locations
  - Boot block of hard drive, CDROM, DVD-ROM, USB drive, Network, ...
- The boot loader brings the rest of the OS code
- The OS replaces the BIOS services with its own
  - But we still invoke them via software interrupt

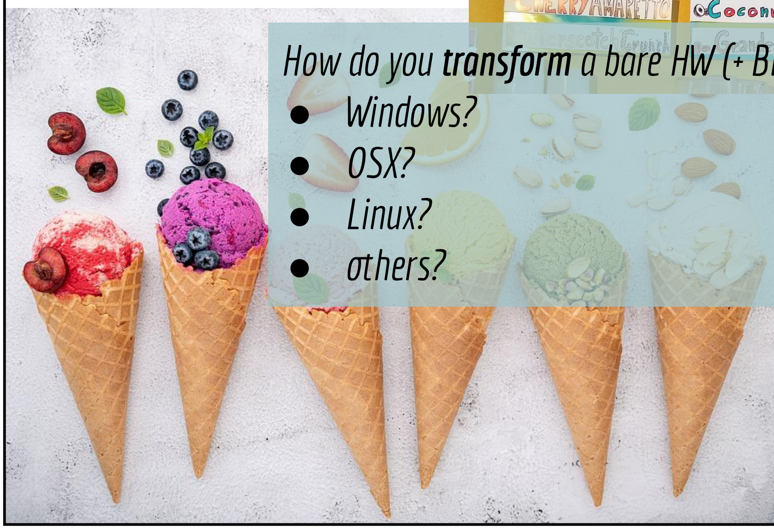
Interrupts: Response in HW & Vector Table



Interrupt Vector Table
(Intel: Interrupt Descriptor Table)

- *Initially at address ZERO in BIOS ROM*
- *Can be overridden by setting IDT Register*





How do you ***transform*** *a bare HW (+ BIOS) into:*

- *Windows?*
- *OSX?*
- *Linux?*
- *others?*

BIOS vs. OS

- **BIOS** of a physical machine (bare hardware) provides very basic services
- Bare hardware + OS ⇒ Computing Environment of a specific "flavor"
  - Bare hardware + Linux = Linux Machine
  - Bare hardware + Windows = Windows Machine
- On a Linux machine, the kernel code replaces/enhances the basic BIOS functions with its own
- *What if you run a (Linux) system program that creates an illusion of a bare hardware (+ BIOS)?*

System Calls

- CPU enters its Interrupt Cycle when
  - There is a hardware interrupt triggered externally / *asynchronously*
  - A program issues a software interrupt (assembly instruction)
- System Call Implementation
  - Linux/OSX
    - On 32-bit architecture `INT 0x80` (assembly instruction)
    - On 64-bit architecture `SYSCALL or SYSENTER` (assembly instruction)
  - DOS/Windows: `INT 0x21` or `INT 0x2E`
- System calls may require parameters to work with
  - How do you supply the parameters?

Hello World in Assembly

MSDOS

```
    ORG 100H


    MOV DX,msg ; Addr of string to print
    MOV AH,9   ; Func#9: display string
    INT 21H    ; DOS call



    MOV AH,4CH ; Func#4C: exit
    INT 21H    ; DOS call


msg DB “Hello World”,”$”
```

Linux 32-bit

```
    GLOBAL _start
SECTION .text
_start
     MOV EAX,4       ; write()
     MOV EBX,1       ; stdout
     MOV ECX, msg
     MOV EDX, msg.len
     INT 80H         ; Linux call

     MOV EAX,1       ; exit()
     INT 80H         ; Linux call

SECTION .data
msg  DB “Hello World”,”$”
.len EQU $ - msg
```

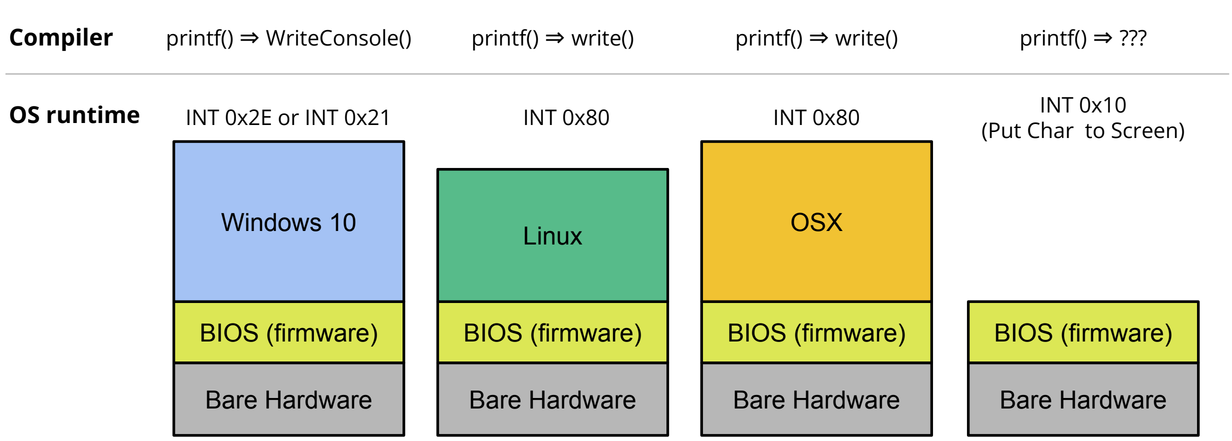
Linux 64-bit

```
    GLOBAL _start
SECTION .text
_start
     MOV RAX,1       ; write()
     MOV RDI,1       ; stdout
     MOV RSI, msg
     MOV RDX, msg.len
     SYSCALL         ; Linux call

     MOV RAX,60      ; exit()
     SYSCALL         ; Linux call

SECTION .data
msg  DB “Hello World”,”$”
.len EQU $-msg
```

Examples: Input/Output

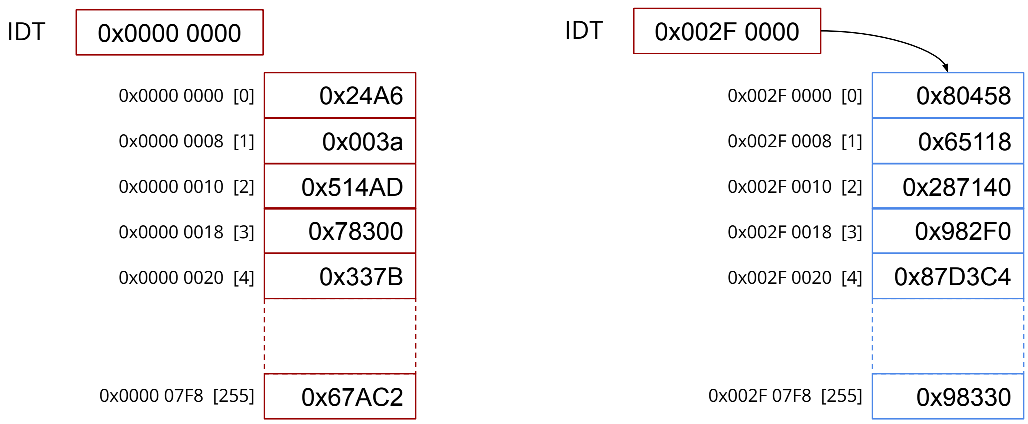
- Short program in various languages: C, C++, Assembly, Rust
  - ~~On WSL~~
  - On Docker Container (Debian)?
  - On Docker Container DOS Box?
  - On Ubuntu VM
- Use `strace` to show `write()` system calls

```
void main() {
  printf (“Hello world”);
}
```



One physical hardware with four different “Look and Feel”

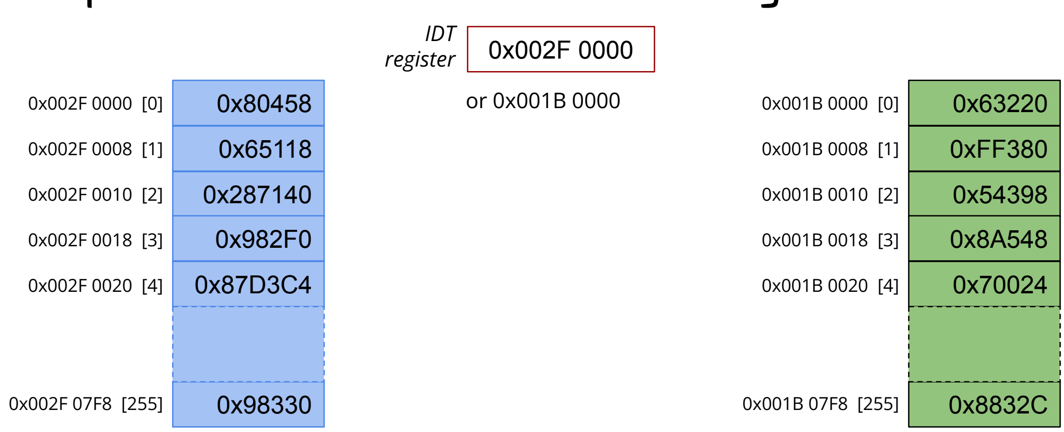
IDT/IVT and IDT register (BIOS vs. OS)



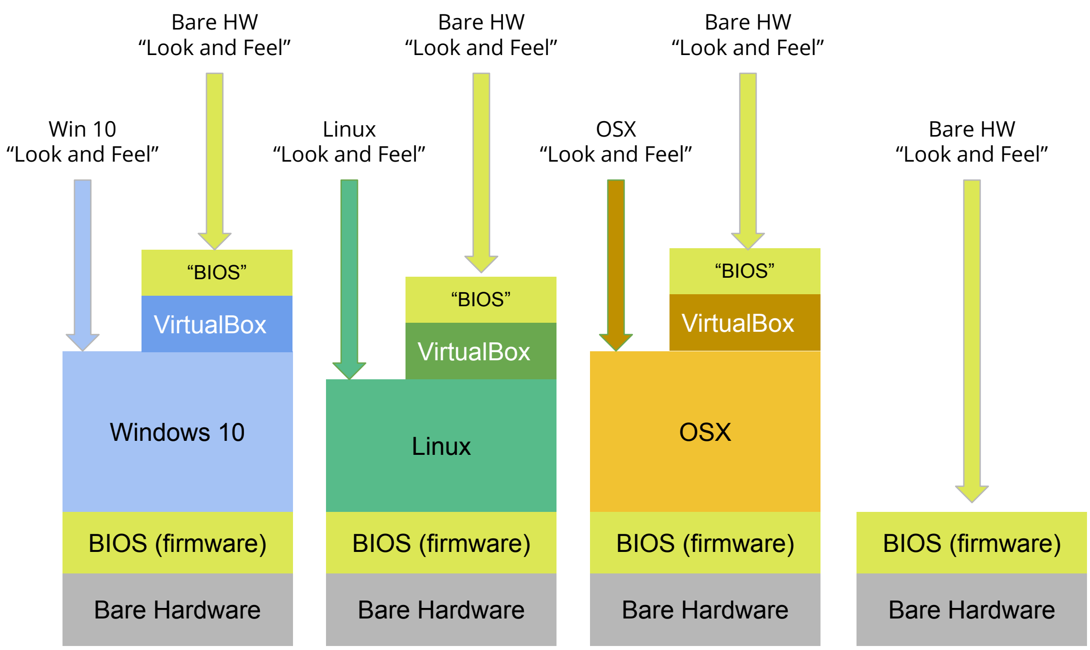
*BIOS IVT **must begin** at absolute address 0x0000 0000 (in ROM)*

*Your OS IVT may begin **anywhere** in RAM*

Multiple IVTs but ONLY ONE IDT register



Virtual Machines



Guest OS Should Run in User Mode (not in Kernel Mode)

Why?

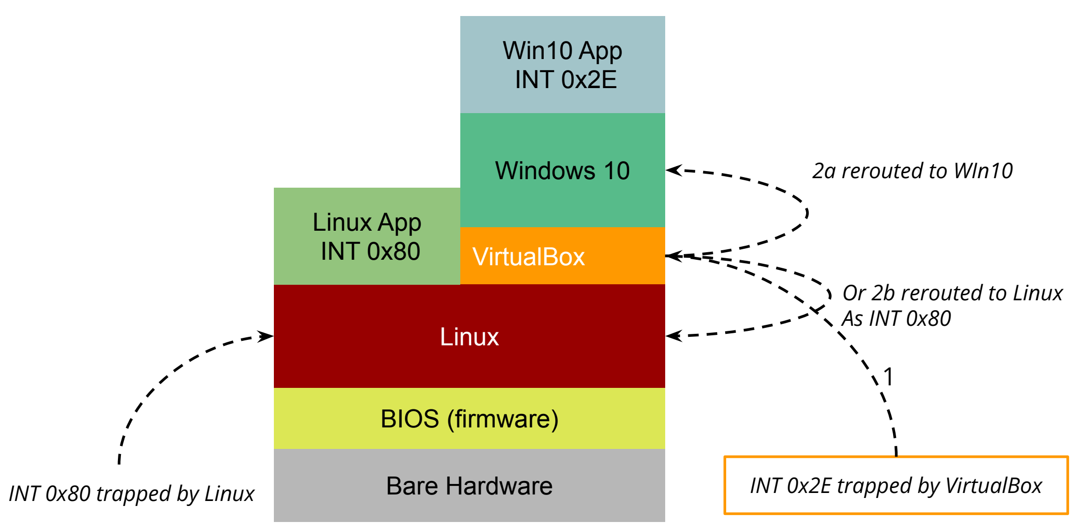
Virtual Machines vs. Emulators

- A Virtual Machine creates an environment that **mimics the bare hardware of the host** (*as much as possible*)
- An Emulator creates an environment that mimics a particular bare hardware, **NOT** necessarily that of the host
  - Android Emulator
- Implications
  - Instructions running on a VM can *run directly* on the host CPU
  - Instructions running on an emulator must first be *translated to equivalent instructions* on the host CPU

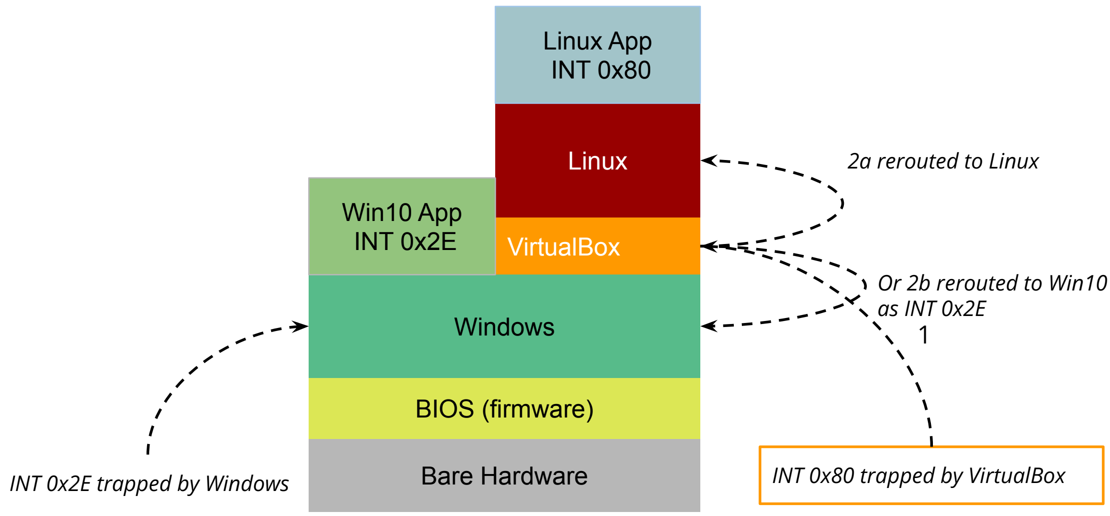
System Calls on VM

How to handle privileged instructions issued by a program running in guest OS?

Windows Guest on Linux Host



Linux Guest on Windows Host



System Calls on VM

- Install a hypervisor / VM monitor that intercepts any system calls originating from a VM
    - CPU should have a additional bit to distinguish “host context” vs. “guest context”
- System calls in host context are handled without involving hypervisor
- System calls in guest context are trapped and *analyzed* by hypervisor and rerouted to either host OS, or guest OS

Hardware Protection Support for Virtualization

- Without Virtualization
  - CPU operates in TWO modes: Kernel Mode and User Mode
  - Only 1-bit is required in the CPU status
- With Virtualization, extra (hardware) bit required
  - CPU operates in FOUR modes
    - (Kernel | User) + (Real| Virtual)
    - 2 bits requires in the CPU status

Real/Virt	User/Kern	CPU Operation Mode
0	0	Kernel mode non-virtualized
0	1	User mode non-virtualized
1	0	Kernel mode virtualized
1	1	User mode virtualized

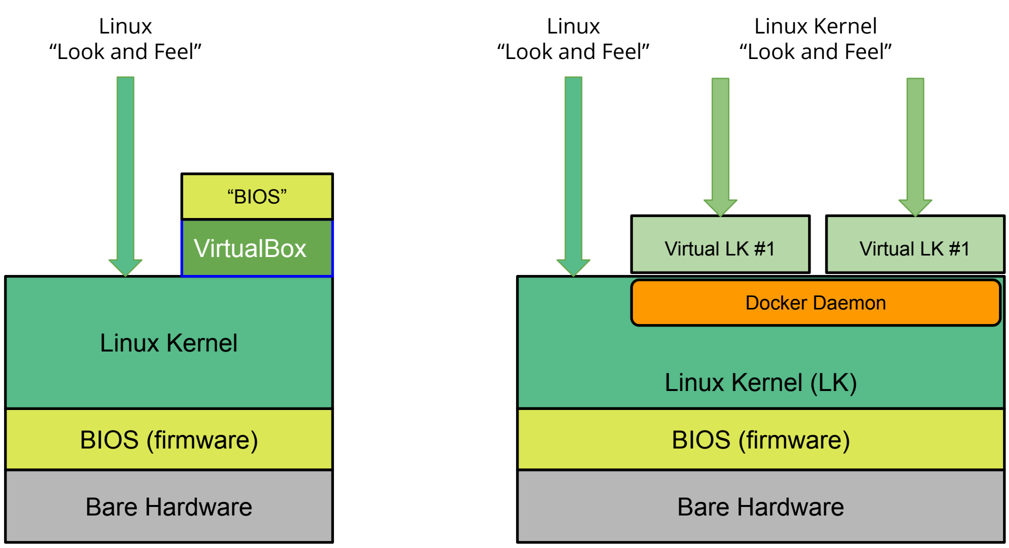
JVM (Java "Virtual Machine") is not a virtual machine!

JVM + JIT (Just In Time compiler) are *almost* a virtual machine!

Virtual Machines vs. Linux Containers

- Each VM instance loads a copy of guest OS
  - Each guest OS is totally isolated from the other guest OSes
- Linux containers are a "minified" Linux environment
  - Multiple Linux Containers running on one host can share the same copy of host kernel
  - Each instance of Linux Container is isolated/sandboxed from the others
- Supporting Linux features for implementing containers
  - chroot: allows a particular directory in a Linux FS to be used as a shadow root
  - kernel namespace: resources in a Linux system are assigned a unique name

Containers



WSL: Lightweight VM = mergeOf(VM, Container)