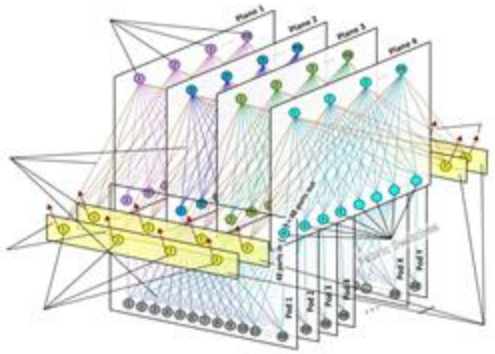
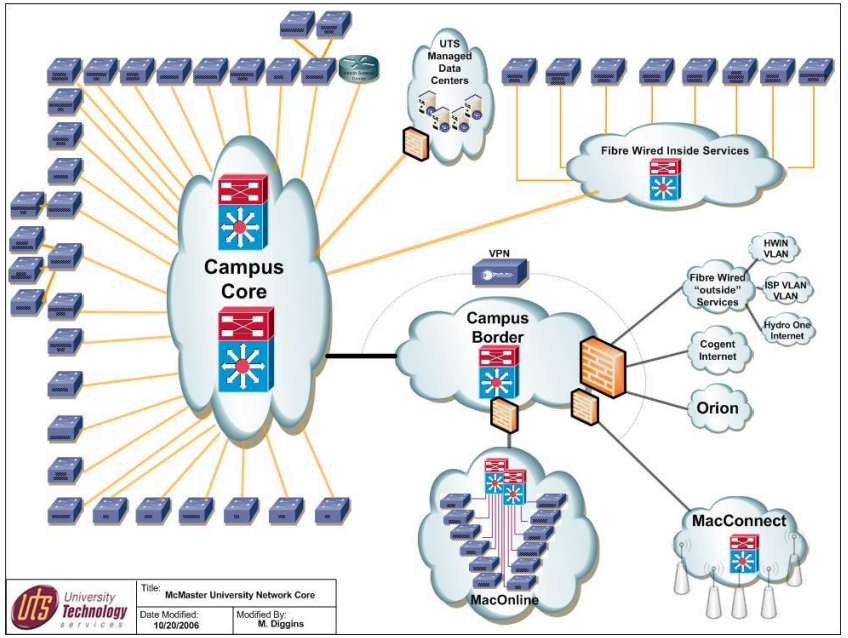
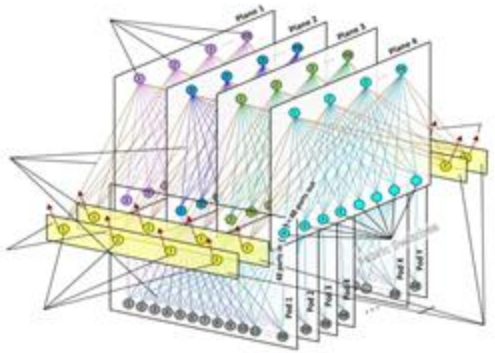


Large Systems:

Design +
Implementation +
Administration





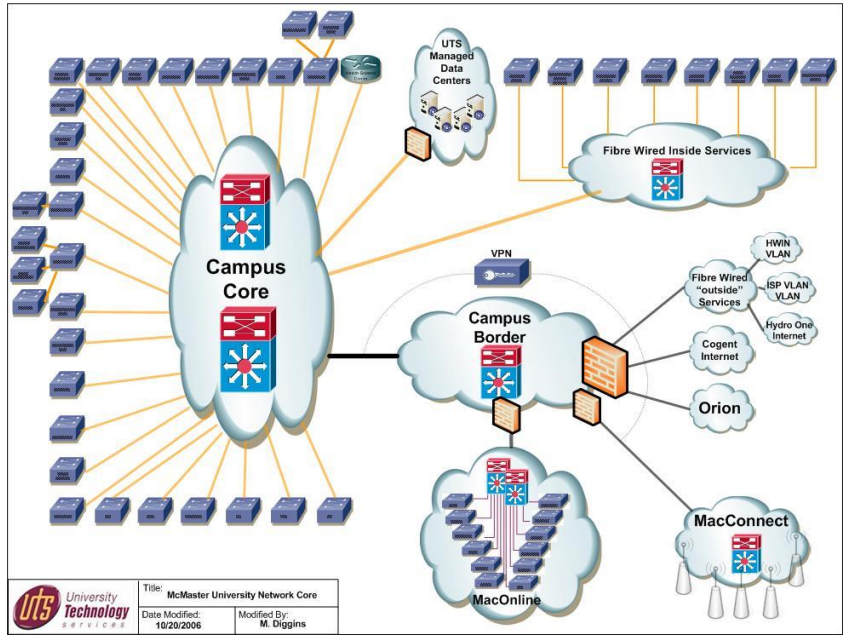
Large Systems:

Design + Implementation

2024-2025

- Week1-L2: Virtualization- Part 2

Shashikant Ilager
shashikantilager.com

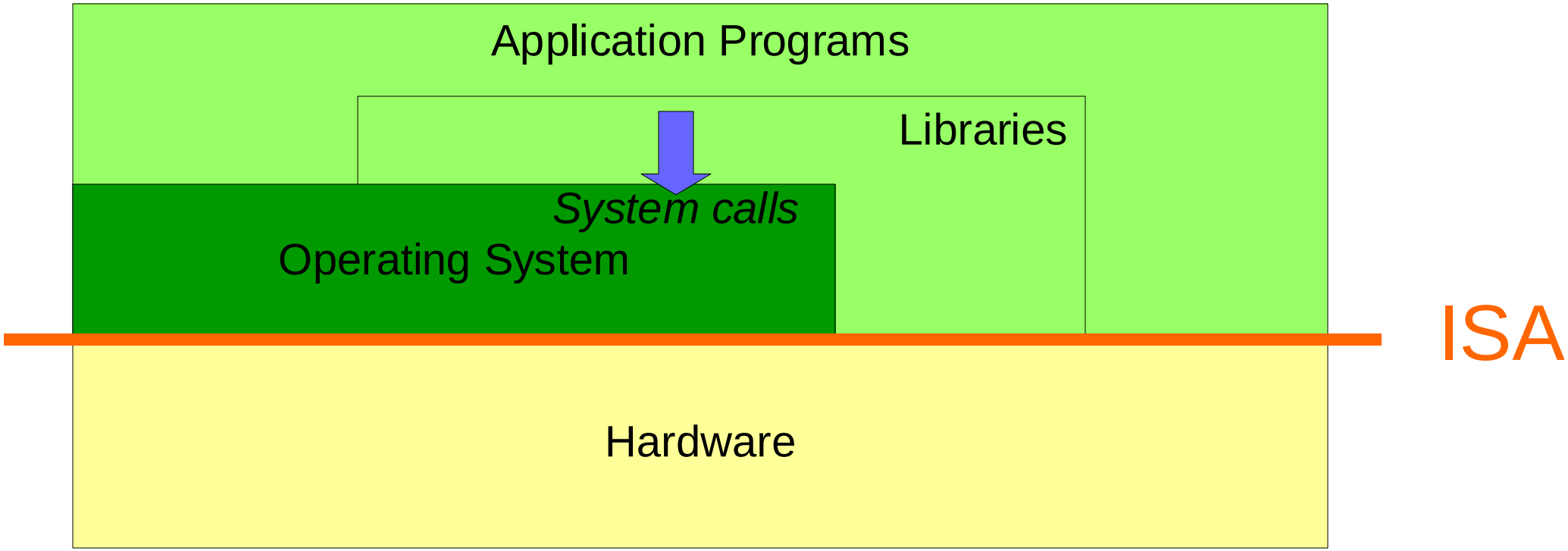


Virtual Machines

Virtualize the machine?

- What is the machine?
- The machine is defined by an interface
- Virtualization = 1 physical machine should offer that interface to multiple virtual machines
- 3 interfaces to choose from:
  - Instruction Set Architecture (ISA)
  - Application Binary Interface (ABI)
  - Application Programming Interface (API)

Interface 1: ISA



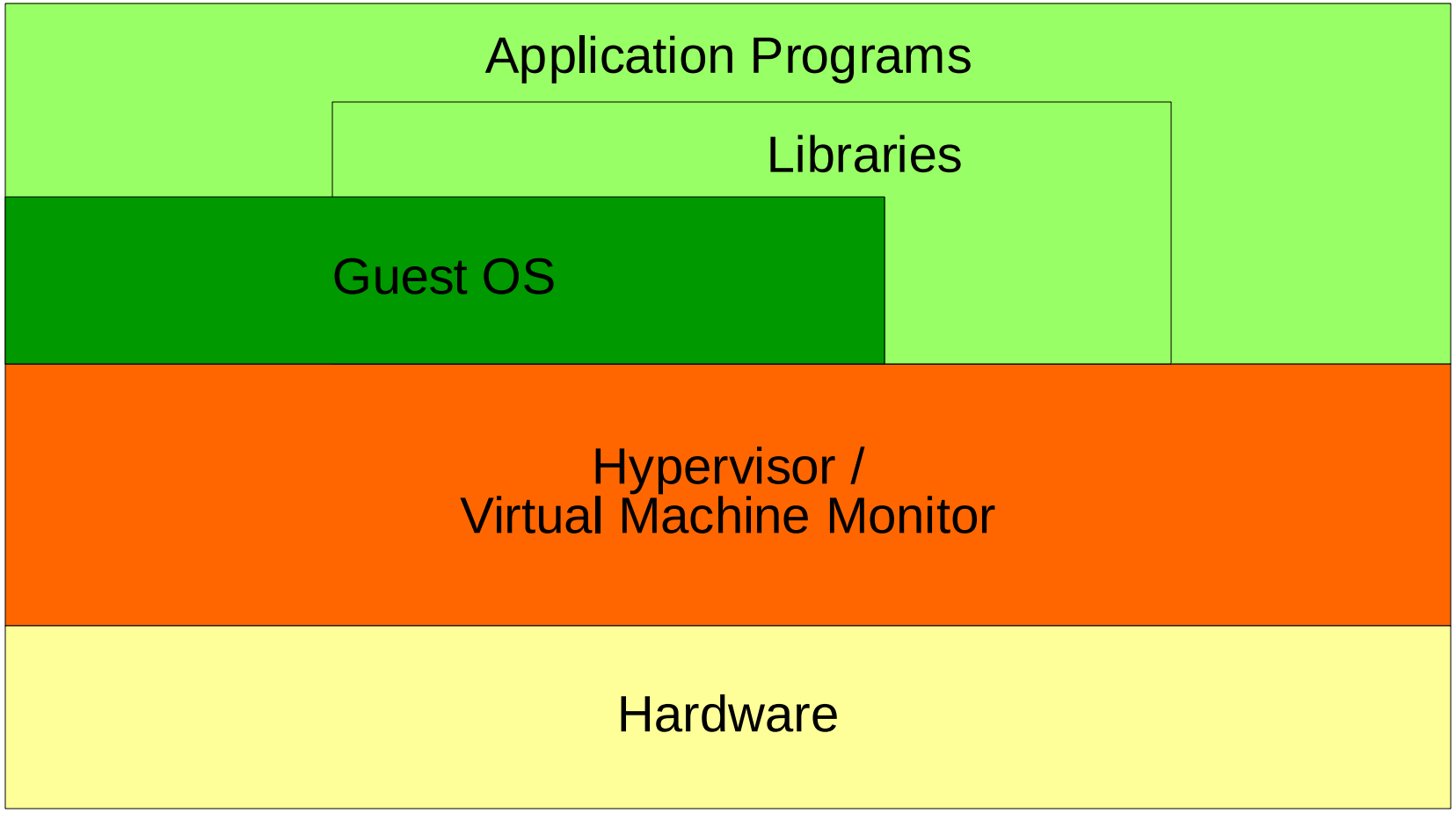
Virtualize a complete machine, that can run an OS, supporting multiple processes = System VM

Example Virtualizing ISA

Support a machine's complete ISA

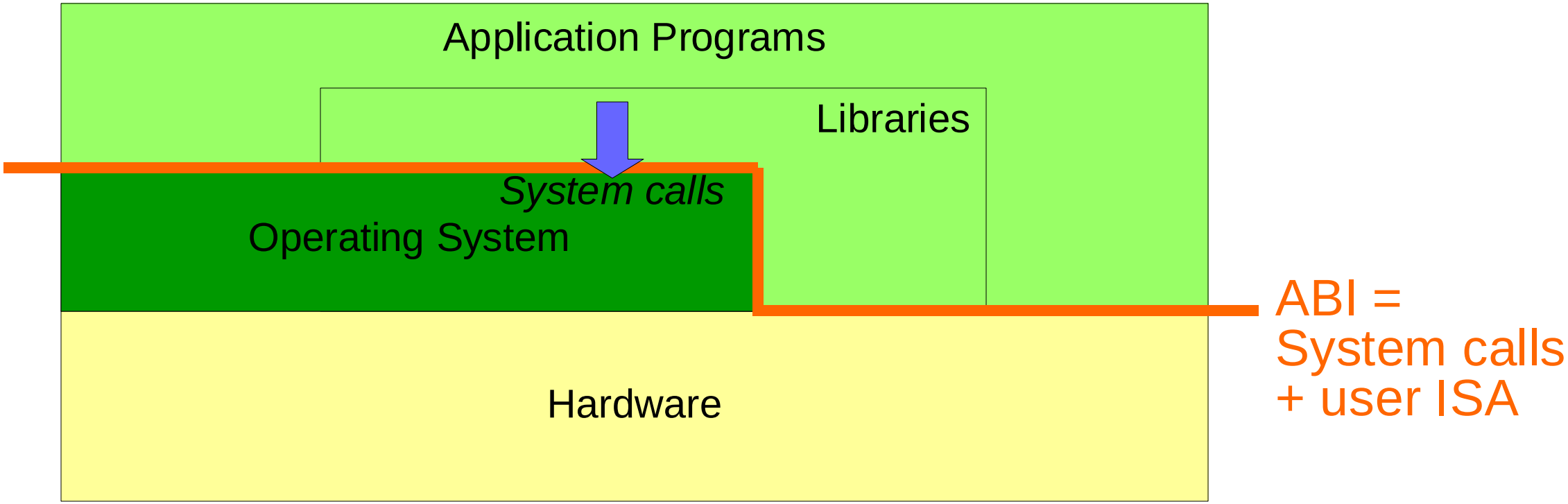
- VM/370
- Xen*
- KVM*
- VMWare ESX
- MS Hyper-V*

System VM Implementation



"Type 1 Hypervisor"

Interface 2: ABI

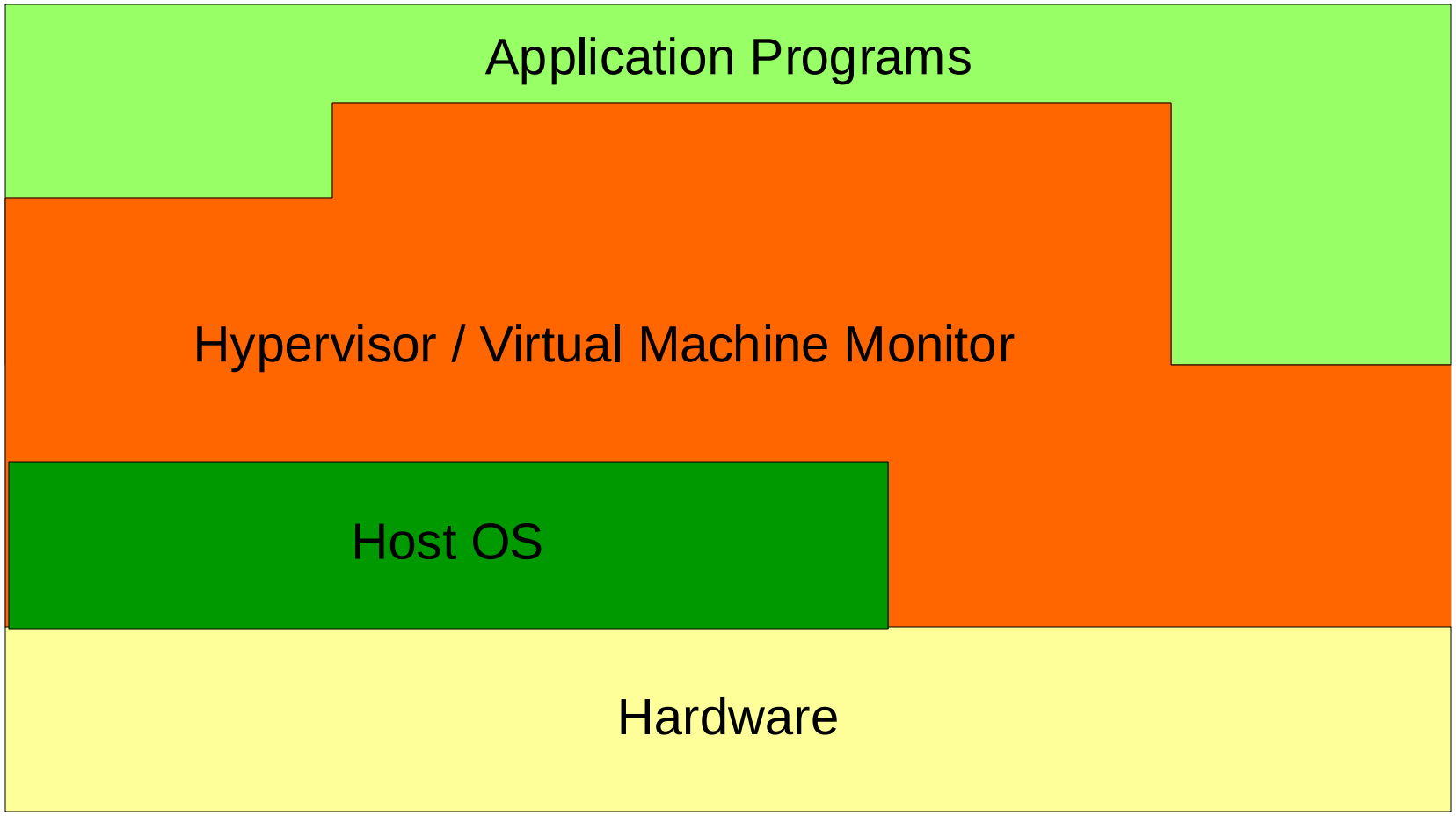


- Virtualize the environment of a single process (binary)
- = Process VM

Example Virtualizing ABI

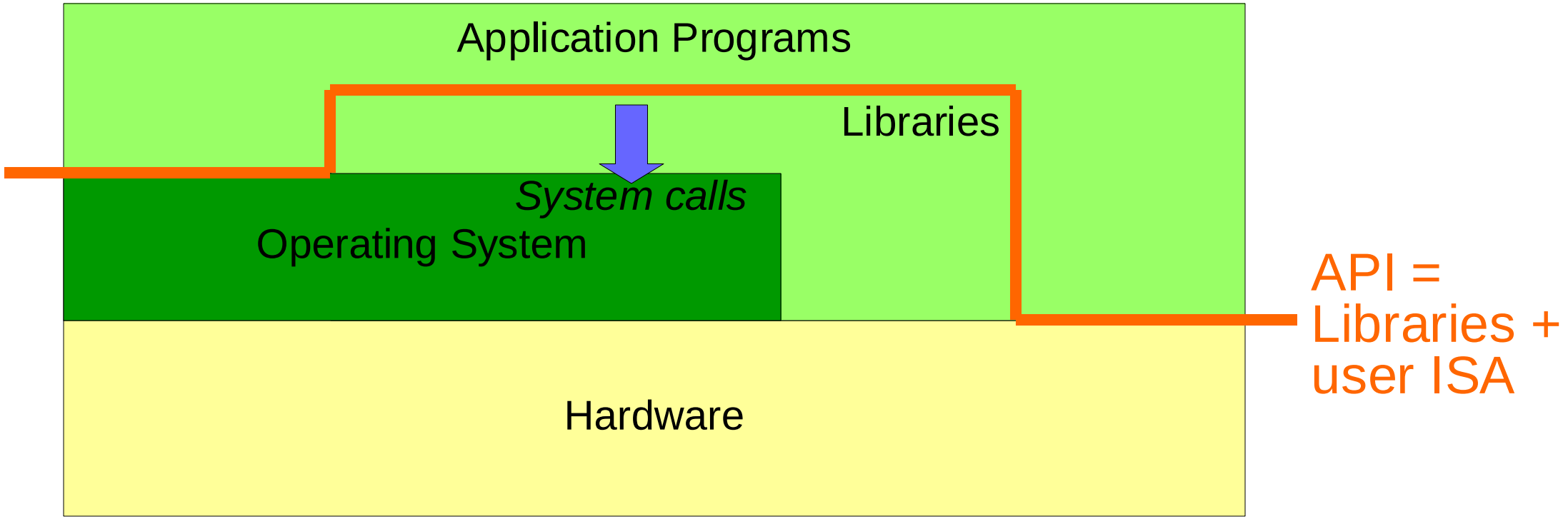
- Run binaries unmodified on a different platform
- Java binaries (.jar)
  - Consist of Java bytecode to be executed on e.g. Linux-ARM
- Wine: Run Win32-x86 binaries on e.g. Linux
- Windows Subsystem for Linux (WSL) version 1
  - Run unmodified Linux binaries on Win10
  - Version 2 is more System VM
- Rosetta
  - Run Intel binaries on Apple ARM CPUs
  - Used when M1 Apple silicon were introduced in 2021 in Macbooks

Process VM Implementation



"Type 2 Hypervisor"

Interface 3: API



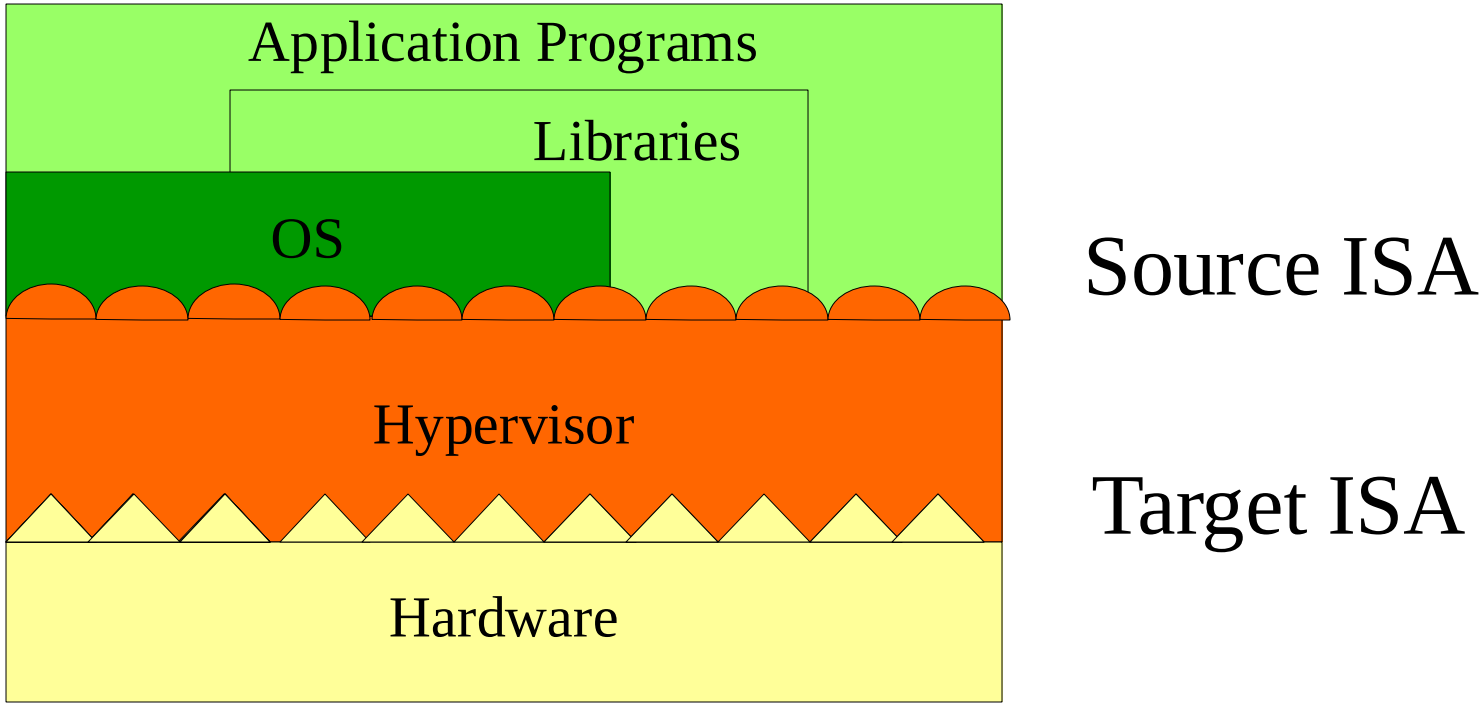
Virtualization provides the same interface at programming language (source) level.

Example Virtualizing API

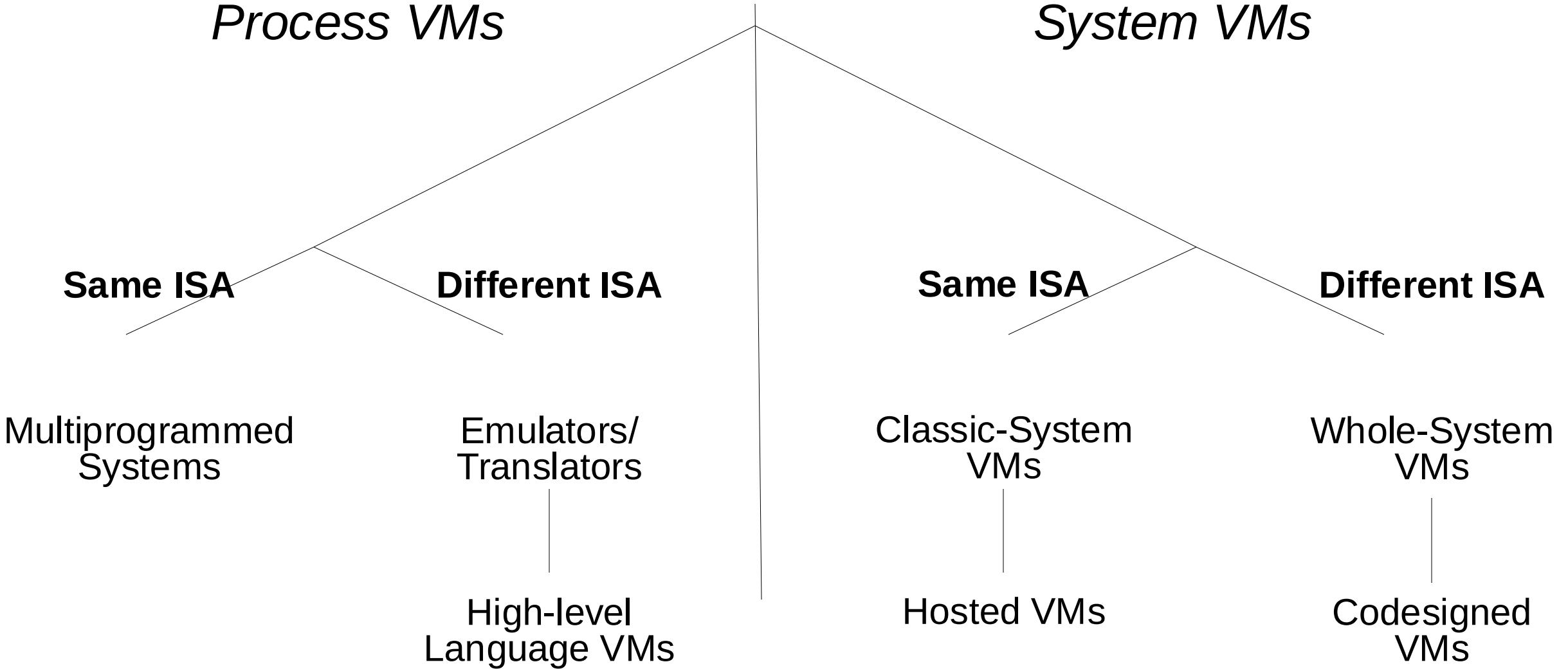
- Recompile applications from source
- Runs on any platform with same API
  - E.g. Linux-x86 and Linux-ARM
- (Assuming platform-independent code)

What ISA? Same or different

- Same: Run Win32-x86 on Linux-x86
- Diff: Run Linux-ARM on Win32-x86

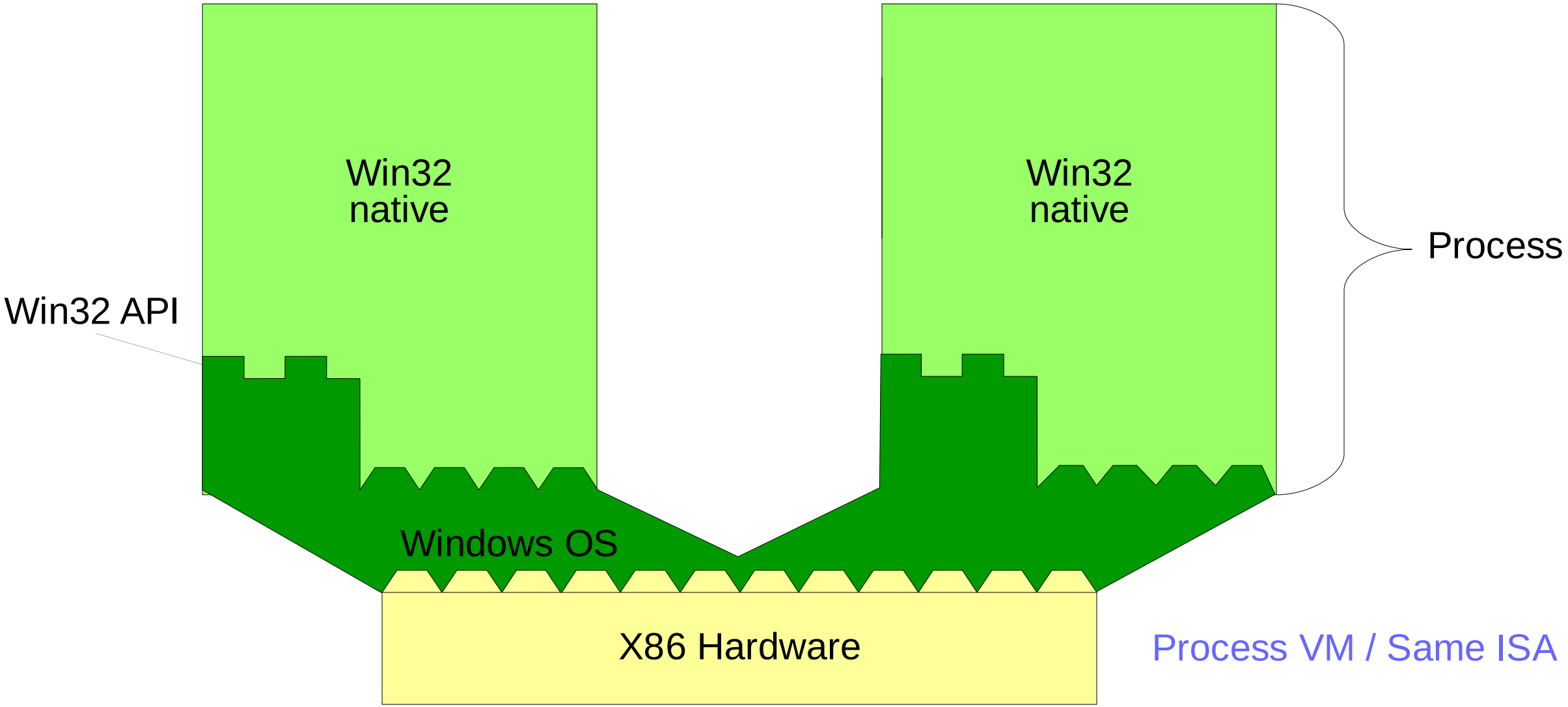


Taxonomy


Process VMs
System VMs
Same ISA
Different ISA
Same ISA
Different ISA
Multiprogrammed Systems
Emulators/ Translators
High-level Language VMs
Classic-System VMs
Hosted VMs
Whole-System VMs
Codesigned VMs

Where does Java stand in the previous taxonomy?

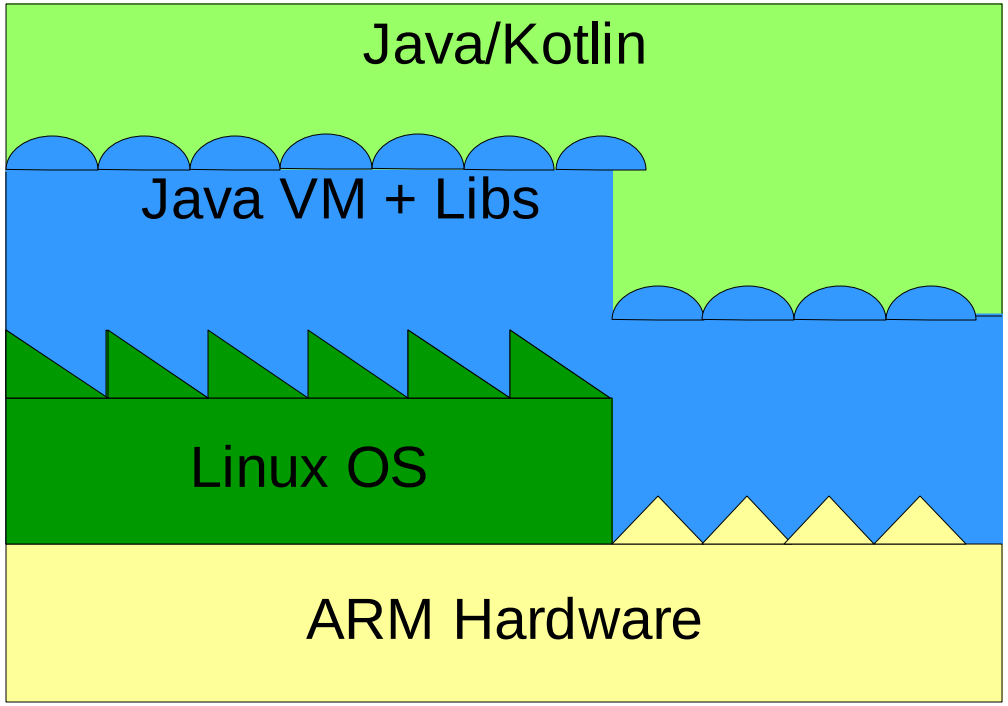
Example: Windows Multiprogramming



One CPU, illusion of processes running in parallel

Example: Android

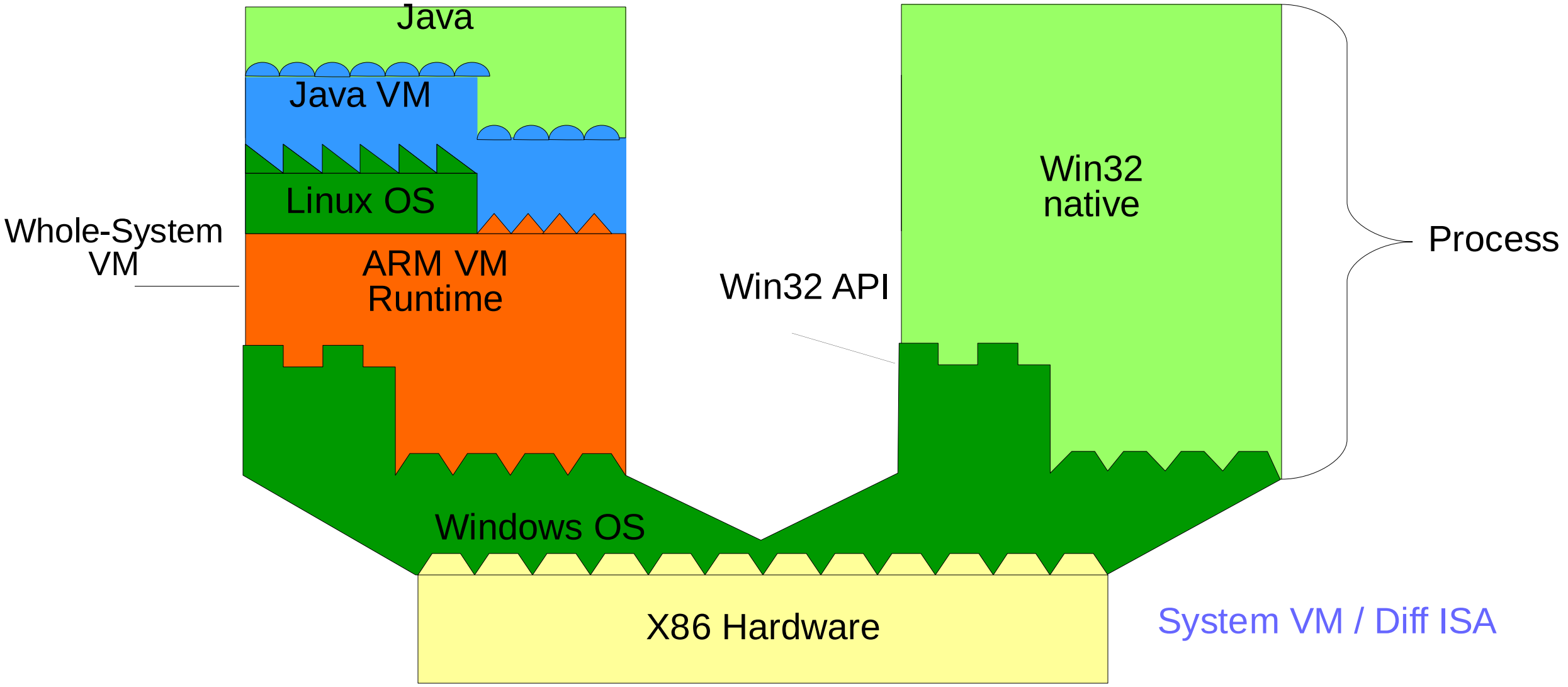
High-level Language VM (HLL-VM)



Different ISA: Java vs. ARM

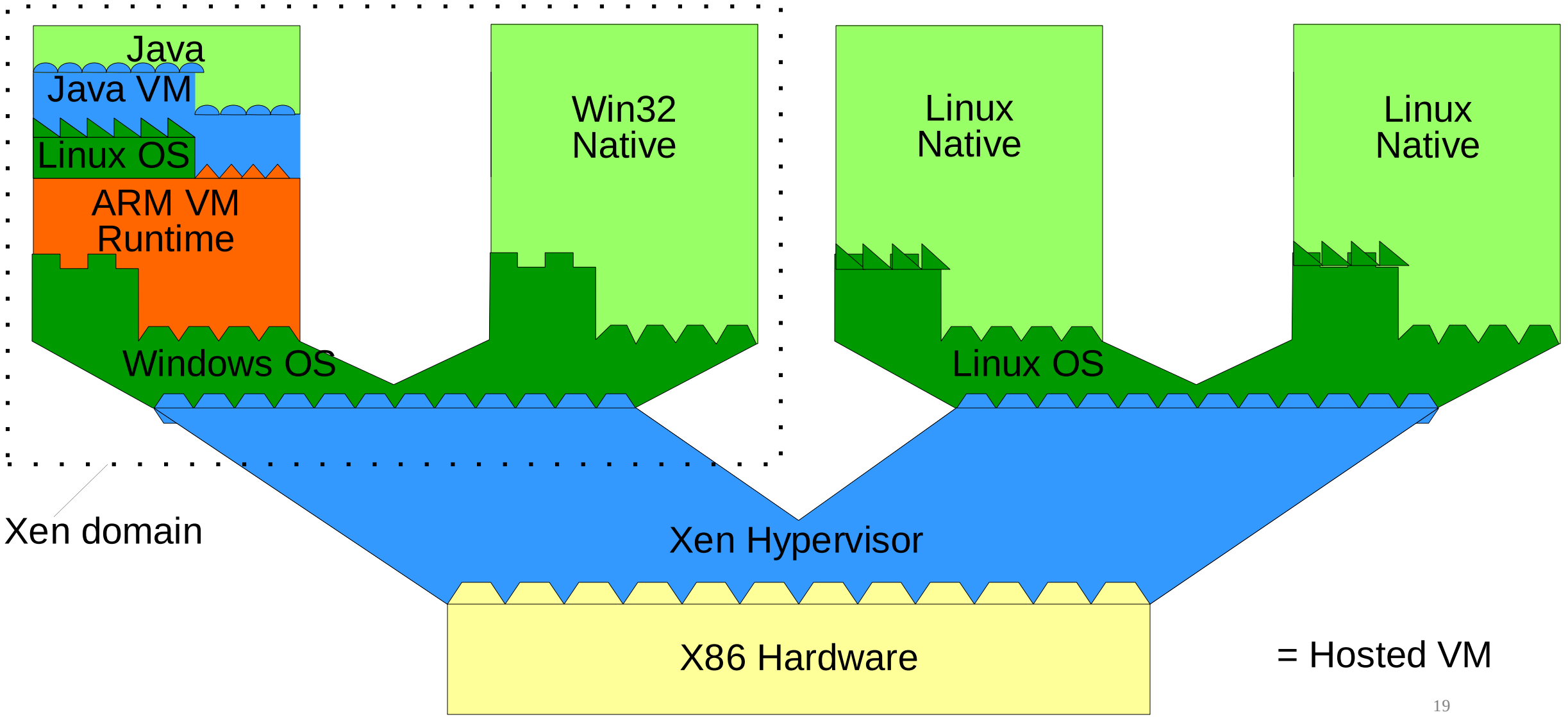
Process VM / Diff ISA

Example: Android Emulation

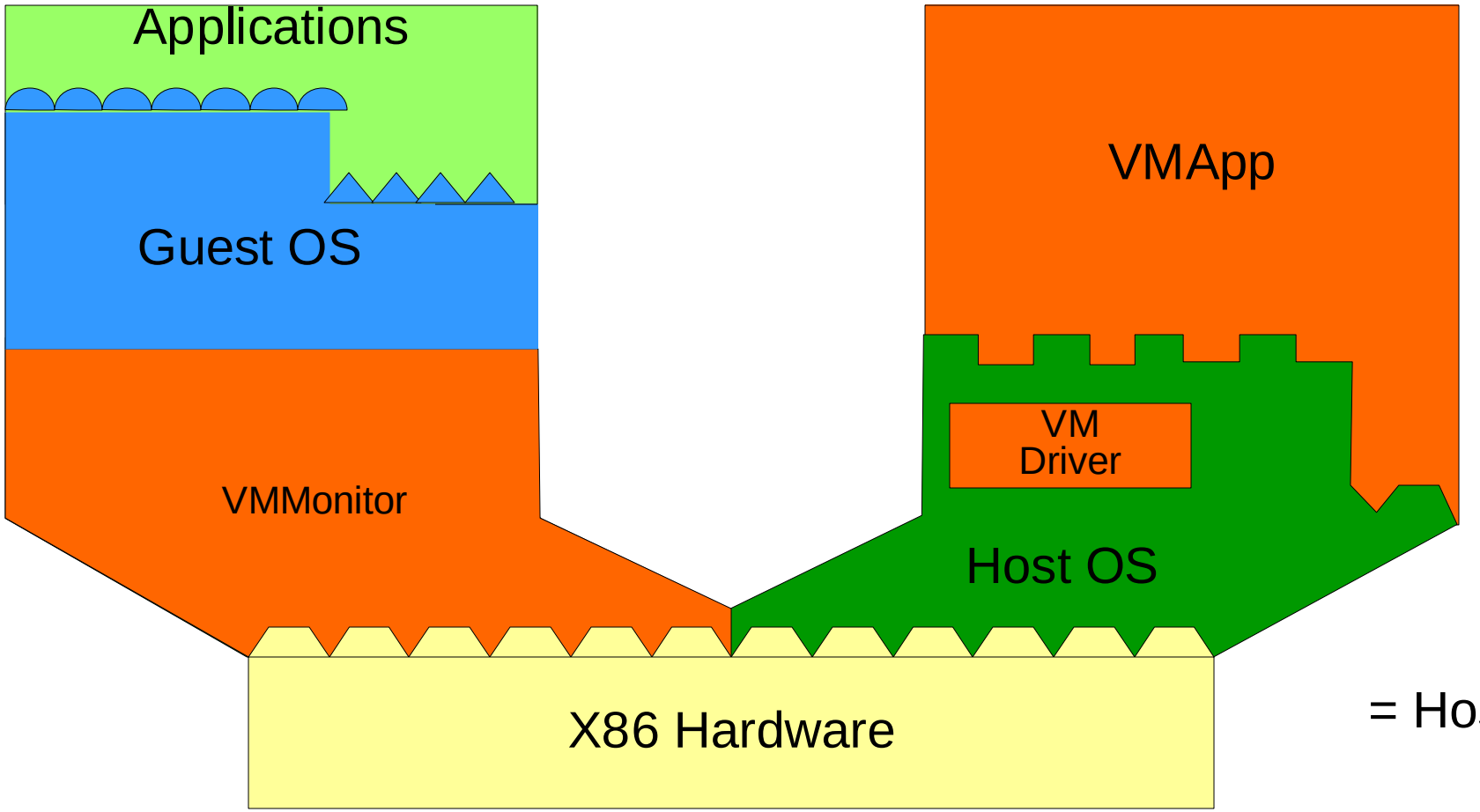


System VM / Diff ISA

Example: Android Emulation on Xen



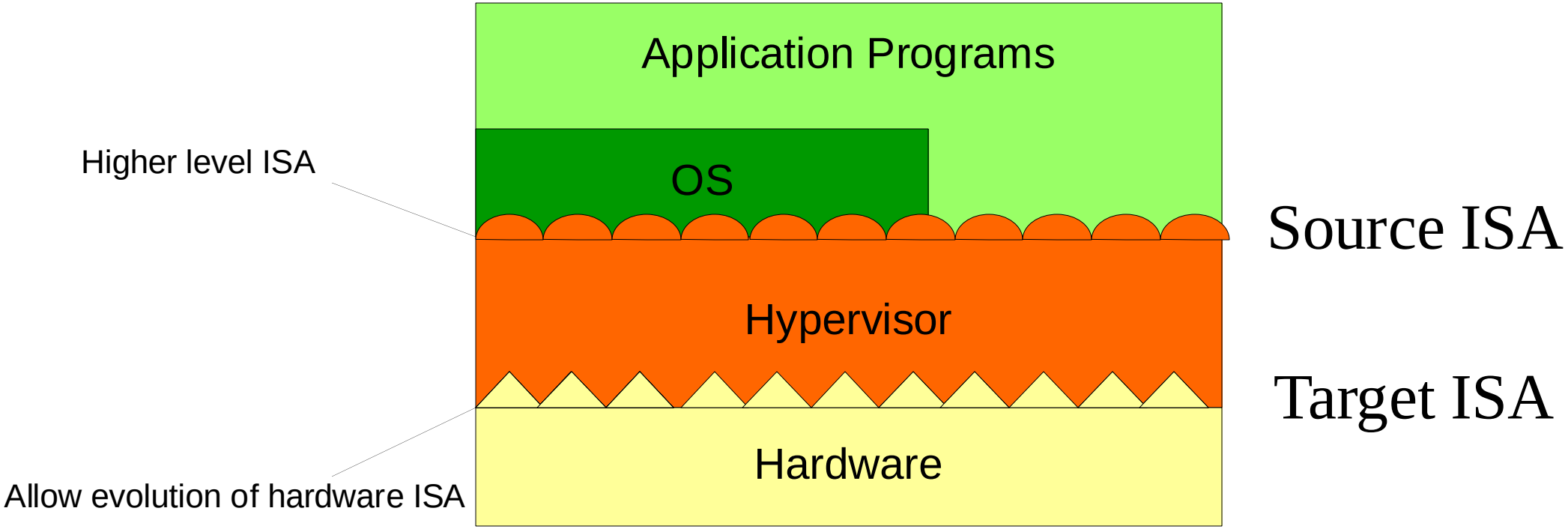
Example: VMWare



= Hosted VM

System VM / Same ISA

Example: AS/400



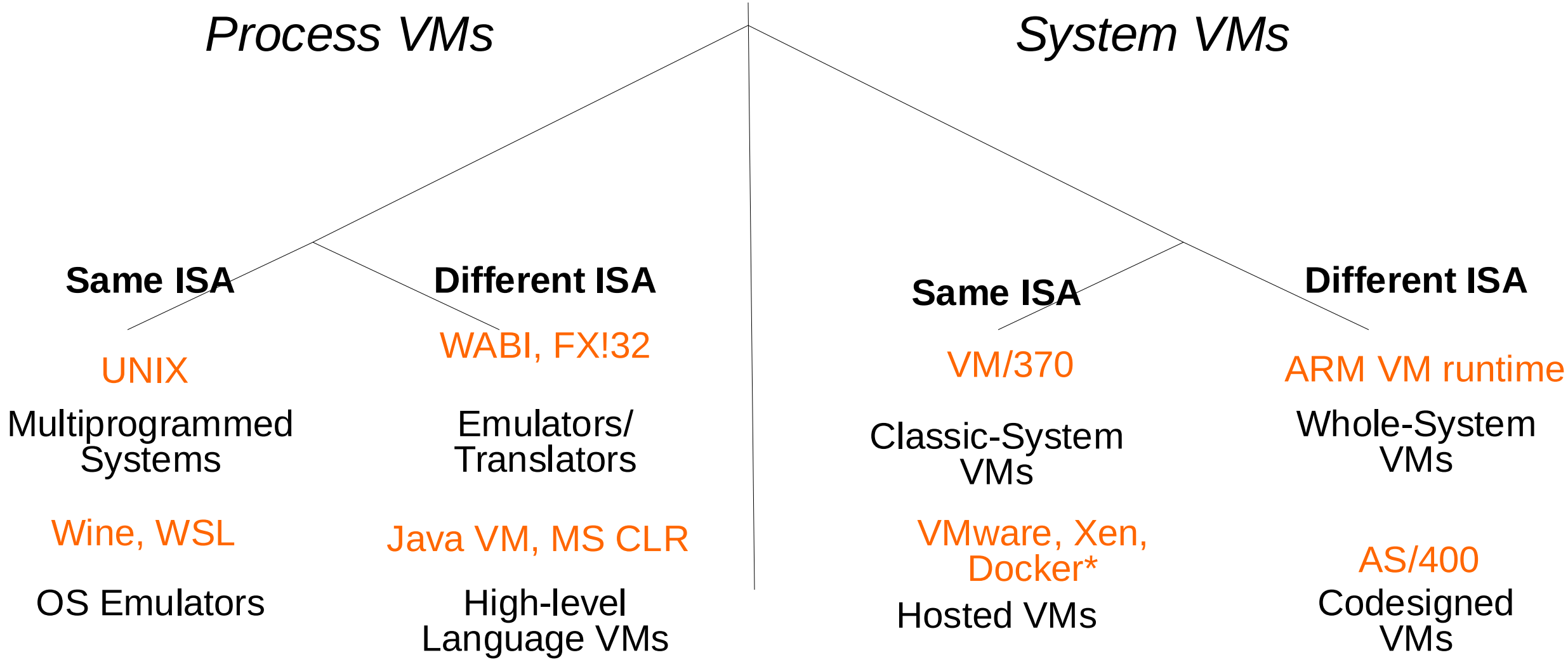
Source ISA

Target ISA

= Co-designed VM

System VM / Diff ISA

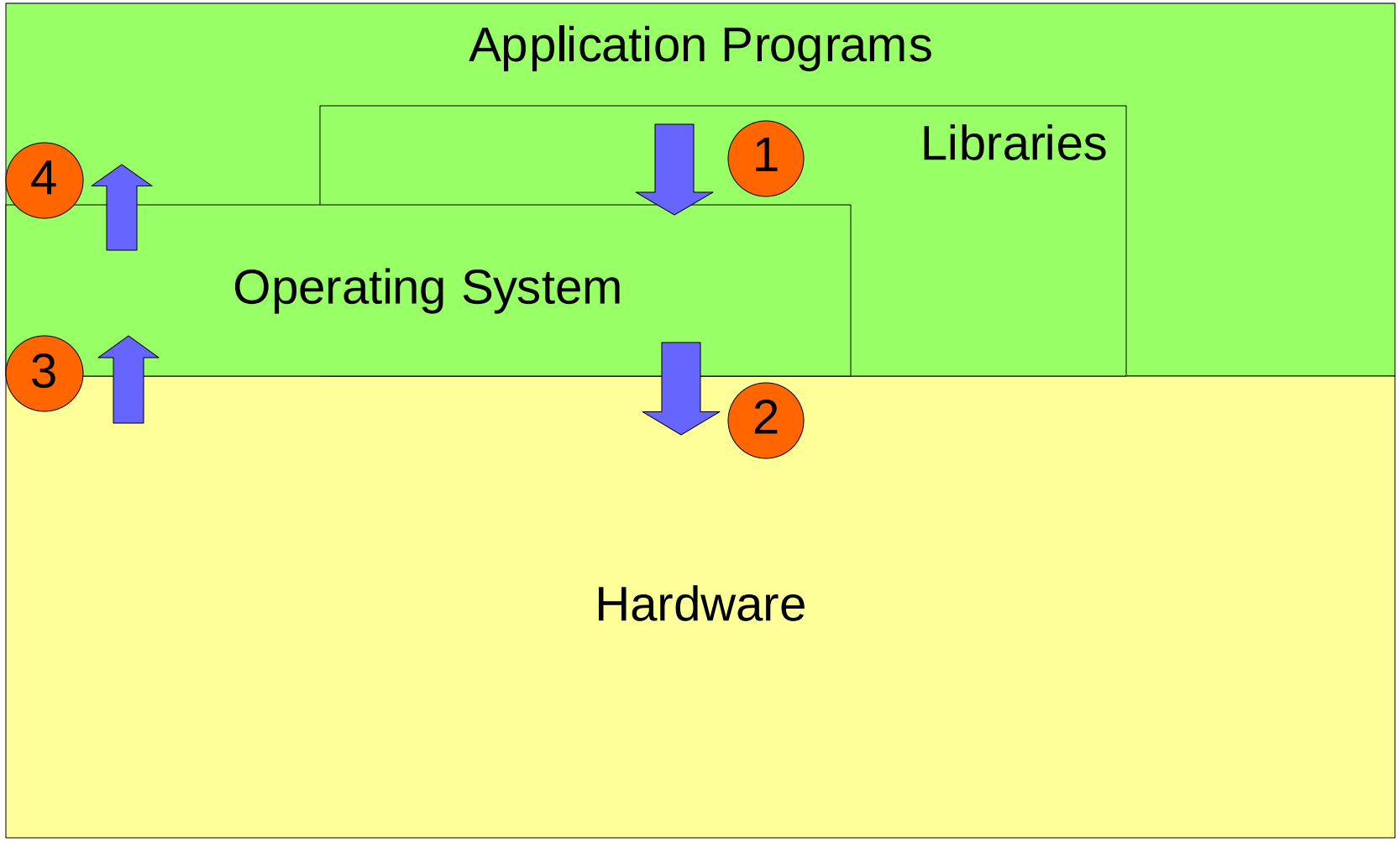
Taxonomy Examples



Recap

- What is the user part of an ISA?
- What is the system part of an ISA?
- What functionality do they provide?

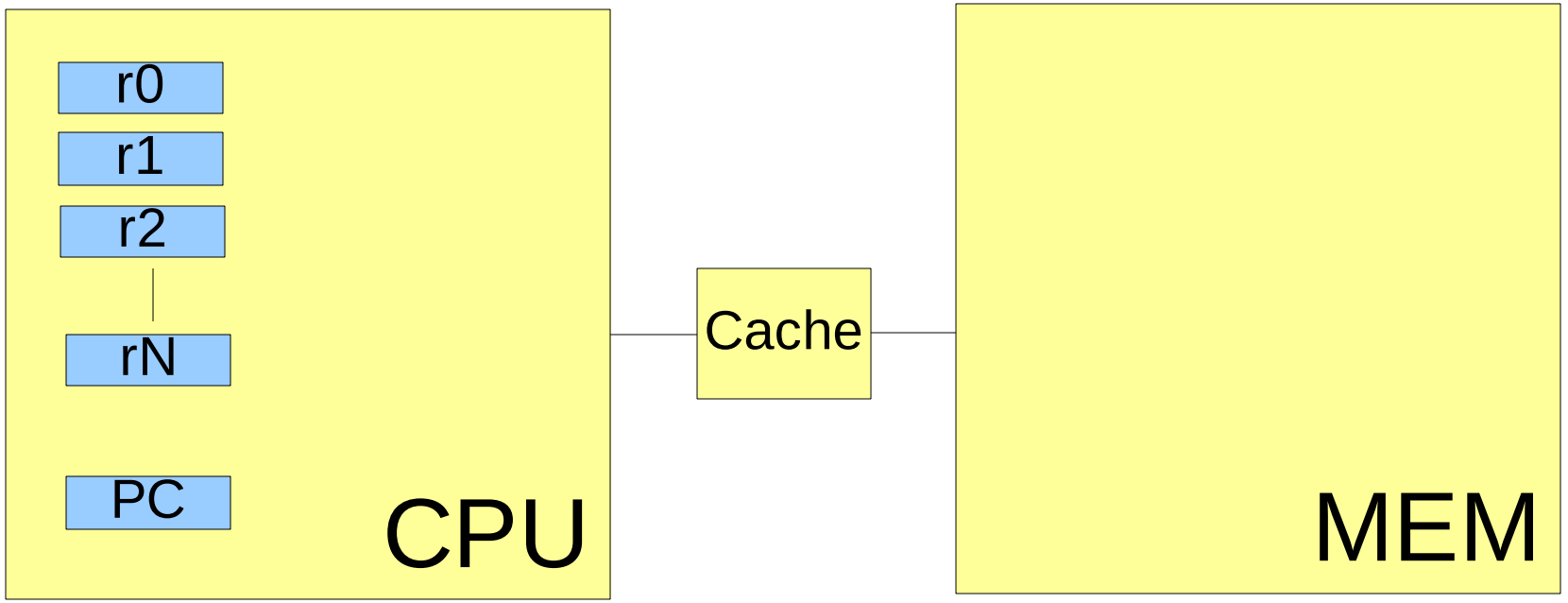
Recap



Arrows?
What runs in User / Kernel Mode?

Implementing Virtual Machines with Different ISAs

Recall: Registers + Memory



VM implementation: Emulation

- Emulation = implement interface of one system on another system with a different interface
- Example: x86 instruction

```
-addl %edx,4(%eax)
```

- Emulated via PowerPC instructions:

```
-lwz  r4,0(r1)
-addi r5,r4,4
-lwzx r5,r2,r5
-lwz  r4,12(r1)
-add  r5,r4,r5
-stw  r5,12(r1)
```

Emulation Performance + Methods

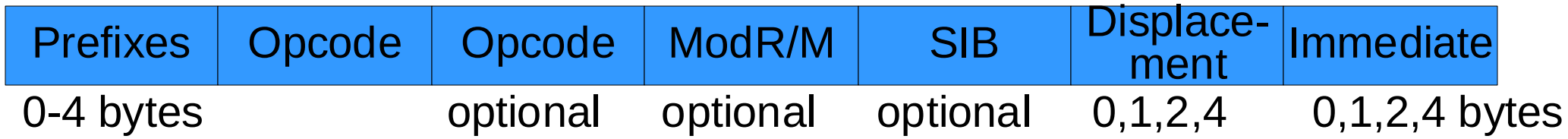
- Can be slow because of mapping Source to Target!
- Range of emulation methods:
  - Interpretation
  - Binary translation
- Interpretation:
  - Decode a **single** source instruction and execute using target instructions
- Binary translation:
  - Translate a **block** of source instructions once and reuse

Interpretation

- Source instruction is a series of bytes
- Different formats
  - RISC: clean and simple
  - CISC: complex with legacy
  - Non-hardware: Java bytecodes
- Complexity of format influences interpretation performance!

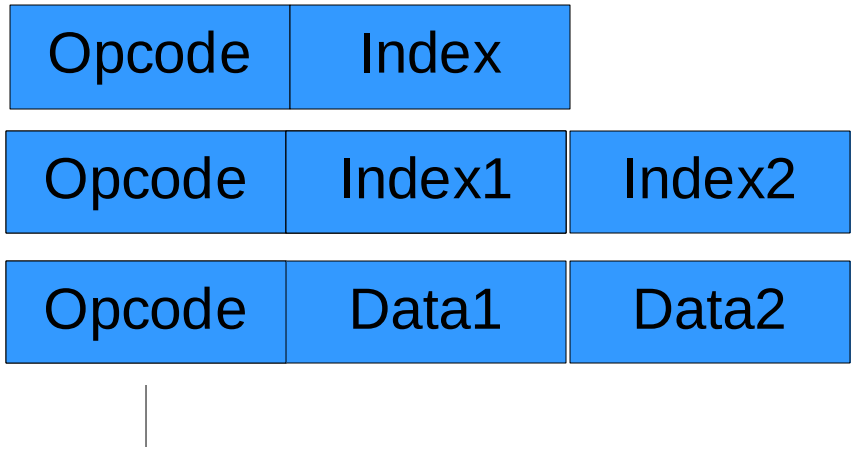
Example Formats

- x86:



- Java:

Software developer's manual: 3796 pages!



Java VM Specification: 604 pages

Binary Translation

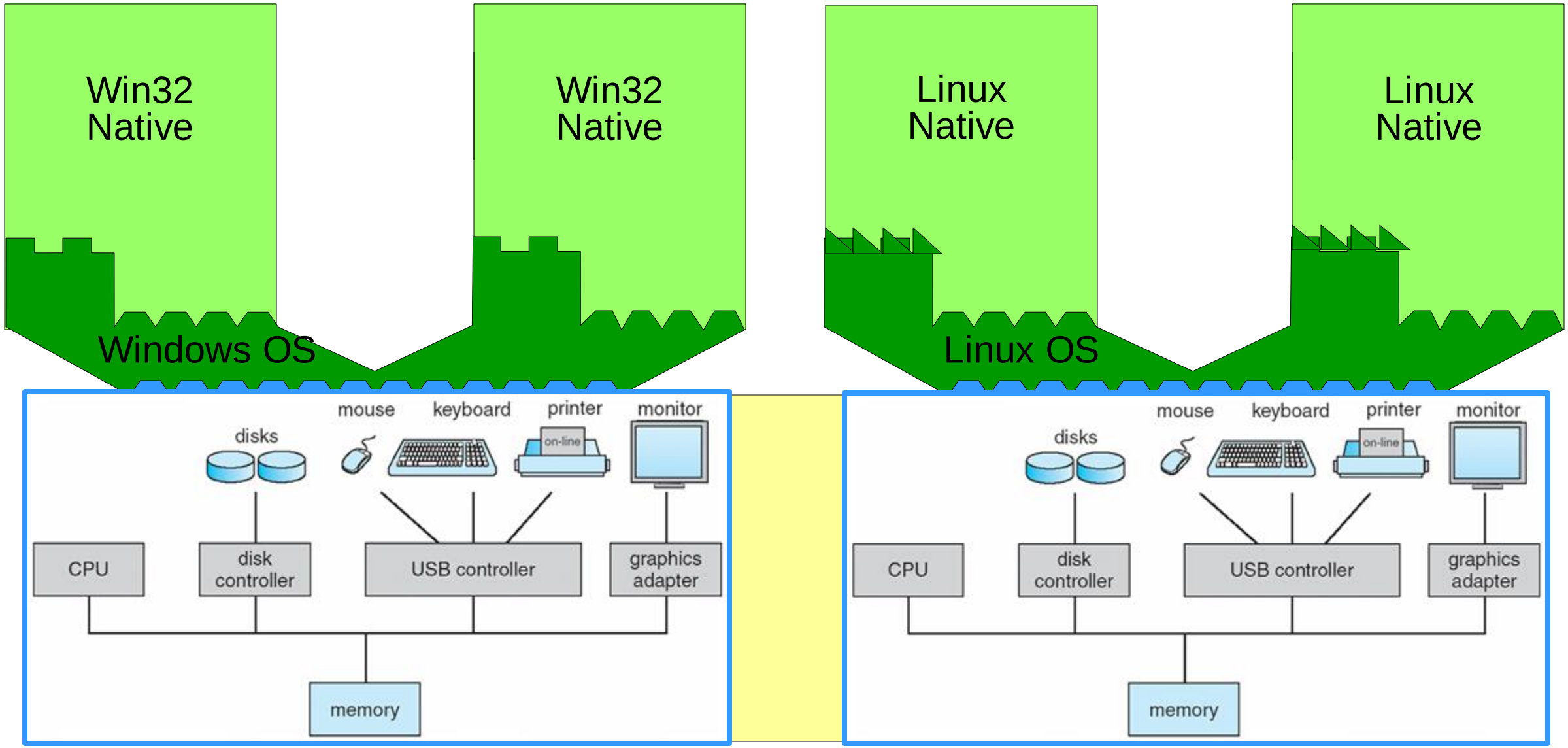
- Per-instruction interpretation slow
  - Especially when complex
- Alternative:
  - Translate blocks of source instructions once
  - Reuse
- cf. Just-in-Time compilers
- Hard

Performance Tradeoff

- E(n) = time needed to execute an instruction n times
- Formula: E(n) = S + n*T
- S = startup time
- T = time required per emulation of the instruction

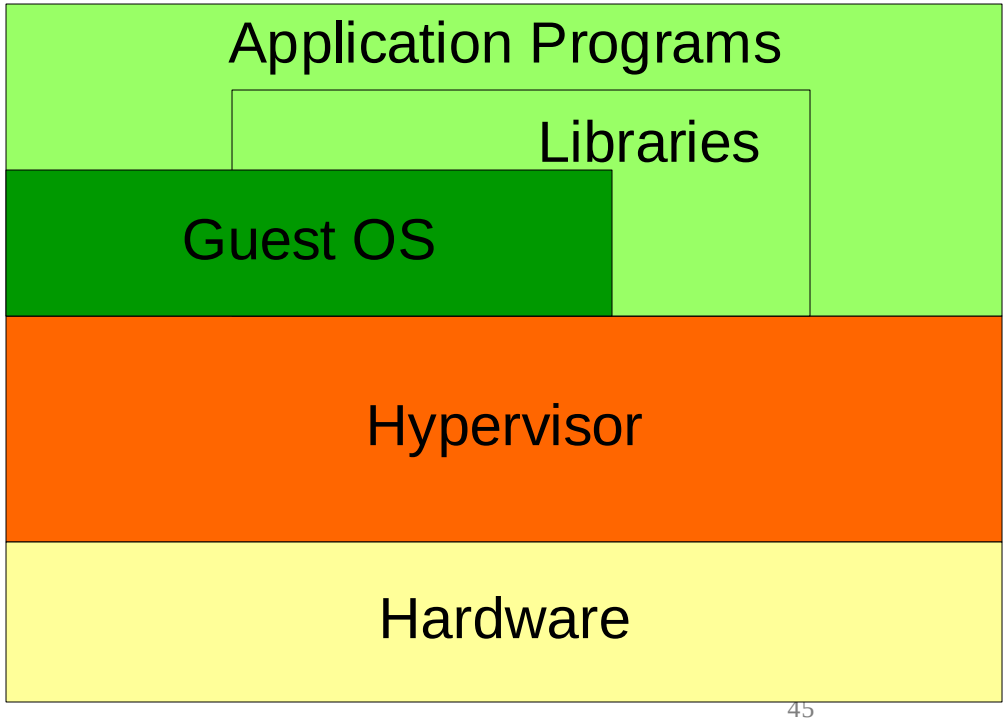
- Interpretation:
  - S low, T high
- Binary translation
  - S high, T low

Simplistic: Each VM sees own hardware



Same ISA VMs

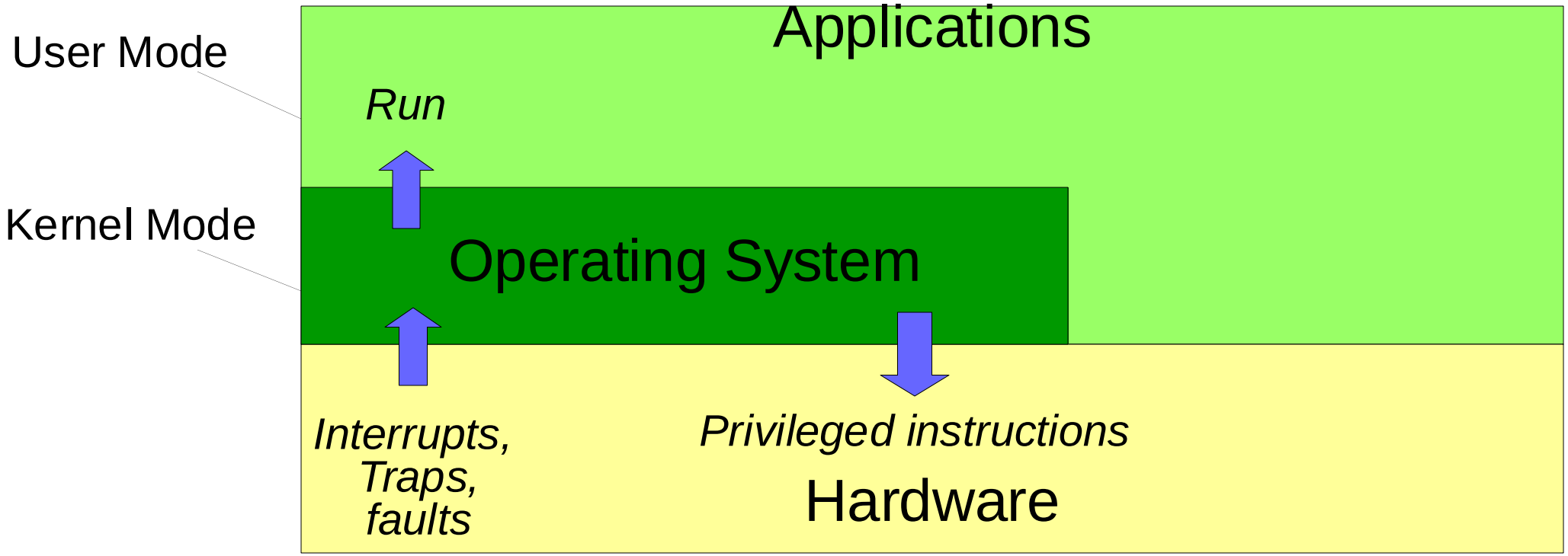
- Emulation needed for different ISA VMs
- For same ISA: Theoretically, source instructions can be executed directly on target
- Fastest
- Does this work for all
- instructions?



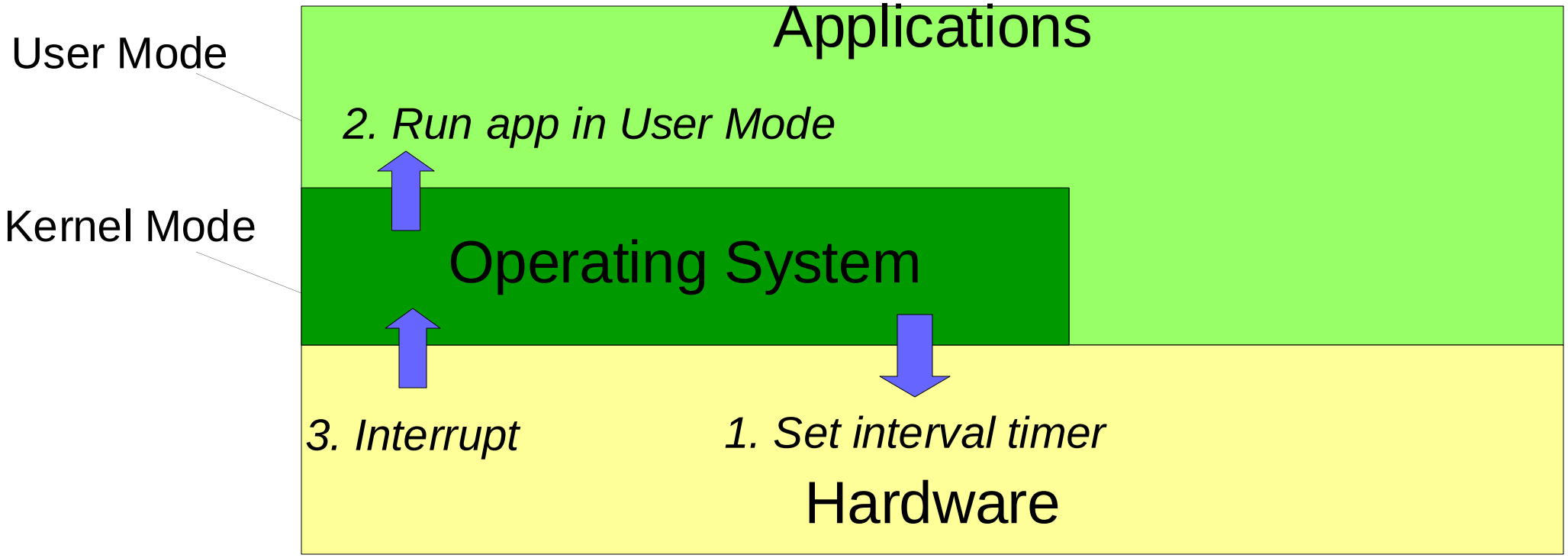
Same ISA VMs (cont'd)

- No: Privileged / System ISA instructions need to be controlled
- Why?
- How?
  - Guest OS runs in CPU User Mode
  - System ISA instructions called in User Mode activate Kernel Mode i.e., cause a Trap
- Hypervisor in Kernel Mode then emulates privileged instruction

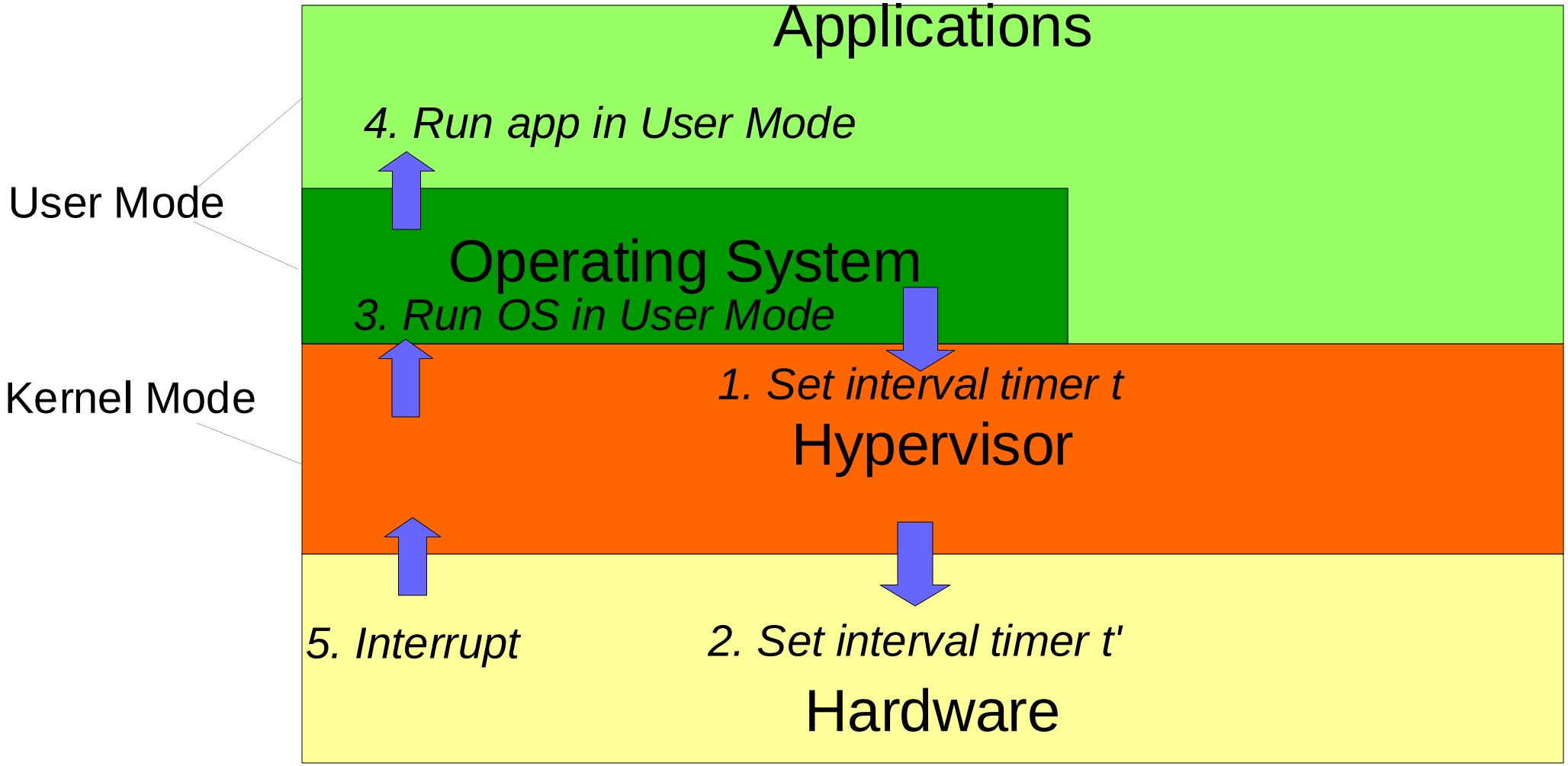
Example: Normal App Scheduling



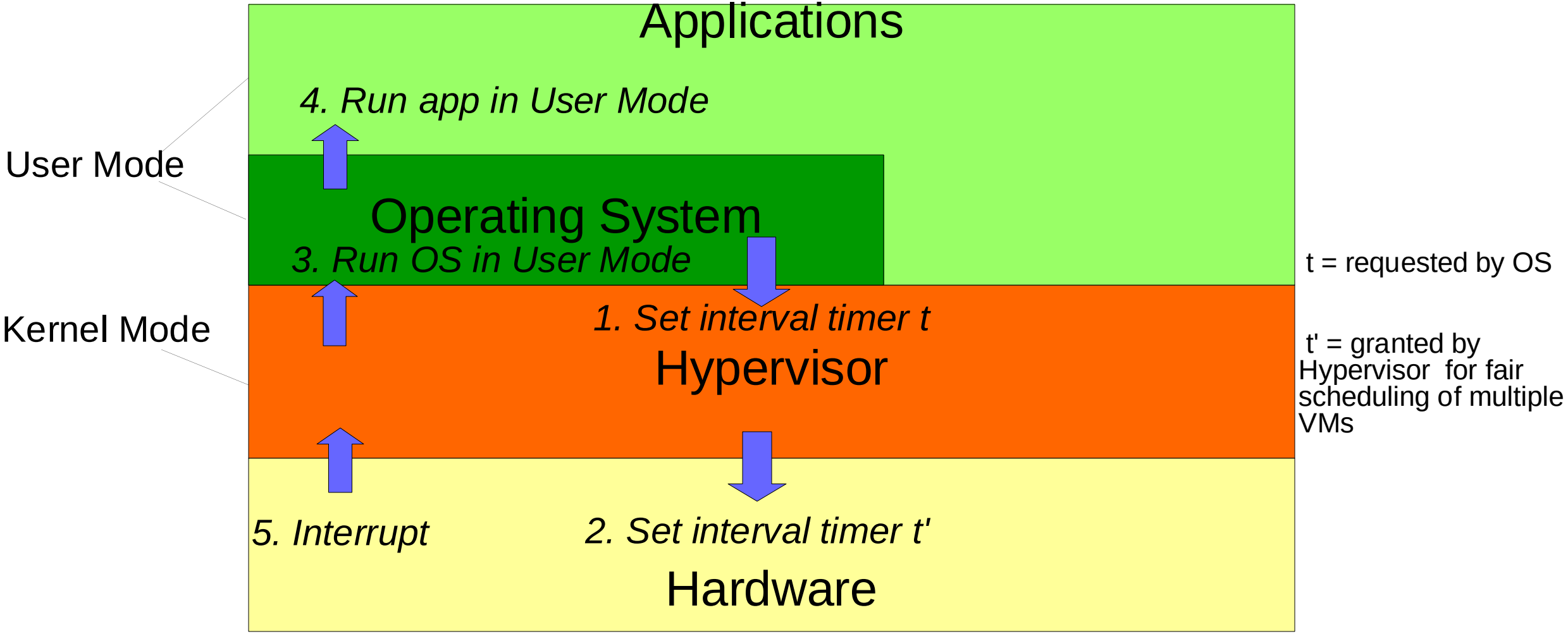
Example: App Scheduling



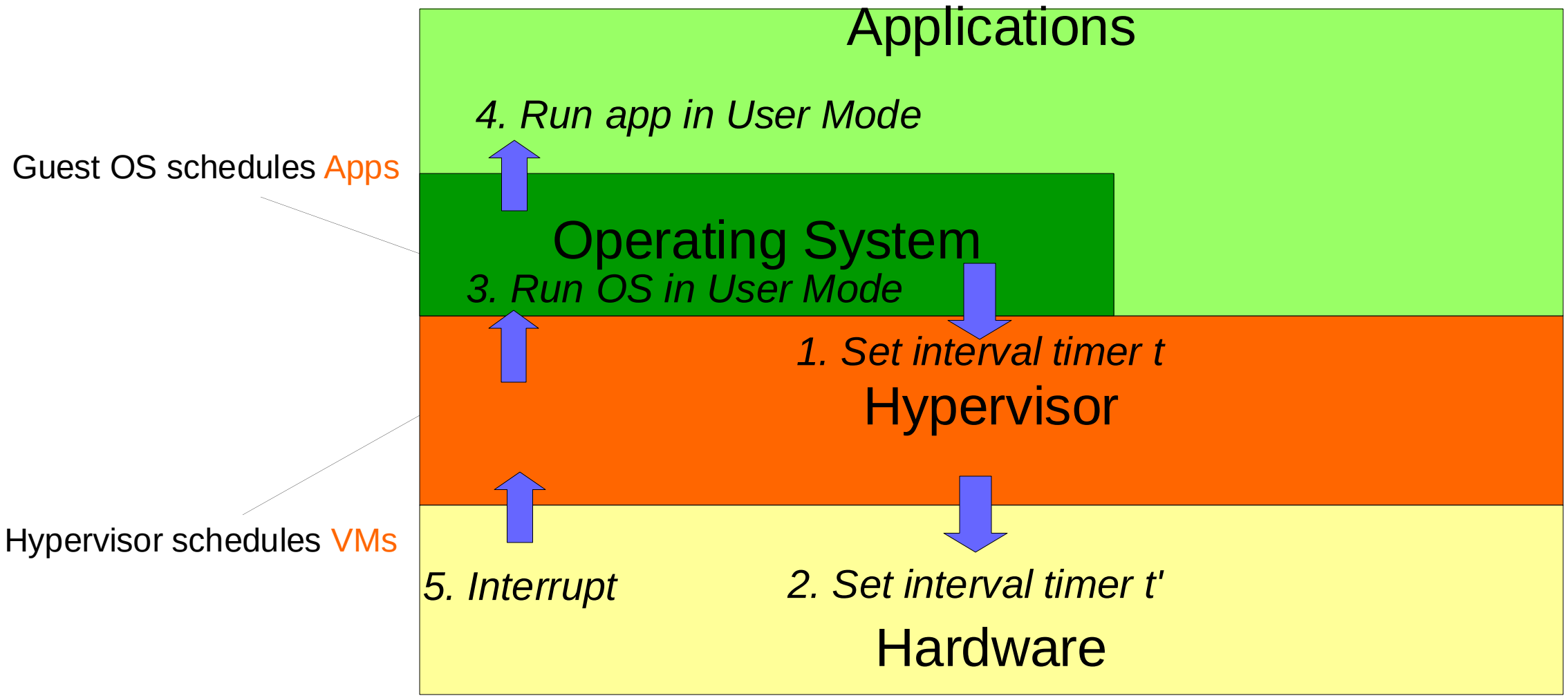
Example: App Scheduling with VMs



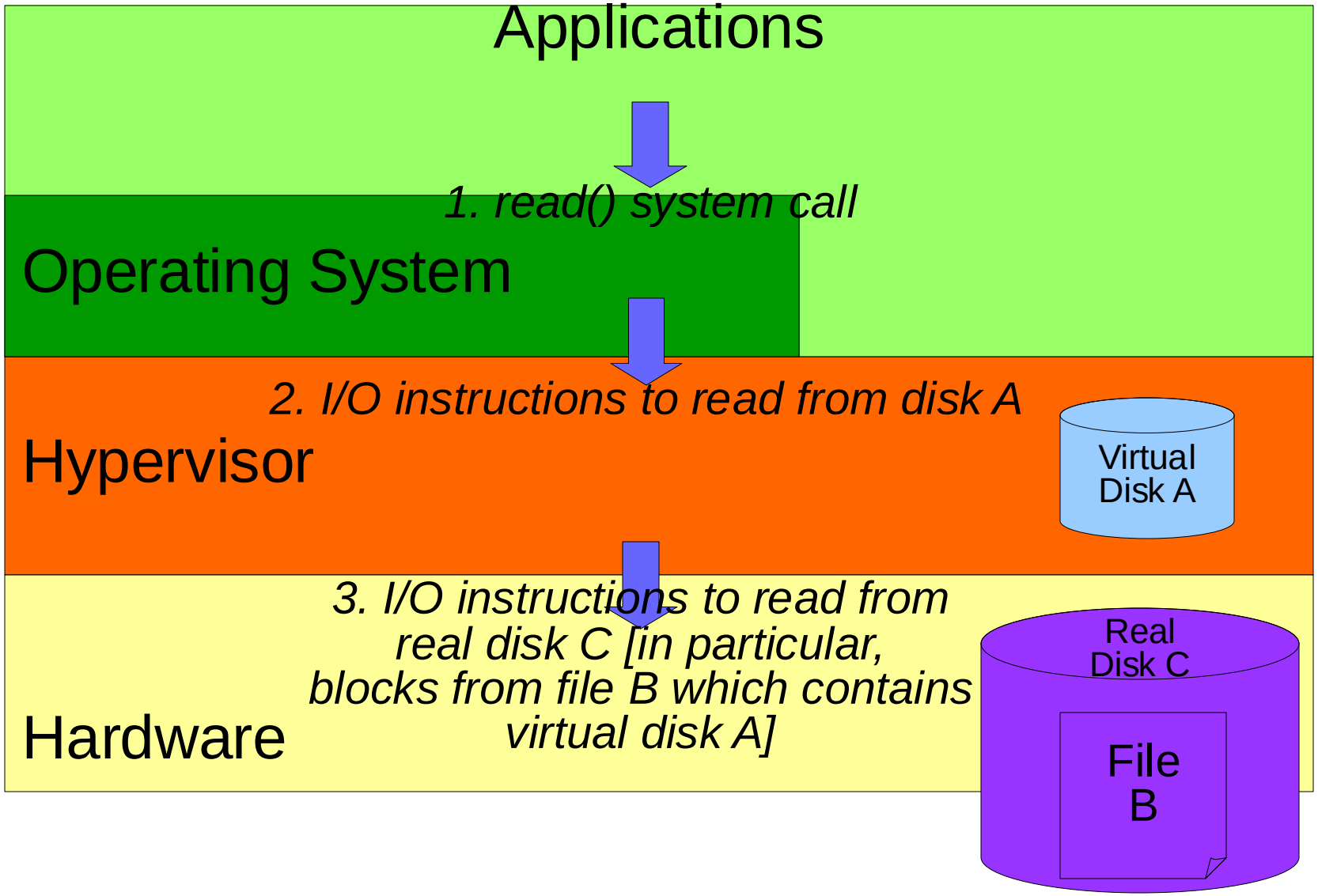
Example: App Scheduling with VMs



Example: App Scheduling with VMs

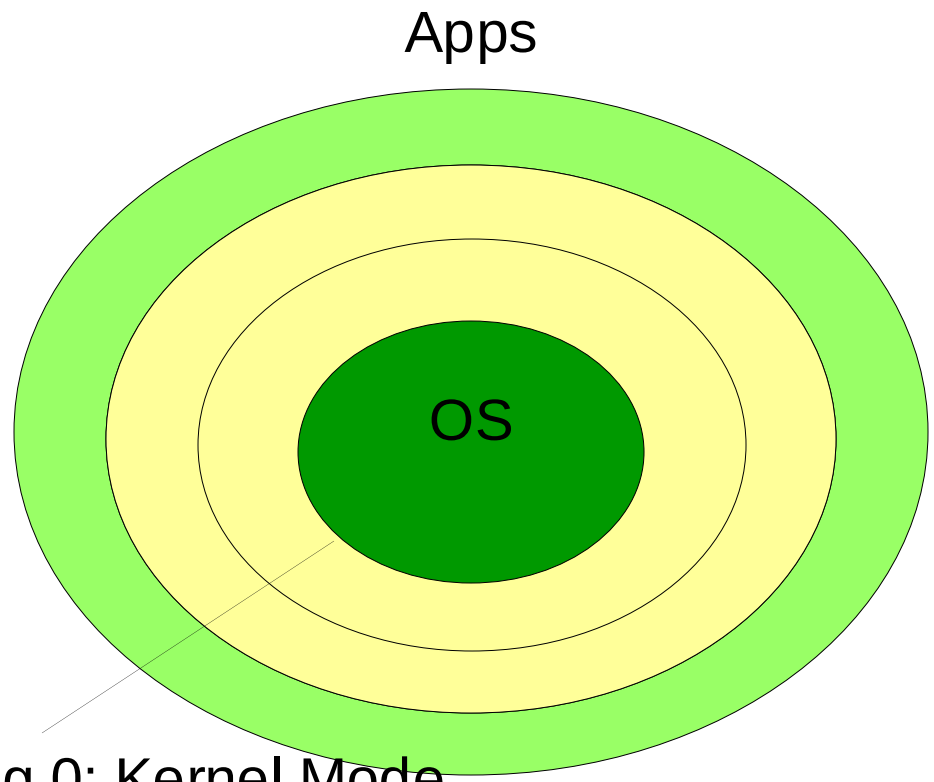


Example: Reading from disk

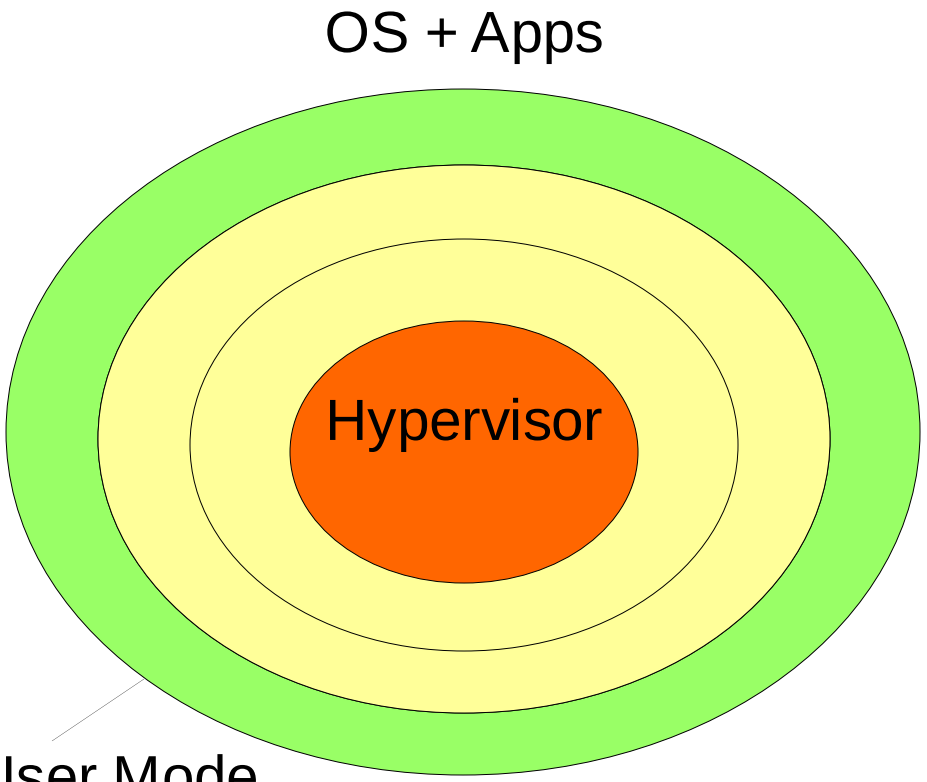


x86 Same ISA Problems (1/3)

Normal:



VM: OS no longer in kernel mode!



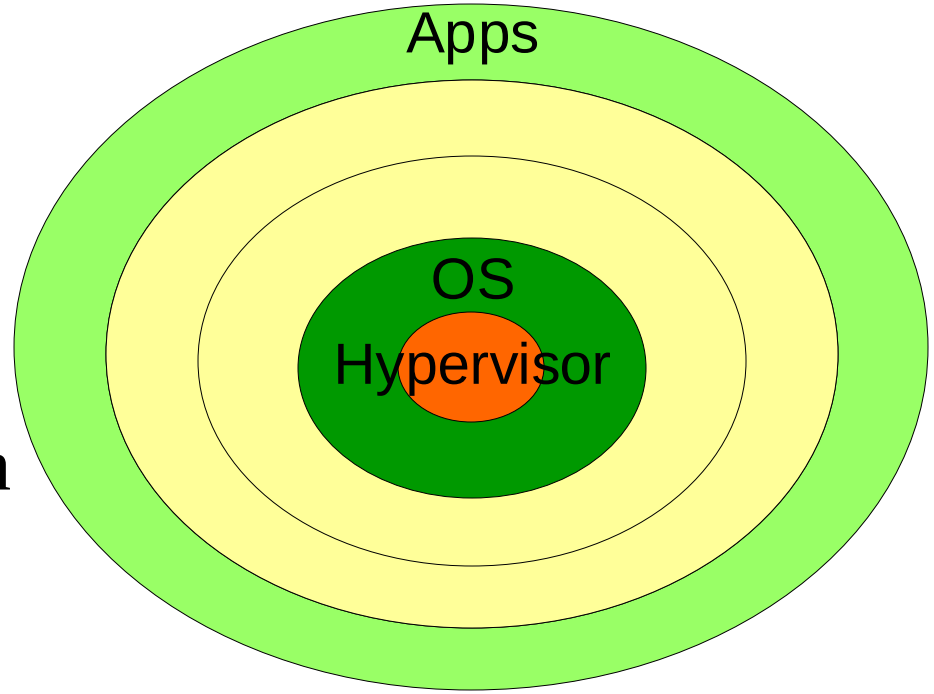
x86 Same ISA Problems (2/3)

- In x86, not all system instructions in User Mode activate Kernel Mode!
- E.g., RDTSC (Read Time-Stamp Counter): Reads the processor's time-stamp counter.

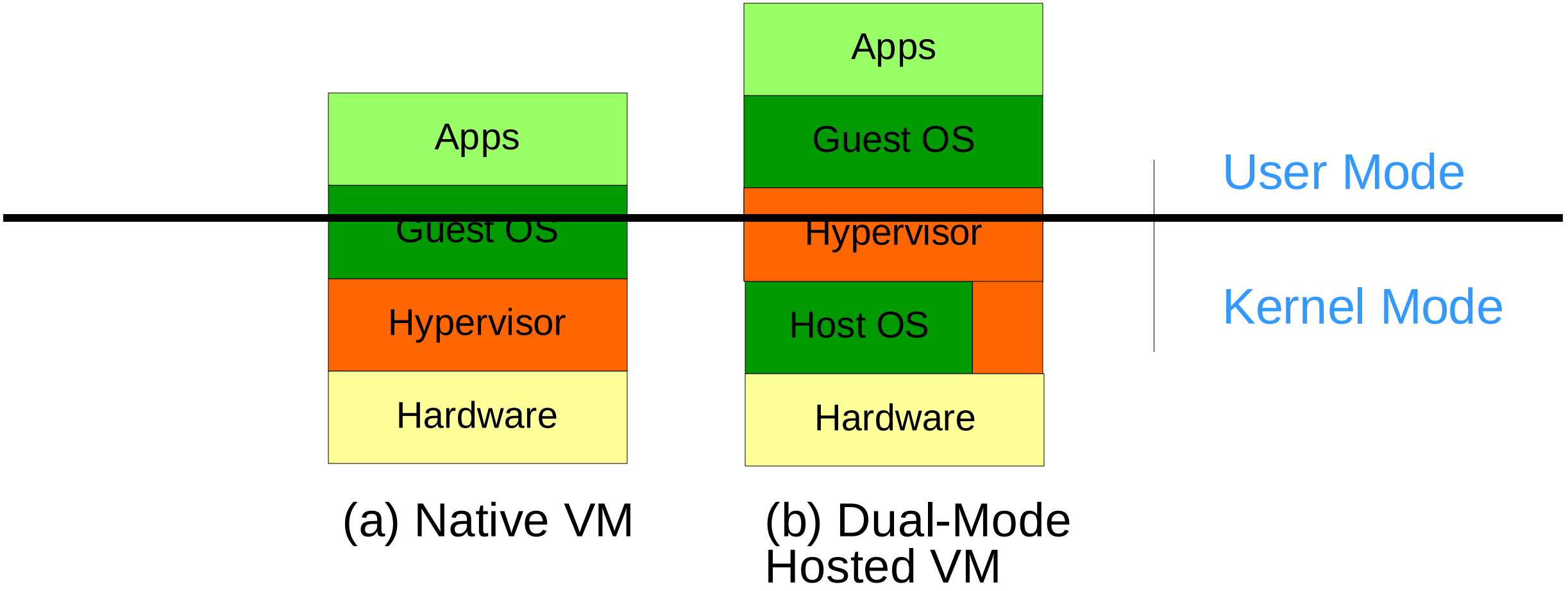
  CPUID: Provides information about the CPU.
- → When Guest OS runs in User Mode, not all system instruction calls observed by Hypervisor
- Old solution: patch all binary code!
  - Replace these critical instructions with explicit traps to Kernel Mode

x86 Same ISA Problems (3/3)

- New solution: Intel VT-x
- Allows Guest OS to run in Kernel Mode (Ring 0)
- Shared resources still controlled by Hypervisor
- Using extra mode: VMX
  - VMX Root for Hypervisor
  - VMX Non-root for Guest OS
- "Ring -1"
- `vmcall` for fast OS→Hypervisor communication
- Also hardware support for VM context switch



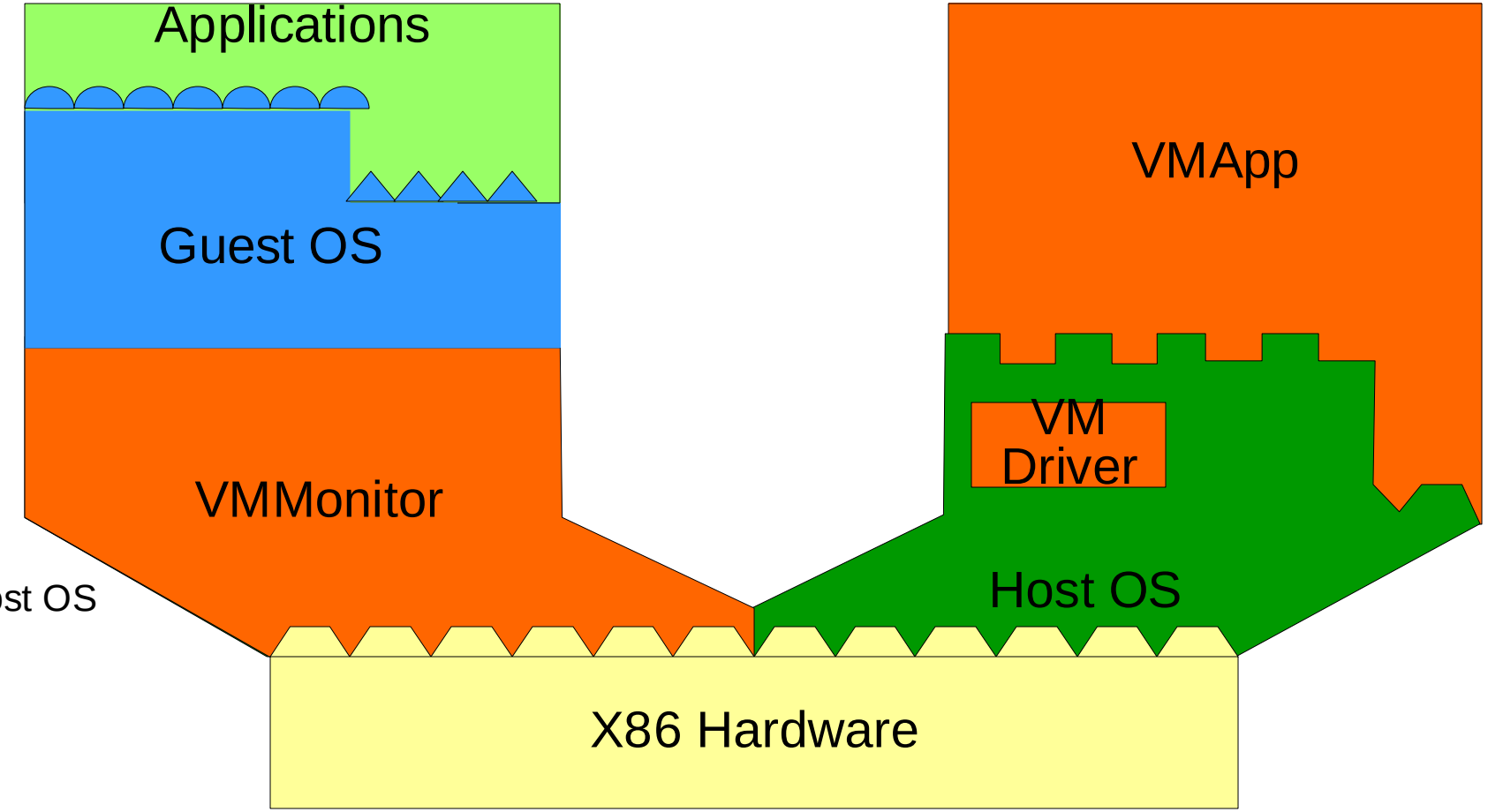
Native and Hosted VMs



VMWare Workstation

- Install on top of existing host OS
  - Easy to use
  - Can use myriad of device drivers available in host OS
- Different from ESX:
  - More like true Hypervisor that implements all drivers itself.

VMWare Architecture



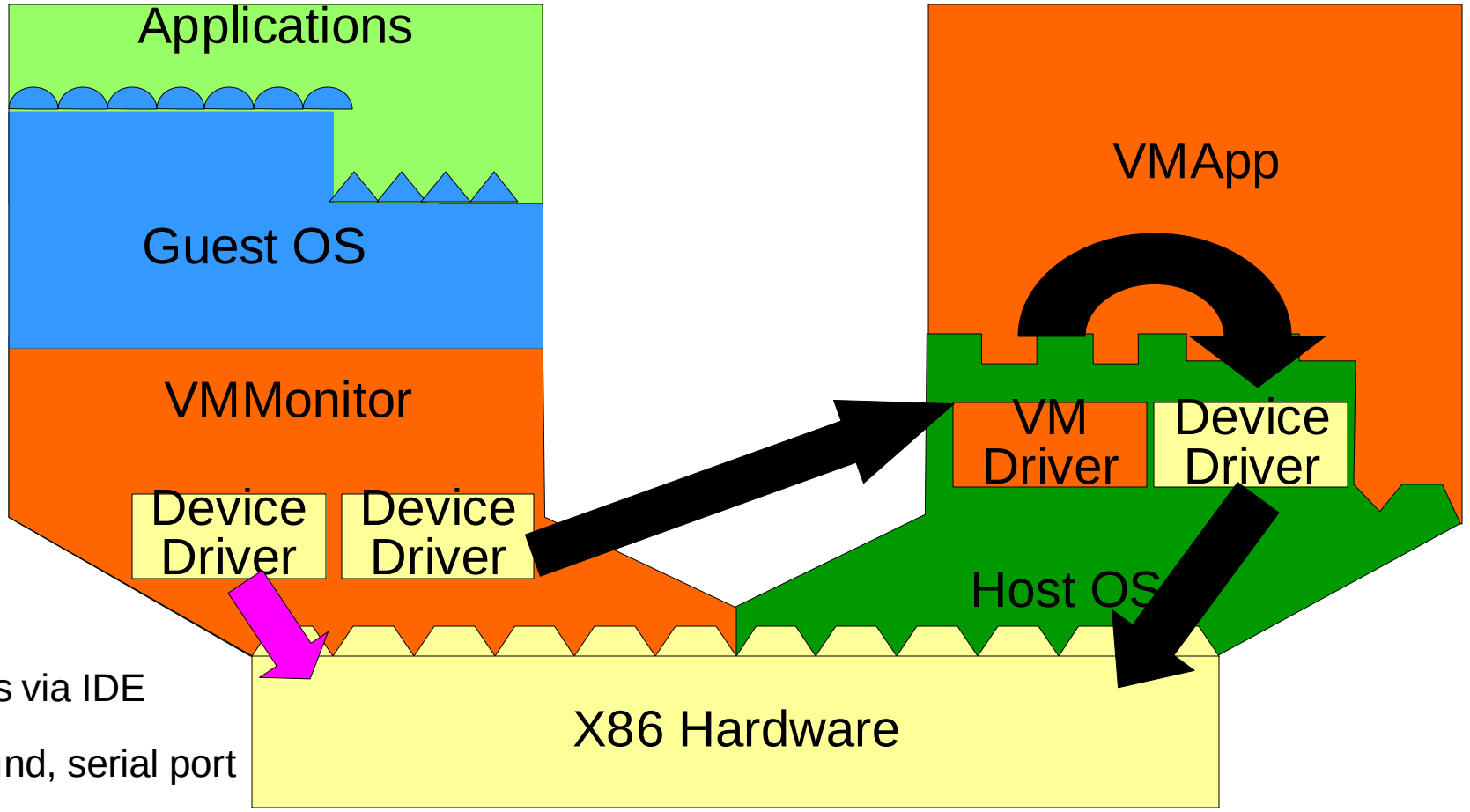
Hypervisor in Kernel Mode, alongside Host OS

User Process for translating Hypervisor requests into system calls to host OS

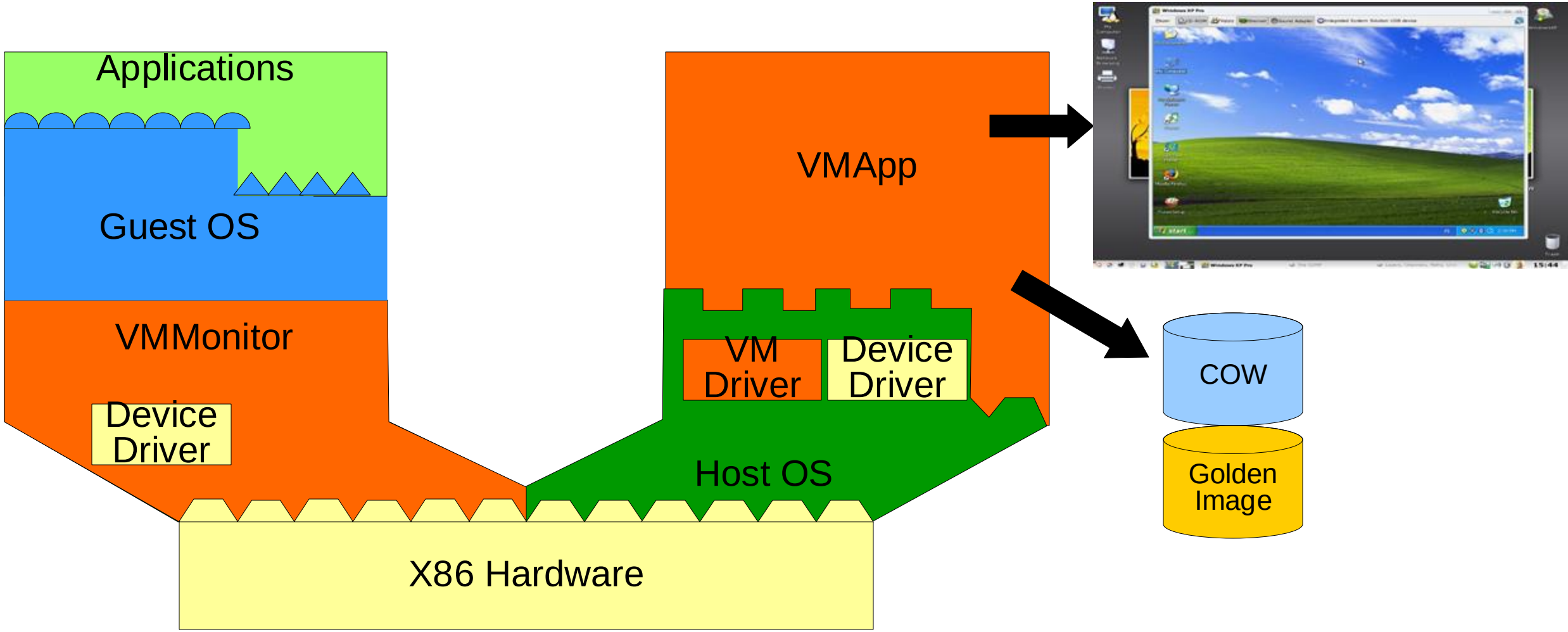
VMWare Workstation

- Adds 3 components
- VMMonitor
  - Hypervisor in Kernel Mode, alongside Host OS
- VMApp
  - User Process for translating Hypervisor requests into system calls to host OS
- VMDriver
  - Extension of the host OS
  - Support switching between Host OS and Hypervisor
  - Enable VMMonitor↔VMApp communication

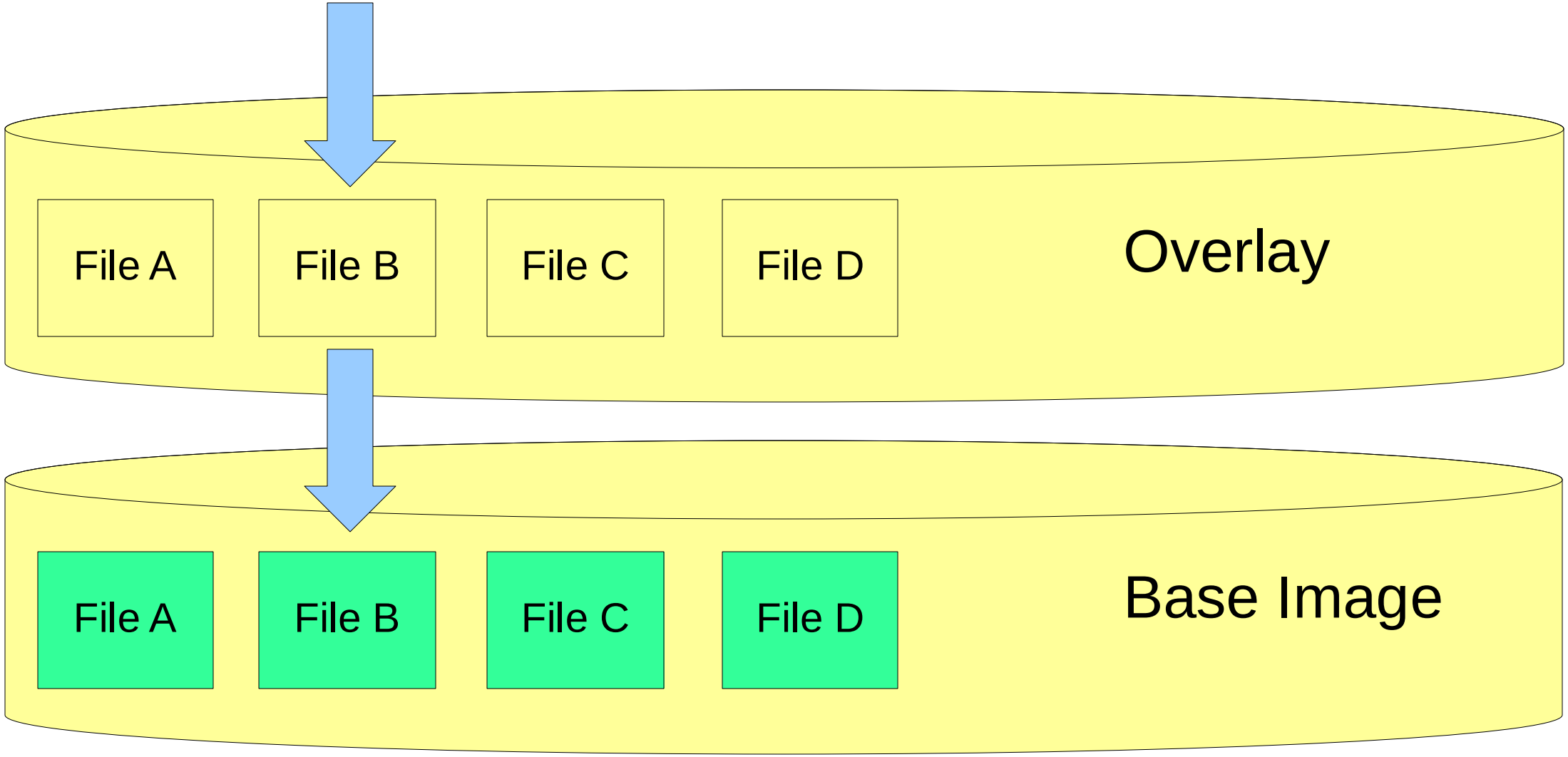
VMWare I/O


Applications
Guest OS
VMMonitor
Device Driver
Device Driver
VMApp
VM Driver
Device Driver
Host OS
X86 Hardware
Direct support, e.g. disk access via IDE
Use host support, e.g. CD, sound, serial port

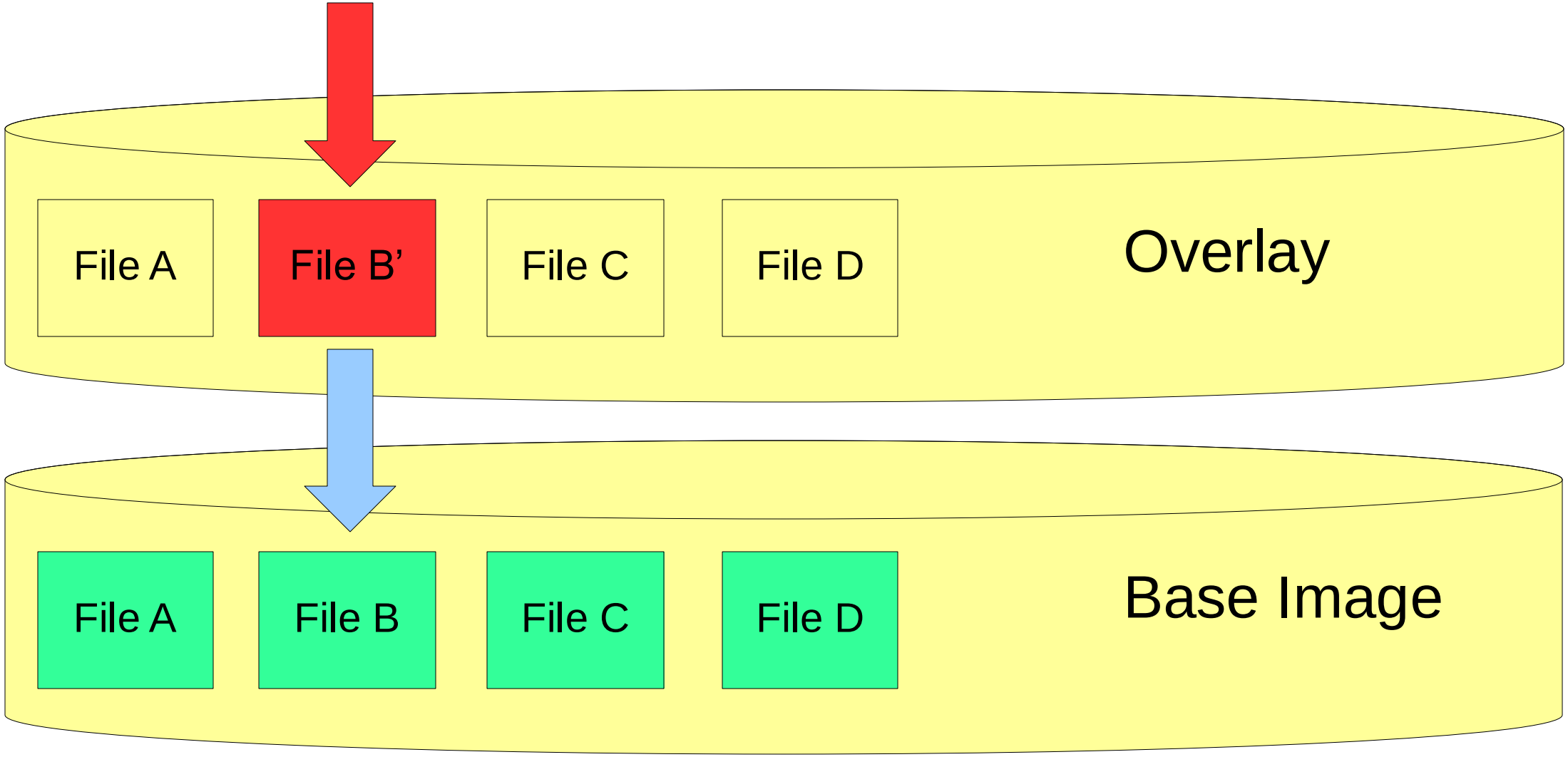
VMWare: New Capabilities



Copy-on-Write



Copy-on-Write



QUIZ- Week 2a