CS 3214
Operating Systems

Virtualization

Godmar Back

Definitions for “virtual machine”

- Term is somewhat ill-defined, generally
  - A machine that’s implemented in software, rather than hardware
  - A self-contained environment that acts like a computer
  - An abstract specification for a computing device (instruction set, etc.)
- Common distinction:
  - (language-based) virtual machines
    - Instruction set usually does not resemble any existing architecture
    - Java VM, .Net CLR, many others
  - virtual machine monitors (VMM)
    - instruction set fully or partially taken from a real architecture

Example Java Bytecode

```
public class h
{
    public static void main(String []av) {
        System.out.println("Hello, World");
    }
}
```

```
Compiled from "h.java"
public class h {
  public h();
    Code:
       0: aload_0
       1: invokespecial #8                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #13                 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #19                 // String Hello, World
       5: invokevirtual #21                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

- Not the kind of VM this discussion is about!

Use of Virtual Machines

- Test applications
- Program / debug OS; fault injection
- Bundle applications + OS (“Virtual appliances”)
- Monitor for intrusions
- Resource sharing/hosting `cloud computing’
- Migration
- Replication
- Simulate networks

History of virtual machines

"Virtual machines have finally arrived. Dismissed for a number of years as merely academic curiosities, they are now seen as cost-effective techniques for organizing computer systems resources to provide extraordinary system flexibility and support for certain unique applications."

- See Goldberg [1972], [1974]

Some of these advantages include support of the following ivities concurrently with production uses of the system:

- improving and testing the operating system software[2]
- running hardware diagnostic check-out software[1]
- running different operating systems or versions of an operating system[3,4]
- running with a virtual configuration which is different from the real system, e.g., more memory or processors, different I/O devices[5]
- measuring operating systems[6,7]
- adding hardware enhancements to a configuration without requiring a recoding of the existing operating system(s)[3]

Popek/Goldberg Requirements

- (1974)
  - Equivalence/Fidelity
    - Program should exhibit same behavior
  - Resource control
    - VMM must have full control of resources
  - Efficiency
    - Most instructions should execute natively

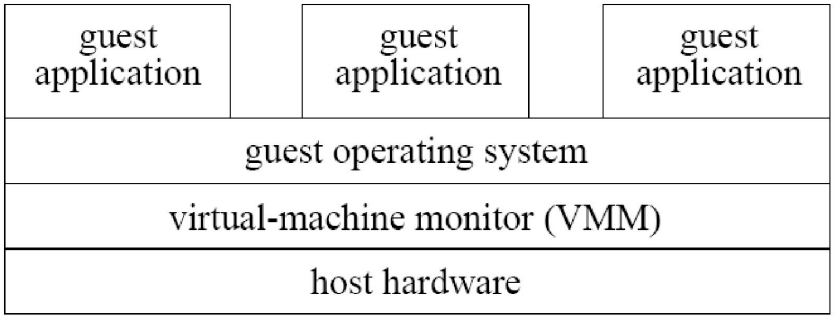
History (cont'd)

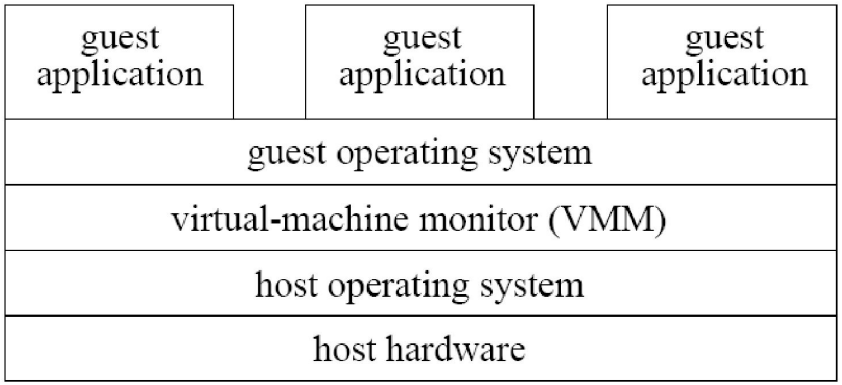
- "Disco" project at Stanford [Bugnion 1997]
  - Created hypervisor to run commodity OS on new "Flash" multiprocessor hardware
  - Based on MIPS
- VMWare was spun off, created VMWare Workstation – first hypervisor for x86
- 2000's
  - Resurgence under Cloud moniker

Virtualizing the CPU

- Basic mode: direct execution
- Requires *Deprivileging*
  - (Code designed to run in supervisor mode will be run in user mode)
- Hardware vs. Software Virtualization
  - Hardware: “trap-and-emulate”
    - Not possible on x86 prior to introduction of Intel/VT & AMD/Pacifica
    - See [Robin 2000]
  - Software:
    - Either require cooperation of guests to not rely on traps for safe deprivileging
    - Or binary translation to avoid running unmodified guest OS code (note: guest user code is always safe to run!)

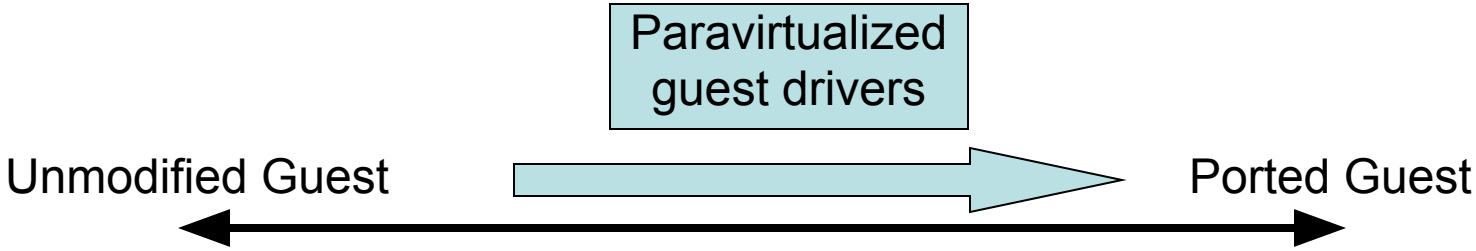
Types of Virtual Machines





- Type I
- Type II

VMM Classification



	Guest OS sees true hardware interface	Guest OS sees (almost) hardware interface, has some awareness of virtualization	Guest OS sees virtualized hardware interface
Hypervisor runs directly on host hardware	VMware ESX MS Virtual Server	Xen Windows 7 (HyperV), KVM	
Hypervisor runs on host OS	qemu, VMware Workstation, VMware GSX, VirtualBox		UML

Type I

Type II

Kernel Support for VMM: skas3, UMLinux, vmware.ko, KVM

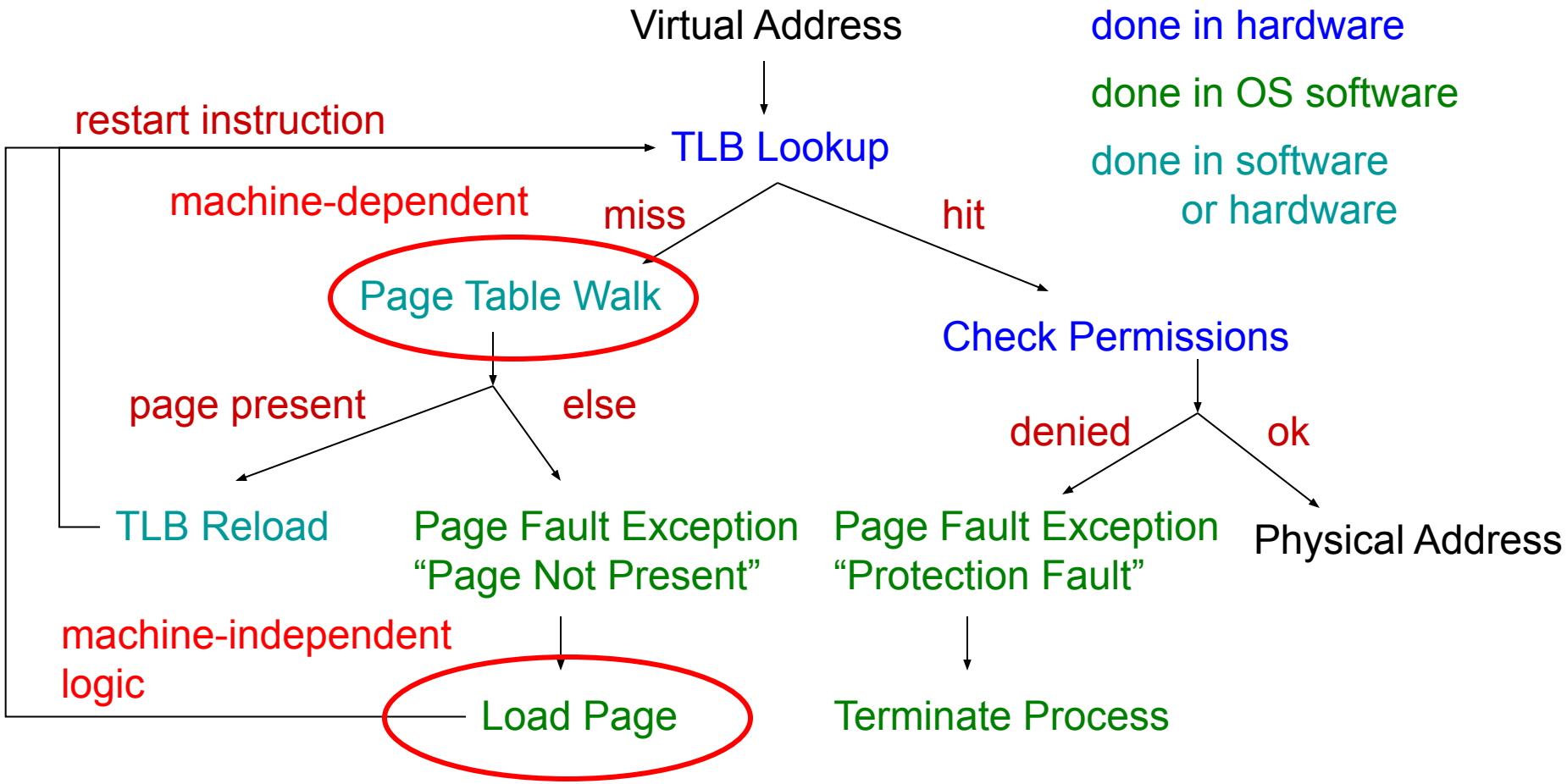
Binary translation vs trap-and-emulate

- Interesting history:
  - IBM/360 (70's) used trap-and-emulate
  - Late 90's: x86 requires binary translation
  - Early 00's: x86 adds hardware for complete trap-and-emulate (*)
  - Late 00's: predominantly hardware-based virtualization + guest accommodation
- (*) Adams [ASPLOS 2006] asked:
  - Is binary translation always slower than trap-and-emulate?
- Surprising result: binary translation beat trap-and-emulate. Why?
  - Binary translation is highly optimized:
    - most instructions are translated as IDENT (identical), preserving most compiler optimizations and only slightly increasing code size
    - binary translation can be *adaptive*: if you know an instruction is going to trap, inline part of all of trap handler. Way cheaper than actually trapping.
  - This trade-off is changing as hardware support gets better, e.g., microcode assist
- See also [PLDI 2012 Agesen]

Virtualizing Memory: MMU

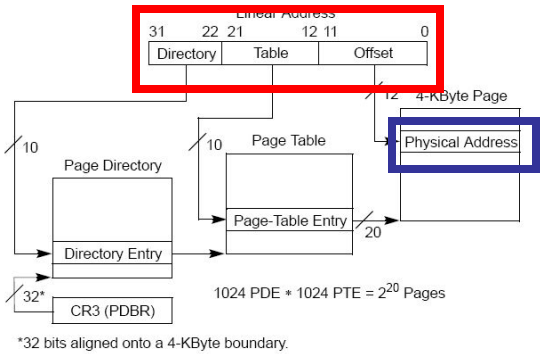
- Guest OS programs page table mapping virtual → physical
  - Hypervisor must map guest's "physical" to machine addresses
- Approaches:
  - Shadow page tables (ESX): hypervisor makes a copy of page table, installs copy in MMU
  - Paravirtualization: ask cooperation of guest to create suitable virtual → hardware page tables (Xen)
  - Hardware assisted: nested page tables: let hardware perform additional translation step

Address Translation & TLB



Shadow Page Tables vs. Paravirtualization vs. Nested Page Tables

- Paravirtualized MMU

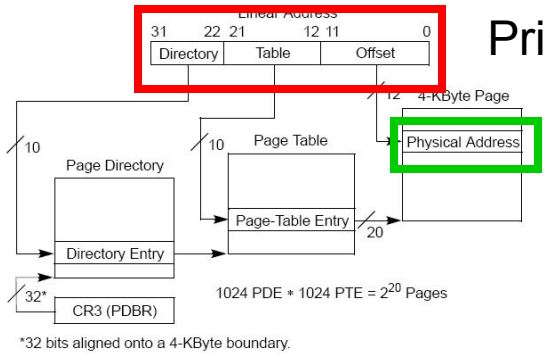


Virtual → Physical → Hardware

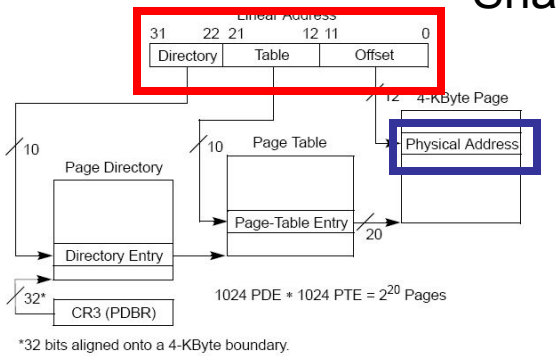
- Nested Page tables eliminate need for either

- Shadow Page Table

Primary



Shadow



Memory Management in ESX

- Have so far discussed how VMM achieves isolation
  - By ensuring proper *translation*
- But VMM must also make resource management decisions:
  - Which guest gets to use which memory, and for how long
- Challenges:
  - OS generally not (yet) designed to have (physical memory) taken out/put in.
  - Assume (more or less contiguous) physical memory starting at 0
  - Assume they can always use all physical memory at no cost (for file caching, etc.)
  - Unaware that they may share actual machine with other guests
  - Already perform page replacement for their processes based on these assumptions

Goals for Virtual Memory

- Performance
  - Is key. Recall that
    - avg access = hit rate * hit latency + miss rate * miss penalty
    - Miss penalty is *huge* for virtual memory
- Overcommiting
  - Want to announce more physical memory to guests that is present, in sum
  - Needs a page replacement policy
- Sharing
  - If guests are running the same code/OS, or process the same data, keep one copy and use copy-on-write

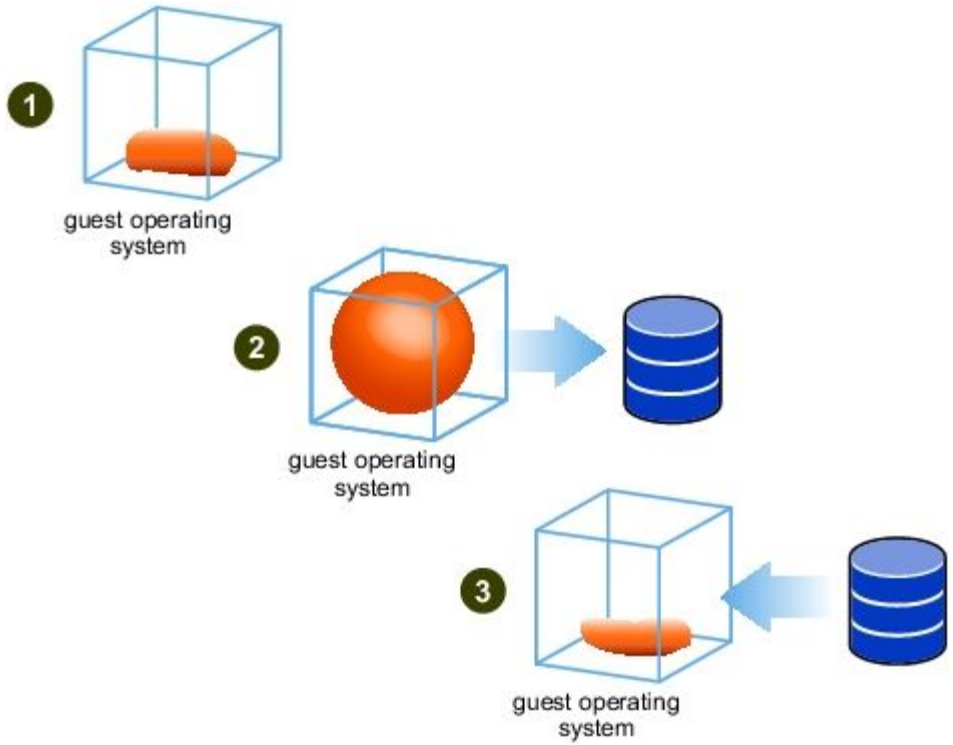
Page Replacement

- Must be able to swap guest pages to disk
  - Question is: which one?
  - VMM has little knowledge about what's going on inside guest. For instance, it doesn't know about guest's internal LRU lists (e.g., Linux page cache)
- Potential problem: Double Paging
  - VMM swaps page out (maybe based on hardware access bit)
  - Guest (observing the same fact) – also wants to "swap it out" – then VMM must bring in the page from disk just so guest can write it out
- Need a better solution

Ballooning

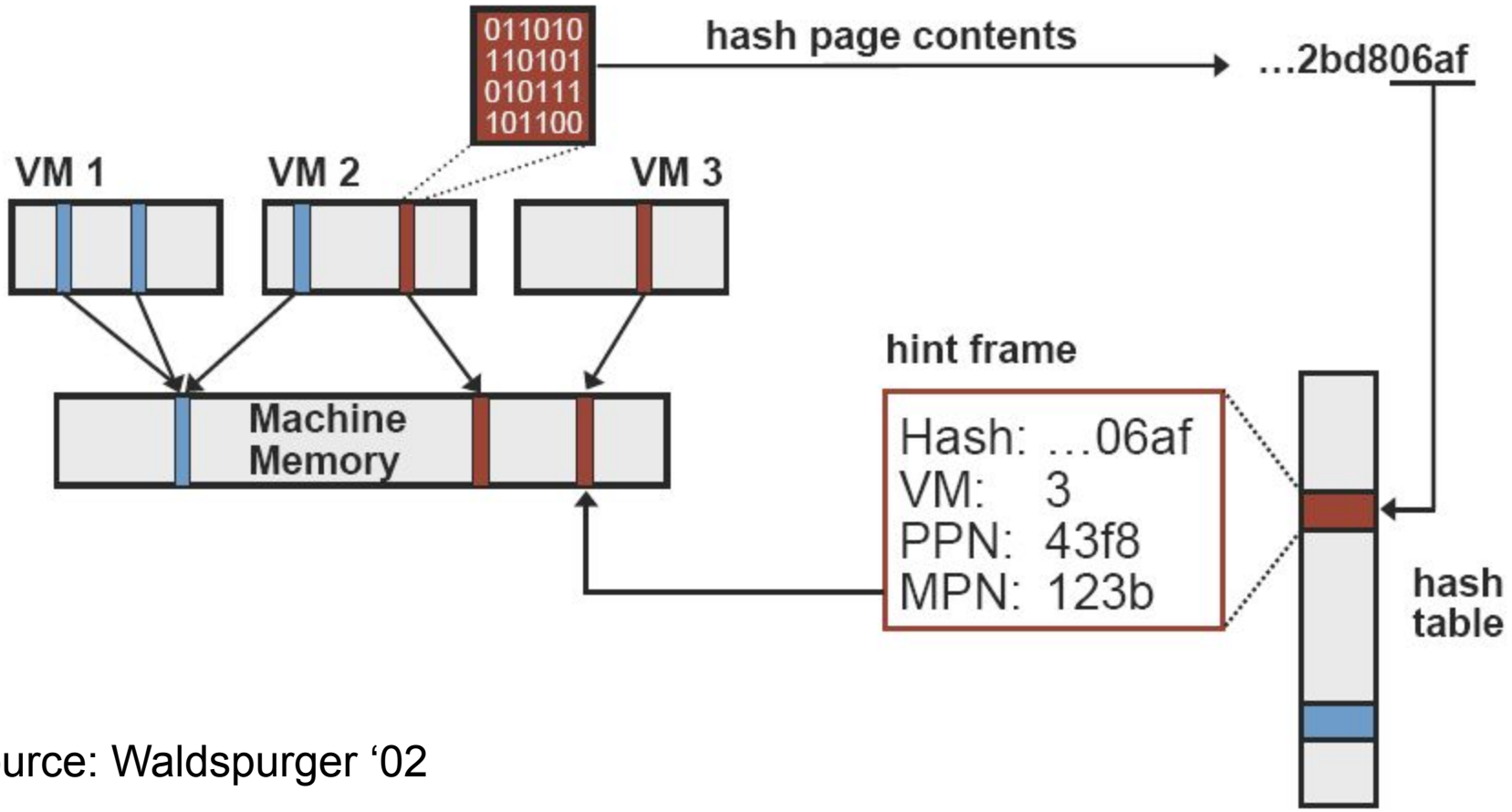
- What if we could trick guest into reducing its memory footprint?
- Download balloon driver into guest kernel
  - Balloon driver allocates pages, possibly triggering guest's replacement policies.
  - Balloon driver pins page (as far as guest is concerned) and (secretly to guest) tells VMM that it can use that memory for other guests
  - Deflating the balloon increases guest's free page pool
- Relies on existing memory in-kernel allocators (e.g., Linux's get_free_page()
- If not enough memory is freed up by ballooning, do random page replacement

Ballooning



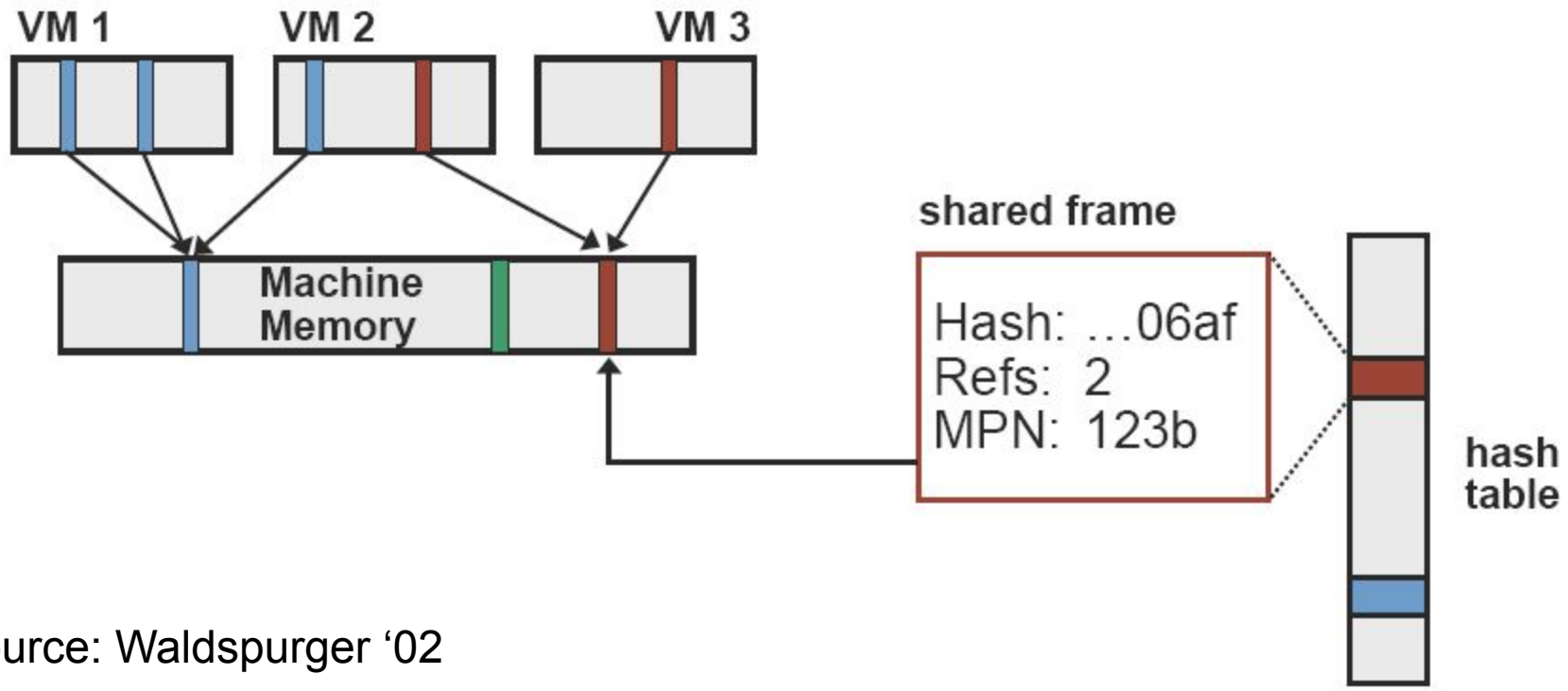
Source: VMware

Page Sharing (1)



Source: Waldspurger ‘02

Page Sharing (2)



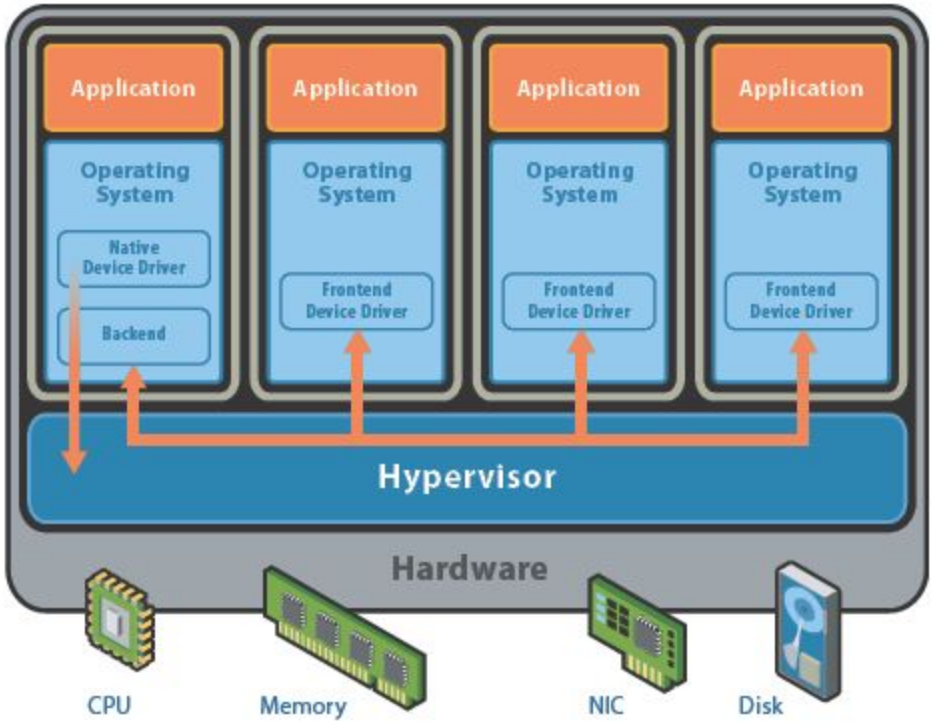
Source: Waldspurger ‘02

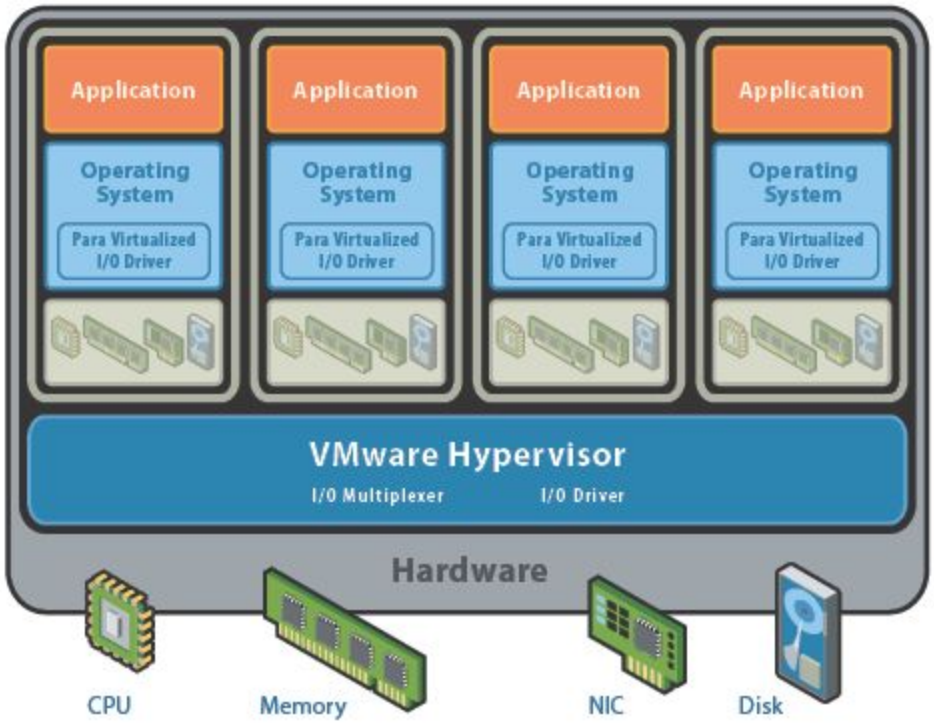
Virtualizing I/O

- Most challenging of the three
  - Consider Gigabit networking, 3D graphics devices
- Modern device drivers are tightly interwoven with memory & CPU management
  - E.g. direct-mapped I/O, DMA
  - Interrupt scheduling

Virtualizing I/O

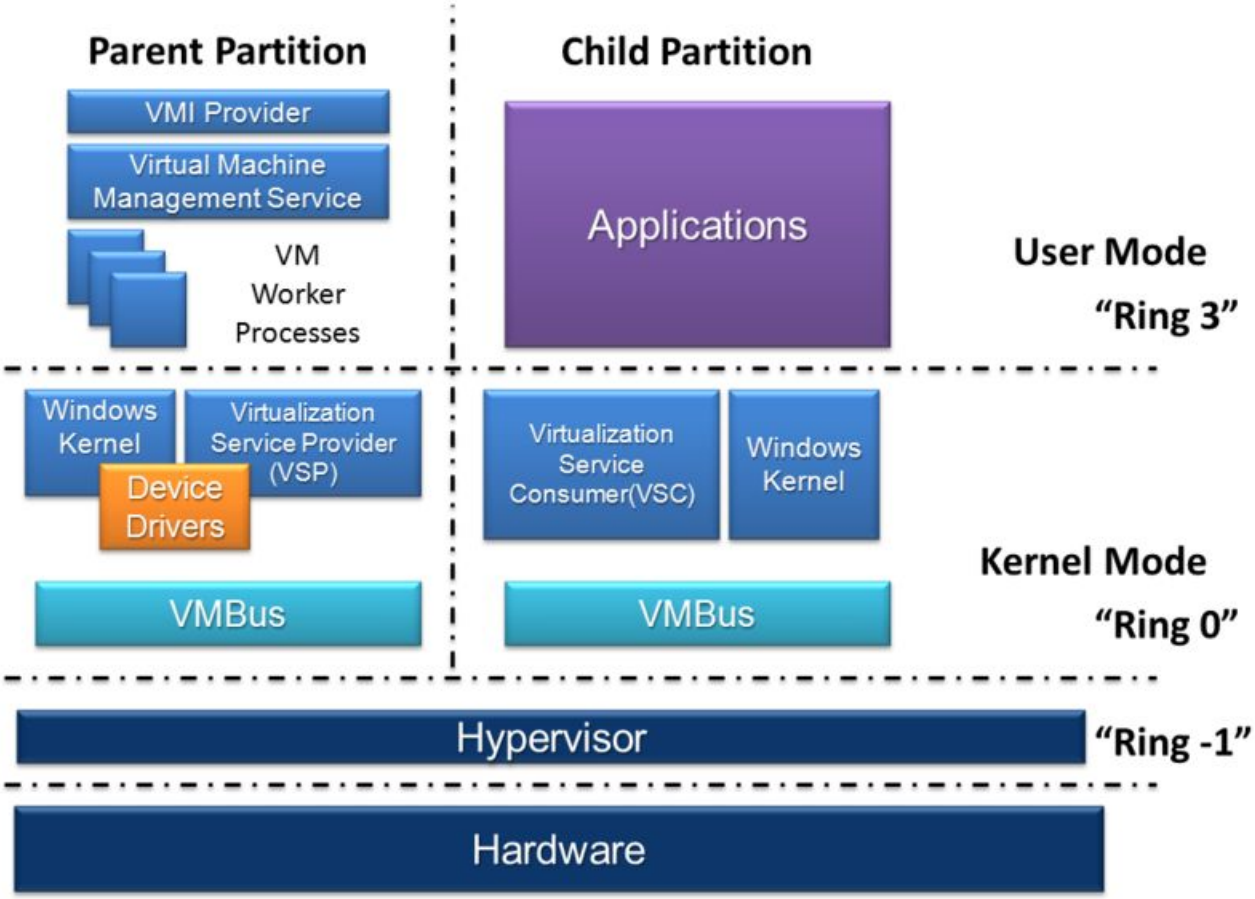
- Xen
- ESX





Source: VMware white paper on virtualization considerations.

Windows Hyper V



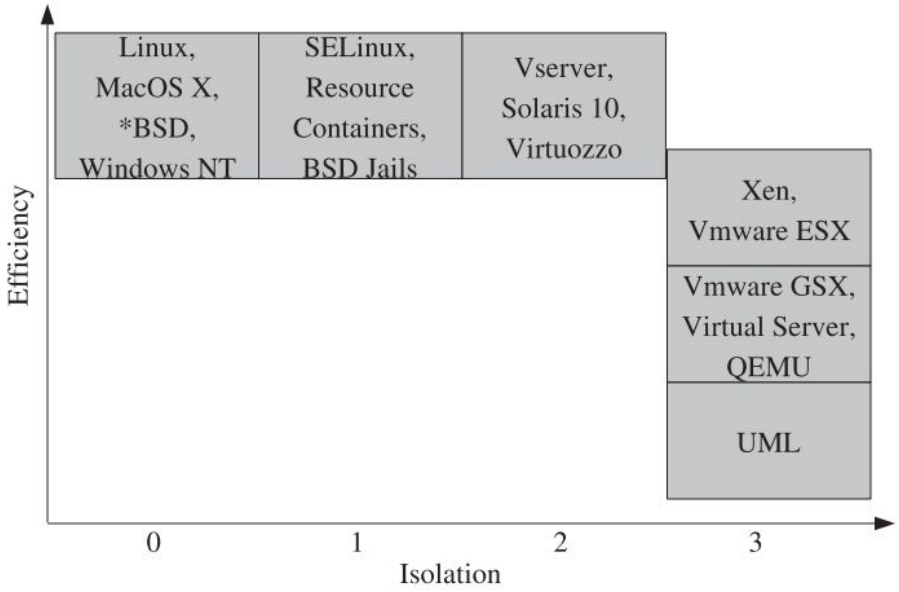
Source: Wikipedia Commons

IOMMU & Self-Virtualizing HW

- IOMMU – hardware support to protect DMA, interrupts space
- Self-Virtualizing – device is aware of existence of multiple VMs above it

Container-Based Virtualization

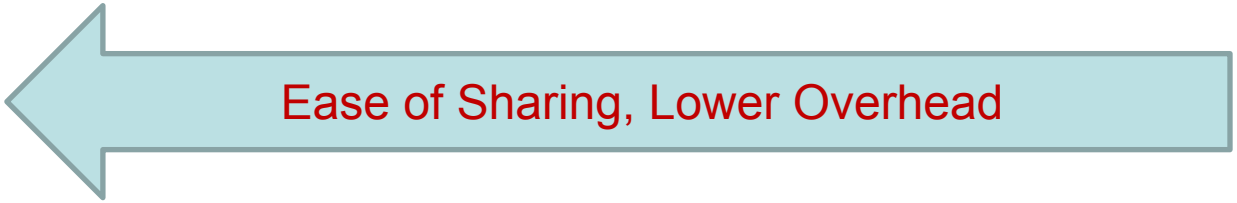
- Provide OS-level virtualization [Soltesz 2007]
- Three levels of isolation:
  - Namespace separation (Security Isolation)
  - Resource Isolation
  - Fault Isolation
- Examples
  - chroot, “jails”
  - Solaris Containers
  - Linux “LXC”
- Commercially available as Docker



Source: Soltesz et al, Eurosys 2007

Virtualization Spectrum

Threads | Processes | Containers | Virtual Machines | Physical Machines



Summary

- Virtualization enables a variety of arrangements/benefits in organizing computer systems
- Two types:
  - Type I: VMM may run on bare hardware
  - Type II: VMM is process running on/integrated with host OS
- Key challenges include virtualization of
  - CPU
  - Memory
  - I/O
- Both correctness and efficiency are important