Virtualization Without Architectural Support

[Hardware and Software Support For Virtualization](#)

Chapter 3

More than a historic reason

- Today, most systems have architectural support
  - X86-64, ARMv8, do it
- Past solutions to “meet” G/P theorem (while ISA didn’t): transcend TAE

	Disco	VMware Workstation	Xen	KVM for ARM
Architecture	MIPS	x86-32	x86-32	ARMv5
Hyp type	Type-1	Type-2 (§4.2.4)	Type-1 with dom0 (§4.4)	Type-2 (§4.5)
Equivalence	Requires modified kernel	Binary-compatible with selected kernels	Required modified (paravirtualized) kernels (§4.3)	Required modified (lightweight paravirtualized kernels (§4.5)
Safety	Via de-privileged execution using strictly virtualized resources	Via dynamic binary translation; isolation achieved via segment truncation	Via de-privileged execution with safe access to physical names	Via de-privileged execution using strictly virtualized resources
Performance	Via localized kernel changes and L2TLB (§4.1.2)	By combining direct execution (or applicastions) with adaptive dynamic binary translation (§4.2.3)	Via paravirtualization of CPU and IO interactions	Via paravirtualization of CPU and IO interactions

Disco

- Research hypervisor for Stanford FLASH (reinvigorated the interest for Virtual Machines)
  - Addresses **fault-containment** without changes in the OS and using a NUMA architecture (the first one)
  - Architected around Trap-and-emulate (TAE)

Region	Base	Length	Access K,S,U	MMU	Cache
USEG	0x0000 0000	2 GB	✓,✓ ✓	mapped	cached
KSEG0	0x8000 0000	512 MB	✓,x,x	unmapped	cached
KSEG1	0xA000 0000	512 MB	✓,x,x	unmapped	uncached
KSSEG	0xC000 0000	512 MB	✓,✓,x	mapped	cached
KSEG3	0xE000 0000	512 MB	✓,x,x	mapped	cached

- **Equivalence**
  - Required to move the kernel code out of KSEG0 (to KSSEG): or every kernel load in guest is a TAE
  - Required **source code of changes** in IRIX and **recompile** the kernel (changing 2 includes, linking options, and 15 assembly statements)
- **Safety**
  - Uses supervisor mode for guest OS
- **Performance**
  - P/G assumes traps were rare is not meet for MIPS (RISC). (e.g., Architected TLB)
  - Read-only privileged registers remapped into memory, **hypercalls**, larger TLB

Disco: Register "Remapping" & Hypercalls

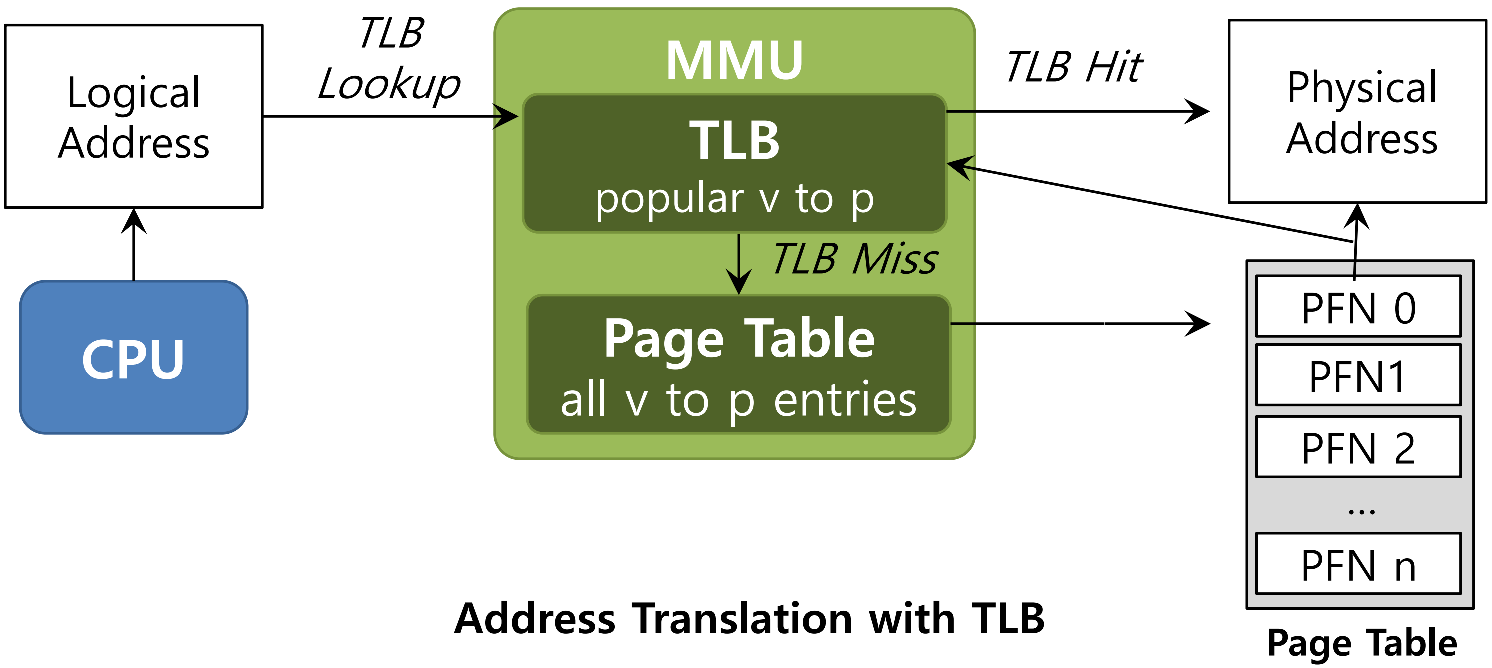
- Avoid frequent traps when guest OS try to access to privileged registers
  - Trap routines and synchronization routines uses them frequently

- Remap the load of those registers to specific memory pages
  - Convert the privileged loads (patching them) into regular memory loads

- **Hypercall** is a higher-level command issued by guest OS to the hypervisor
  - Disco use some to specific tasks
    - V.gr. Free frames guest-physical pages

Review: TLB

- Part of the chip's memory-management unit(MMU).
- A hardware cache of **popular** virtual-to-physical address translation.



Address Translation with TLB

DISCO: L2TLB

- MIPS uses an architected TLB (i.e., the OS will manage it via **exceptions** and software **TLB handler**)
  - Small TLB (<64 entries) and shared across VM KSSEG addresses leads to frequent misses
- Hard for the Hypervisor
  - Hypervisor should understand how guest OS handles TLB (i.e., its internal page table representation) or transfer the control to the guest OS (involving frequent privileged instructions),
  - TLB misses requires **double indirection** (hypervisor→guesOS→app)
  - Needed for a virtualized TLB
- Take advantage of it an implement a second level (in the hypervisor) L2TLB (software) to cache virtual-to-host physical
  - Each VM has his own L2TLB (and **apparently** much greater than the real one)
  - Some nuances with ASID exhaustion

DISCO: Virtualizing the Physical Memory

- Ameliorate the complexities of cc-NUMA mapping
  - Transparent page migration, replication, and sharing
- By default, 1-to-1 hypervisor-physical page to guest-OS physical page
- Others
  - Many-to-one (Copy on write)
  - One-to-many (sharing improves cache pressure in read-only pages)
  - NUMA migrations (reduce the latency to the memory

```
vpuente@compute-gpu-0:~$ sudo numactl --hardware
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 32 33 34 35
node 0 size: 32097 MB
node 0 free: 30877 MB
node 1 cpus: 4 5 6 7 36 37 38 39
node 1 size: 16125 MB
node 1 free: 15219 MB
node 2 cpus: 8 9 10 11 40 41 42 43
node 2 size: 16125 MB
node 2 free: 14937 MB
node 3 cpus: 12 13 14 15 44 45 46 47
node 3 size: 8061 MB
node 3 free: 6498 MB
node 4 cpus: 16 17 18 19 48 49 50 51
node 4 size: 32253 MB
node 4 free: 31868 MB
node 5 cpus: 20 21 22 23 52 53 54 55
node 5 size: 16125 MB
node 5 free: 10104 MB
node 6 cpus: 24 25 26 27 56 57 58 59
node 6 size: 16104 MB
node 6 free: 8043 MB
node 7 cpus: 28 29 30 31 60 61 62 63
node 7 size: 8060 MB
node 7 free: 7271 MB
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  16  16  16  32  32  32  32
  1:  16  10  16  16  32  32  32  32
  2:  16  16  10  16  32  32  32  32
  3:  16  16  16  10  32  32  32  32
  4:  32  32  32  32  10  16  16  16
  5:  32  32  32  32  16  10  16  16
  6:  32  32  32  32  16  16  10  16
  7:  32  32  32  32  16  16  16  10
```

VMWare Workstation

- Launch in 1999 to support virtualization on (initially) ***Wintel*** platform
  - Address *Wintel* limitations in security, reliability, application interoperability, OS migration
  - Focus on unmodified guest-OS
  - Hosted by Windows95/NT or Linux (type-2 hypervisor)

- **Equivalence**
  - X86-32 has 17 virtualization-sensitive (or critical) instructions. Not possible TAE.
  - Use **dynamic binary translation** (an efficient form of emulation)

- **Safety**
  - Use **segment-truncation** to isolate VM. Limited to used features of the ISA (e.g., certain privilege levels usually non used where ignored)

- **Performance**
  - Goal: at least run as fast as the previous generation of processors
  - Use direct execution as much as possible

X86-32 Fundamentals

- Legacy (real, sys management, v8086) and native execution mode (called **protected**), four levels, additional privilege I/O
- `In` protected mode, **kernel `%cpl=0`,** user level `4>%cpl>0` (usually `%cpl=3` in anything but the kernel)
- I/O privilege level (`iopl`) allows to user-level code to enable and disable interrupts
- Memory uses a combination of segmentation and paging
  - Segments define base and bounds (`%cs` code, `%ss` stack, `%dd` data) and extra segments (`%es, %fs, %gs`) a portion of the 32-bit address space
- Paging is applied inside the segments
- Base register of page table (directory) at `%cr3`
- **TLB is not architected**: walker handles this (no trap after TLB miss)

Virtualizing the x86-32 CPU

- Call Direct Execution(**DE**) when under G/P (privileged → TAE)
  - Not enough with a non-virtualizable arch.

- Experience with **SimOS** simulator (full system simulator for FLASH) in Dynamic Binary Translation (**DBT**) showed the potential to circumvent the problem
  - Still x5 slowdown: great for a simulator not so for a virtual machine (violates P/G performance criteria)

- VMWare insights
  - (1) Use DE when possible, Use DBT when isn't
  - (2) x86 segment protection can allow DBT to run also near-to-native speed

Virtualizing the x86-32 CPU

Input: Current state of the virtual CPU
Output: True if the direct execution subsystem may be used;
False if binary translation must be used instead

```
if !cr0.pe then
    return false;
end if
if eflags.v8086 then
    return true
end if
if (eflags.iopl ≥ cpl)||(!eflags.if) then
    return false;
end if
for all seg ← (cs, ds, ss, es, fs, gs) do
    if "seg is not shadowed" then
        return false;
    end if
end for
return true
```

If not protected mode (v.gr. BIOS) DBT

If v8086 mode (v.gr. MSDOS inside windows95) DE

VM can control interrupts (e.g. Linux `ioperm()`syscall)

Non accessible segments (corner case in windows95)

No assumptions about Guest, easy to implement with a handful of ASM instructions

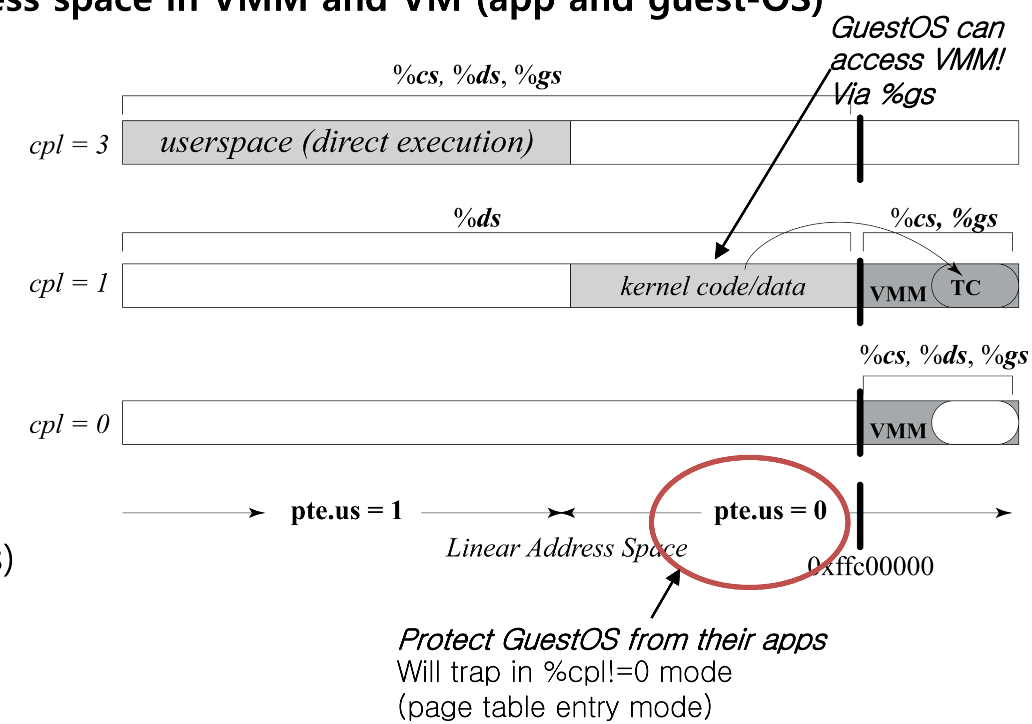
X86-32 is not hybrid-virtualizable

- There are non-privileged user-sensitive instructions (P/G hybrid theorem does not apply!) [**i.e., they are CS or BS in user-mode**]
  - `sgdt, sidt, sldt, smsw`
  - Not useful to applications (according intel manuals)

- Although segment truncation guarantees isolation, it is visible via `lsl` (load segment limit) instruction to communicate from the application with the VMM
  - Allows the app to known if is being used in a virtualized environment
  - Violates the equivalence requirement of P/G

VMWare DBT

- DBT rather than emulate instruction by instruction, compiles a group of then (usually a **basic block**) into a fragment of executable.
  - Fragments are stored in a large buffer called **translation cache (TC)**
- DBT performance very sensitive to hardware config
  - V.gr. Embra (MIPS) had a x5 slowdown due to MMU emulation
- DBT should "**share**" the address space with the DE (Direct Execution) (avoid MMU!)
  - **Segment truncation: divide the address space in VMM and VM (app and guest-OS)**
  - Relay in hardware memory protection to isolate them (different `cpl` and `pte.us`)
    - If cpl of the pte of code < cpl trying to access → trap
- **Adaptive Dynamic Binary Translation**
  - Reuse fragments and chain them directly (by direct jumps avoiding traps)



VMWare World-switch

- Low level mechanism that frees VMM from any interference from the host OS (and vice versa)

- Like a traditional context-switch (save `%eip`, `%esp`, `%efp`, `%cr3`, etc…) but also change also the address space of the kernel (descriptor tables: segments and interrupts) and no assumptions about the caller/callee convention

- Changes in `%cr3` are "subtle"
  - As soon as `%cr3` is changed the instruction stream change (because the page table changes)
  - Guarantee that the outgoing and ingoing context uses the same virtual address in the page where `%cr3` changes

- Temporally descriptor tables will be pointing to an undetermined address (potentially invalid)
  - `%idtr` (interrupt description table) is not guaranteed unless interruptions are reenabled
  - `%gdtr` (segment description table) is not guaranteed unless segment assignation is made

Review: Regular Context Switch

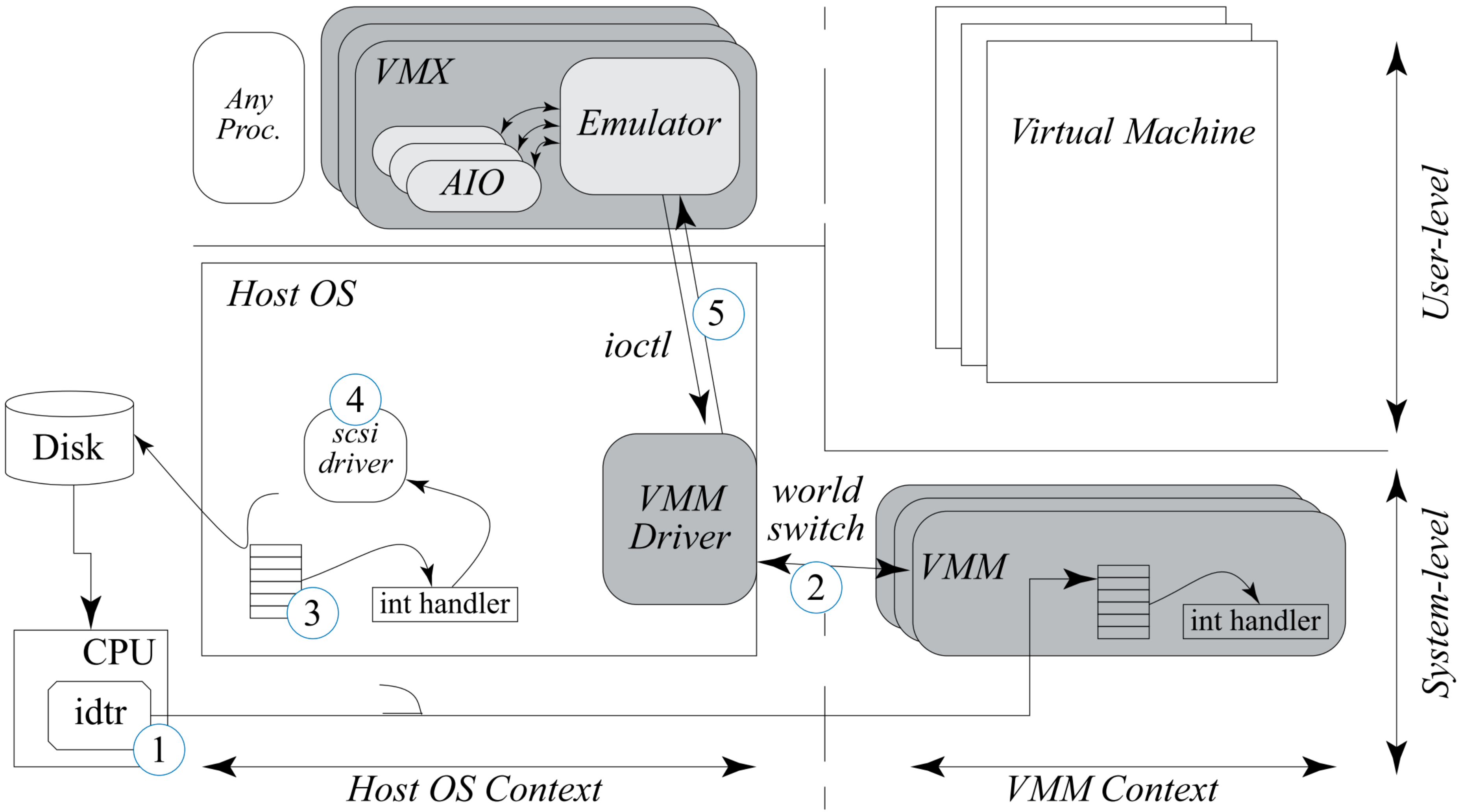
```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
        # Save old registers
         movl 4(%esp), %eax            # put old ptr into eax
        popl 0(%eax)                   # save the old IP (pop from stack to mem)
        movl %esp, 4(%eax)             # and stack
        movl %ebx, 8(%eax)             # and other registers
        movl %ecx, 12(%eax)
        movl %edx, 16(%eax)
        movl %esi, 20(%eax)
        movl %edi, 24(%eax)
        movl %ebp, 28(%eax)

        # Load new registers
        movl 8(%esp), %eax             # put new ptr into eax
        movl 28(%eax), %ebp            # restore other registers
        movl 24(%eax), %edi
        movl 20(%eax), %esi
        movl 16(%eax), %edx
        movl 12(%eax), %ecx
        movl 8(%eax), %ebx
        movl 4(%eax), %esp             # stack is switched here
        pushl 0(%eax)                  # return addr put in place
        ret                            # finally return into new ctxt
```

Sequence of a Disk Interrupt

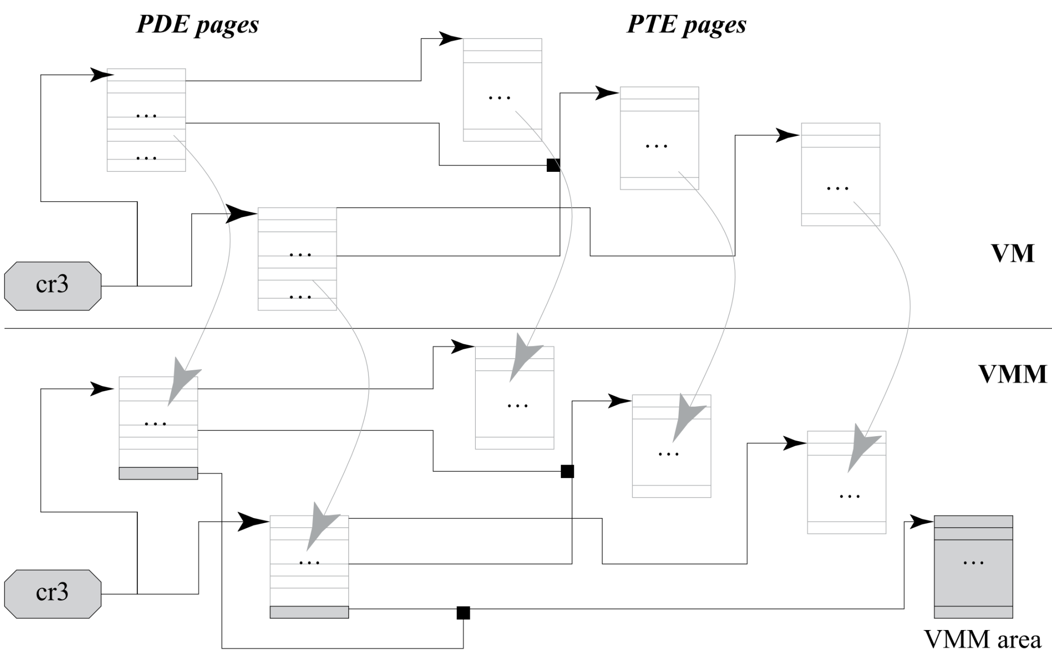
- (1) Hardware interrupts VMM. `%idtr` points to VMM interrupt handler
- (2) VMWare can't handle I/O requests, **world-switch** back to host OS
- (3) **VMM Driver** in HostOS redirects (via software) the same interrupt to the real system
- (4) HostOS interrupt handler (disk)
- (5) Resumes interruption in the code in VMM Driver, moves the data (via `ioctl`) to **VMX** (where the corresponding emulated disk resides). VMX is an application (in **user-space**) of the host emulating VM devices
- VMX is guest-OS dependent (VMWare tools).

Sequence of a Disk Interrupt

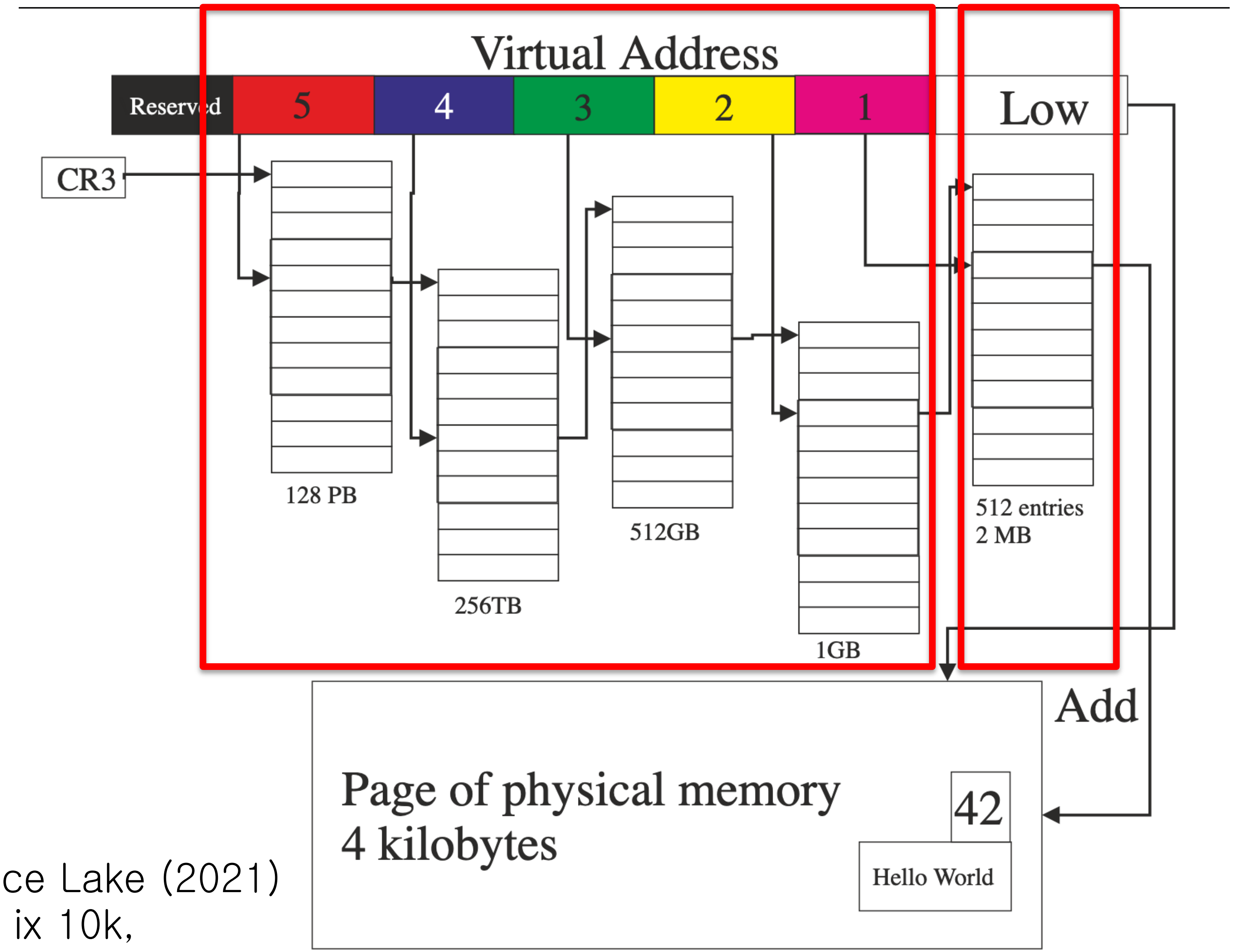


VMWare: Memory

- **Memory Tracing**
  - Trace a page in the guest physical address space and notify the VMM in any write (and some rare reads)
- Shadow Page Tables
  - For each process in the VM we should keep a page table referring to host-physical pages. **Real mapping is hidden from guest-OS**
  - **Memory tracing** to detect changes in the guestOS page tables, and mimic them in the shadow PT
  - Hardware only see shadow PT
  - Frequent TLB flushes



Review: Real PT (5-level page table)



Since Ice Lake (2021)
- Core ix 10k,
- Xeon 3rd Gen

Xen: Paravirtualization Alternative

- VMWare focus on equivalence hinder system-intensive workloads performance

- **Paravirtualization**
  - Trade-off **equivalence** requirements for **performance**
  - Most prominent example of PV is **Xen** (2002) running on x86-32

Xen: Paravirtualization Alternative

- **Xen** is a System Level VMM
- Originally developed in the University of Cambridge
  - Initially released year 2003
- Currently multiple “flavors”
  - XenServer, XenApp/XenDesktop: owned by Citrix and targets support and certified appliances for enterprise environments. Both server and Desktop virtualization (VDI and Applications)
  - XenServer core / Xen-source: open source GPLv3. Only Server virtualizations
- Currently available for IA-32, x86_64 and ARM architectures

Xen: CPU

- Non-virtualizable instructions are **never** executed by the guestOS
  - Guest explicitly communicate with hypervisor when need to change segment descriptors, disable interrupts, receive interrupt notifications, and transition between user and kernel.
- Resulting virtualized ISA meets G/P, therefore TAE for the rest
- Uses rings
  - `%cpl=1` for guestOS
  - `%cpl=0` for applications in the guestOS

Xen: Memory Paravirtualization

- Only Xen hypervisor runs on `%cpl=0`

- Uses **segment truncation**
  - All top 64MB linear space of the segments are excluded from the VM. Xen is available in that region.

- Paravirtualized MMU
  - Each VM is a set of contiguous guest-physical pages and discontinuous host-physical pages. Both are **visible** to the guest-OS
  - Guest-OS can **directly** access to the hardware-handled page tables (no need for memory tracing or shadow page tables)
    - **Reads** are done transparently by TLB walker
    - **Writes** in the PT should be **validated** by the hypervisor. Batched for performance.

Design options for Type-1 hypervisors (I/O)

- Xen vs VMWare ESX