CSIE 5310

Memory Virtualization

Prof. Shih-Wei Li

Department of Computer Science and Information Engineering
National Taiwan University

Agenda

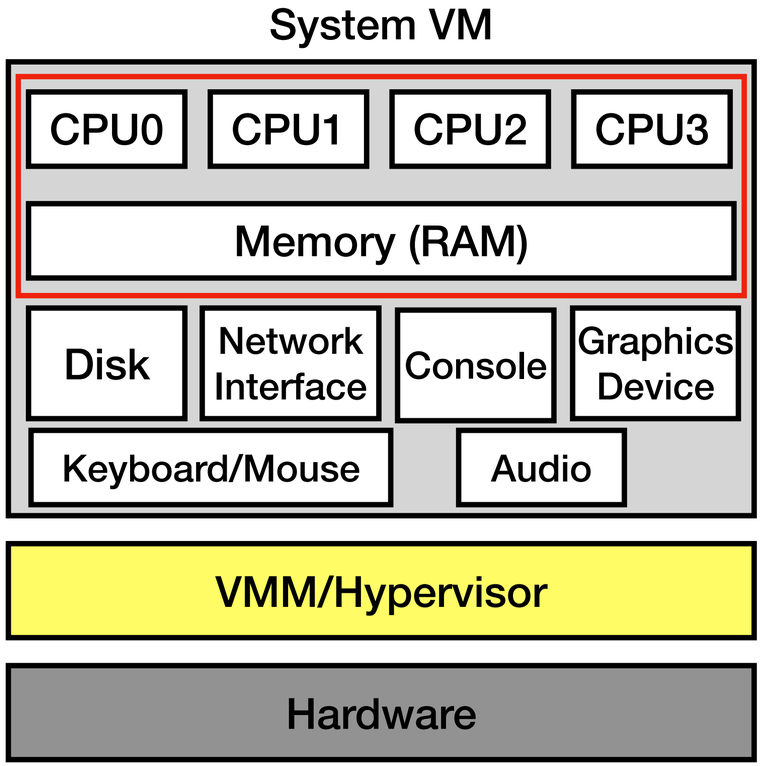
- **Memory Virtualization**
  - **Review on MMU**
  - Goals and Approaches

Today's lecture: Memory Virtualization

- Focus on virtualizing RAM
  - VMMs provide virtualized memory interface
- VMs think they are using real memory
  - CPUs execute programs loaded to memory
  - Instructions (ex: load/store) access memory

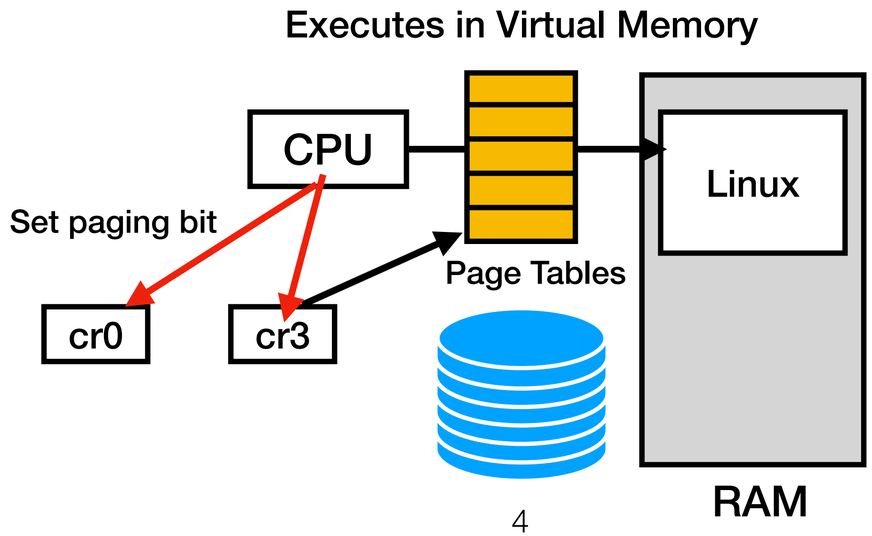
LDR X1, [X2]
STR X3, [X4]

LDR X2, ADDR1
STR X3, ADDR2



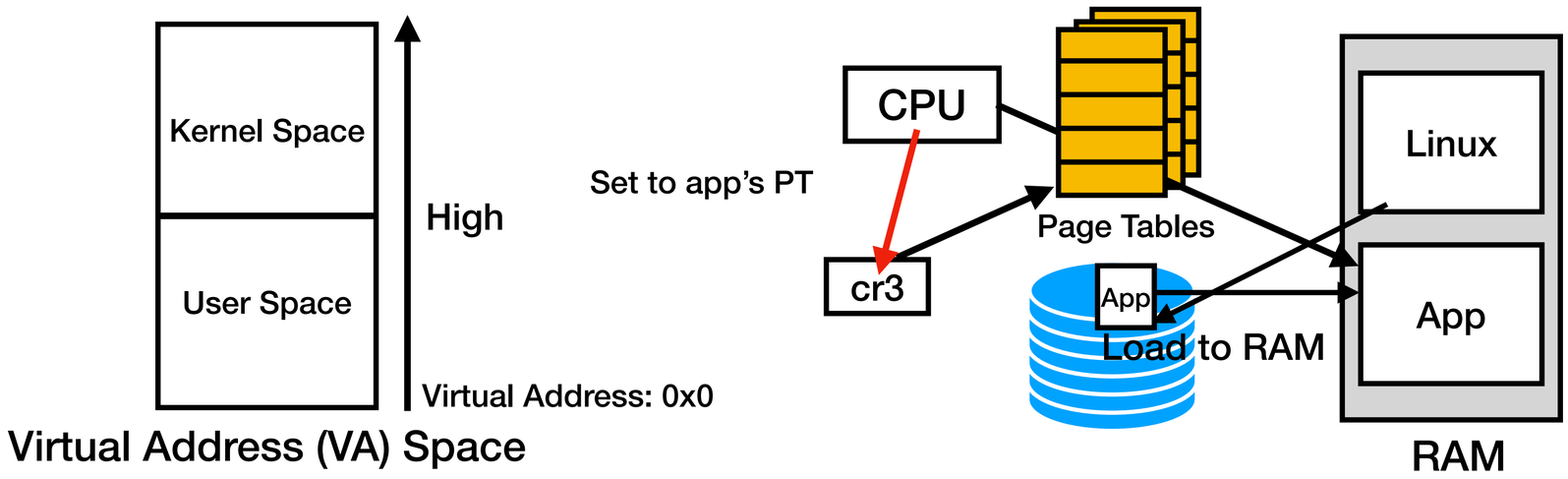
Machine Bootstrapping & Execution

- The OS kernel executes in physical memory and then enables virtual memory
  - Also called enable “**paging**”; use page tables for memory translation



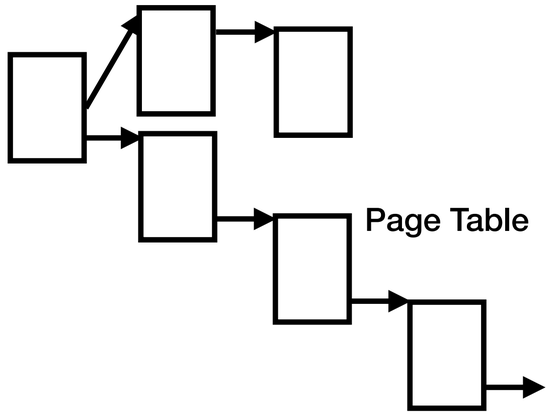
Machine Bootstrapping & Execution

- The kernel loads applications from storage and enters user space
  - Builds page tables for the application — creates its virtual address (VA) space

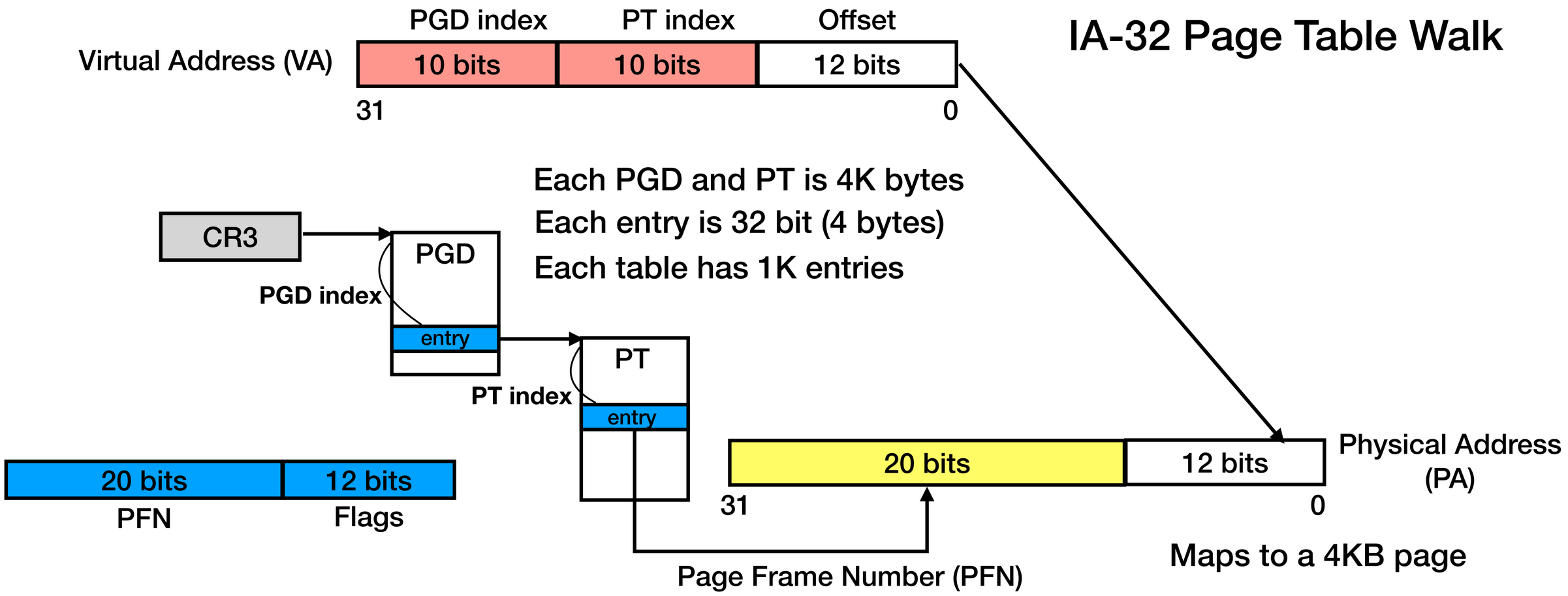


Multi-level page tables

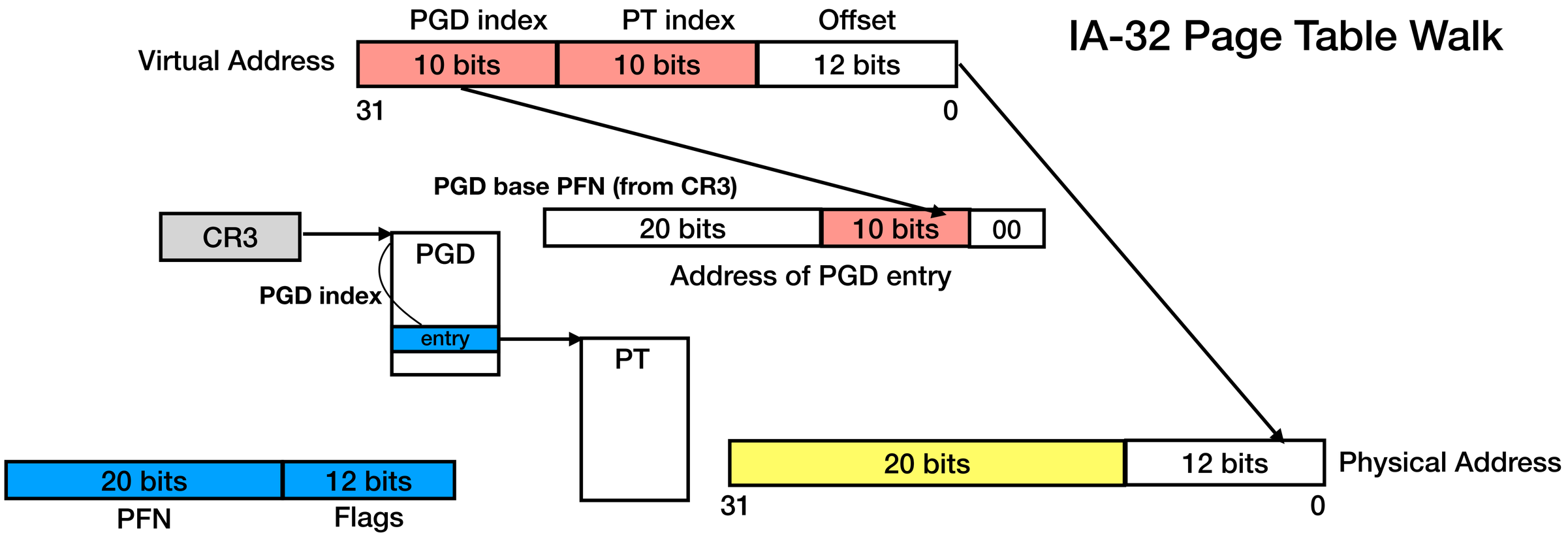
- The hardware (MMU) walks page tables in **physical addresses**
  - Translates virtual addresses (VAs) to physical addresses (PAs)
  - Validates access permissions during the page table walk
- Page tables have multiple levels
  - The exact levels needed depend on the address length
  - Example: more levels are needed for 64-bit addresses than 32-bit
- Support different mapping granularity: regular page (4KB) or huge page (2MB)



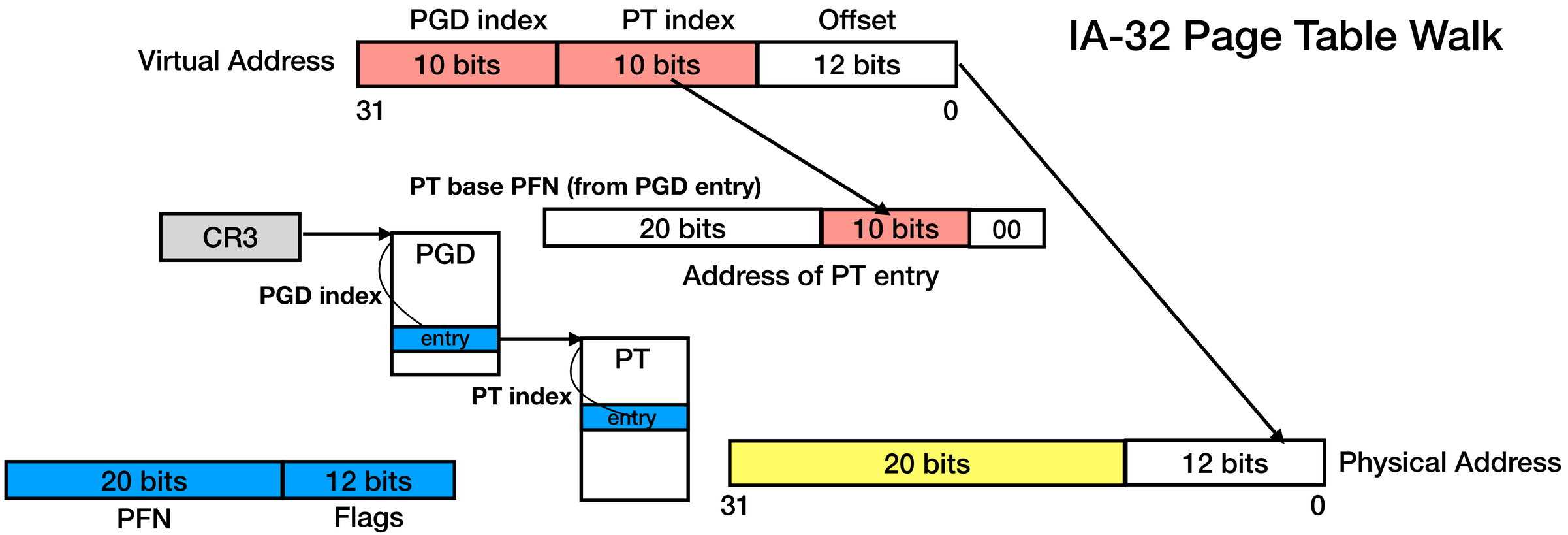
Multi-level page table walk



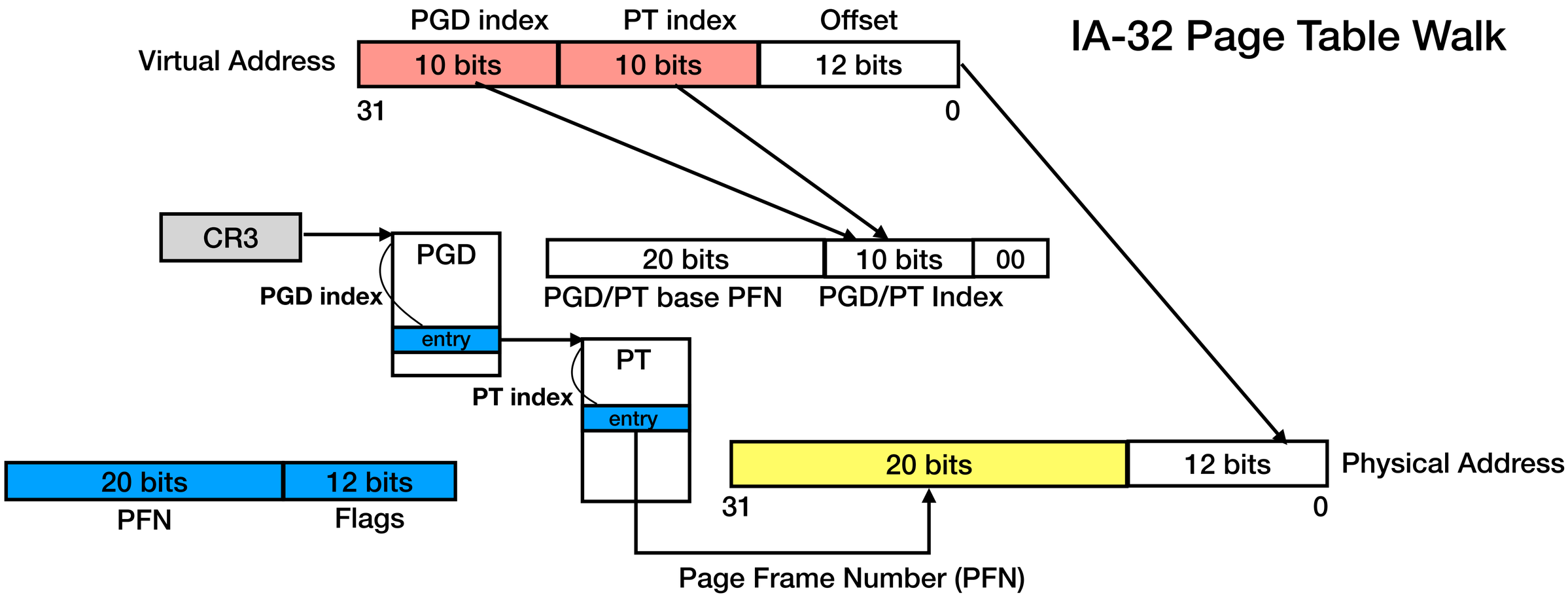
Multi-level page table walk



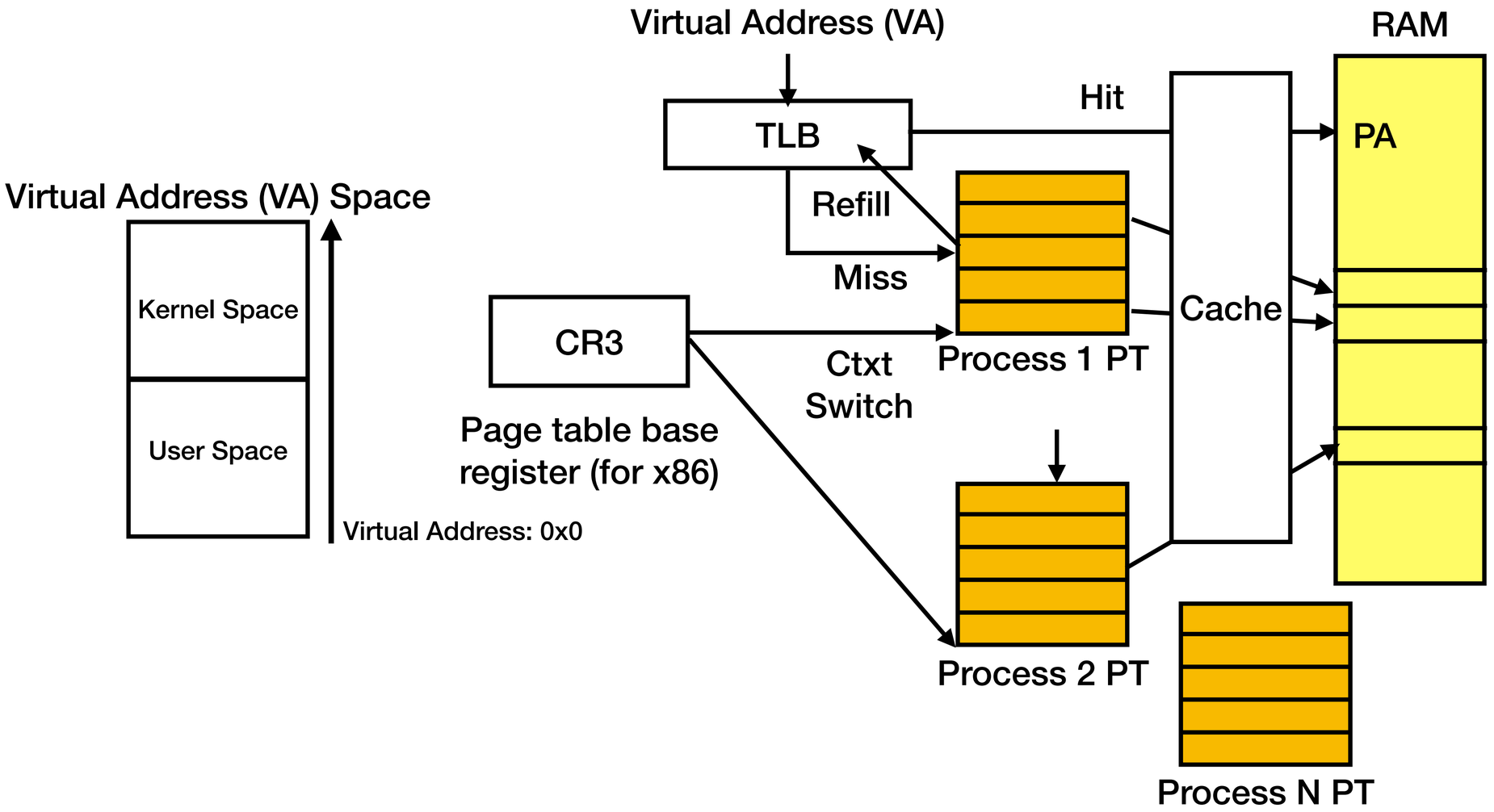
Multi-level page table walk



Multi-level page table walk

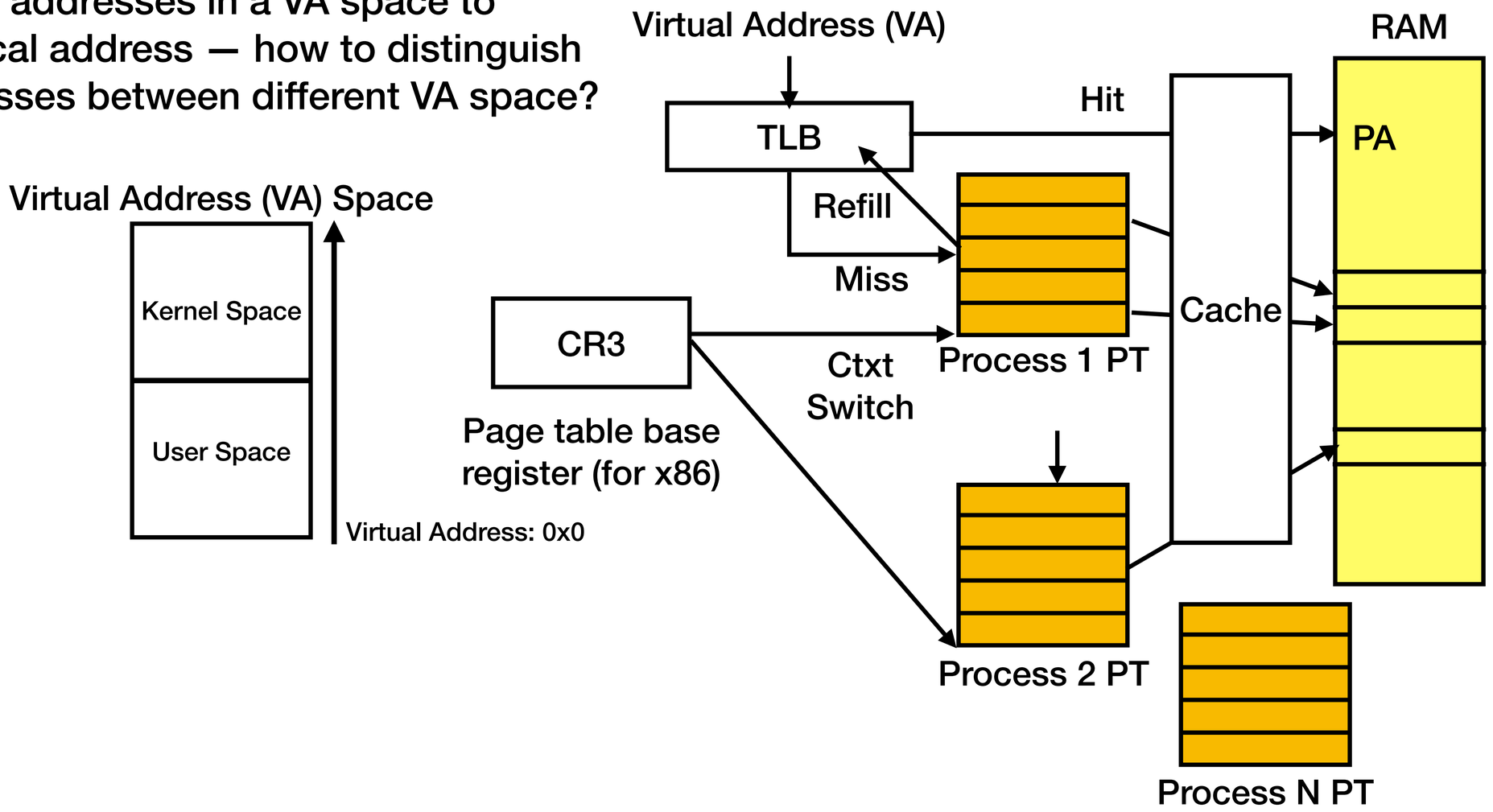


Virtual Memory



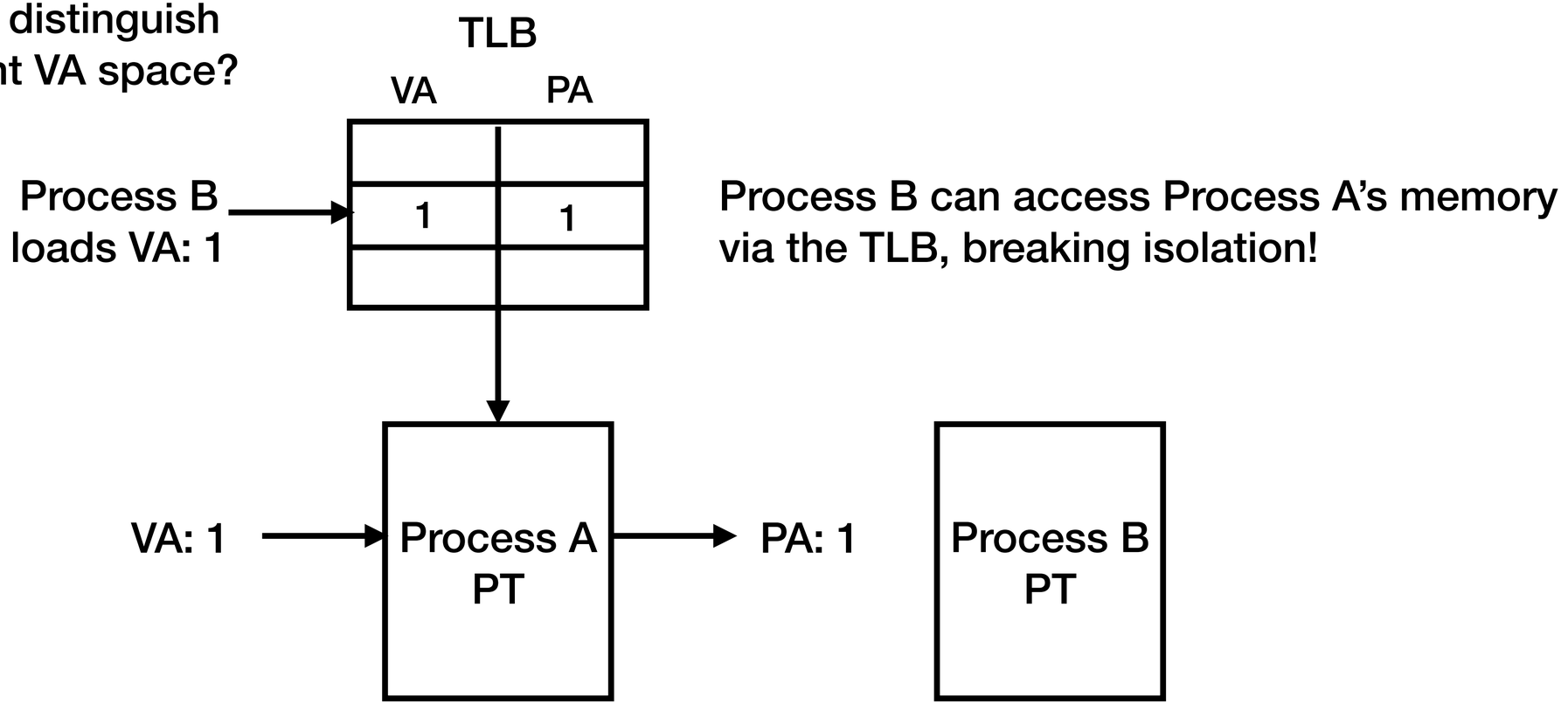
Virtual Memory

Each PT translates the same range of virtual addresses in a VA space to physical address — how to distinguish addresses between different VA space?



Virtual Memory

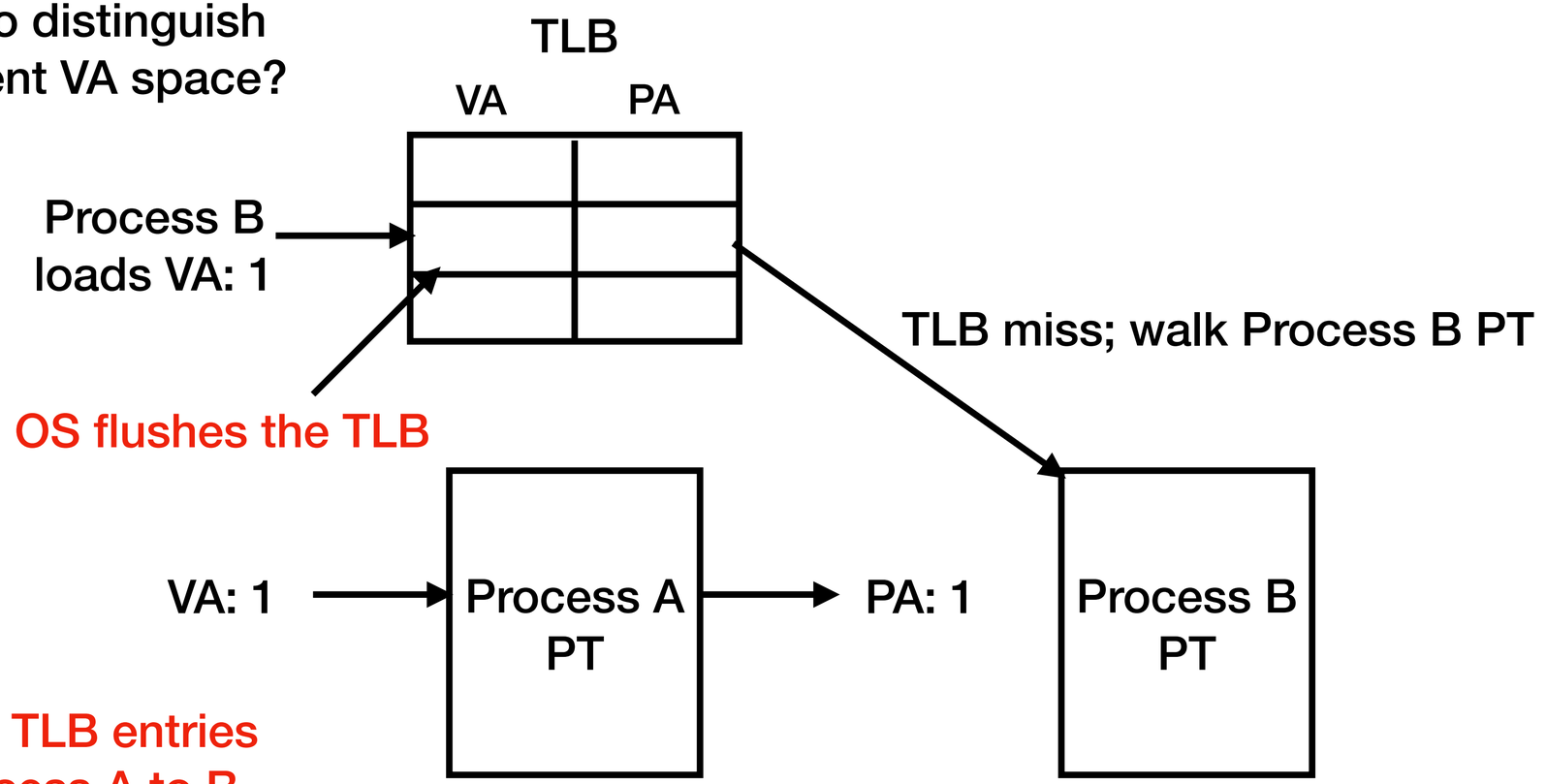
Each PT translates the same range of virtual addresses in a VA space to physical address — how to distinguish addresses between different VA space?



A: the MMU cannot, causing TLB conflict

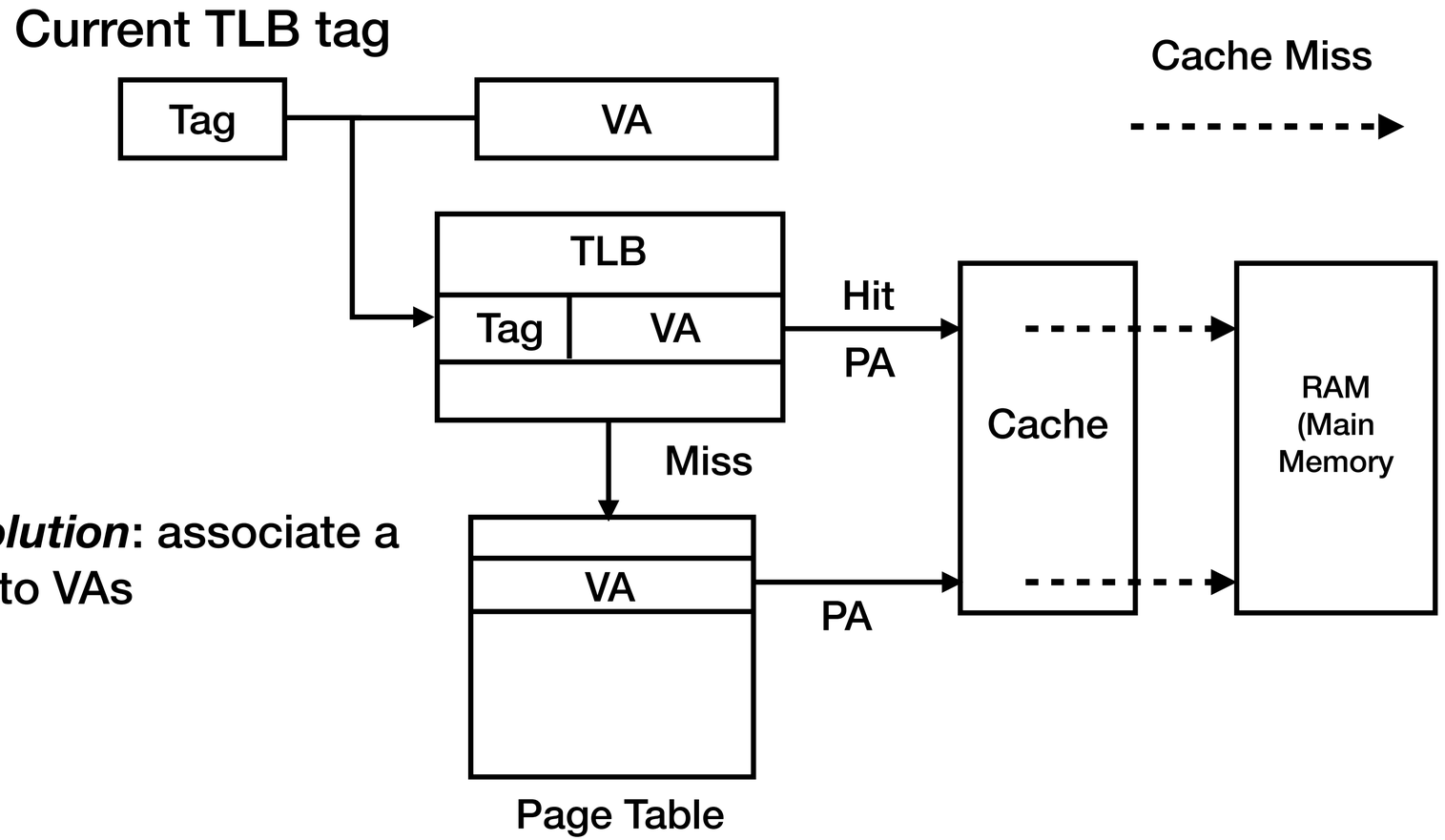
Virtual Memory

Each PT translates the same range of virtual addresses in a VA space to physical address — how to distinguish addresses between different VA space?



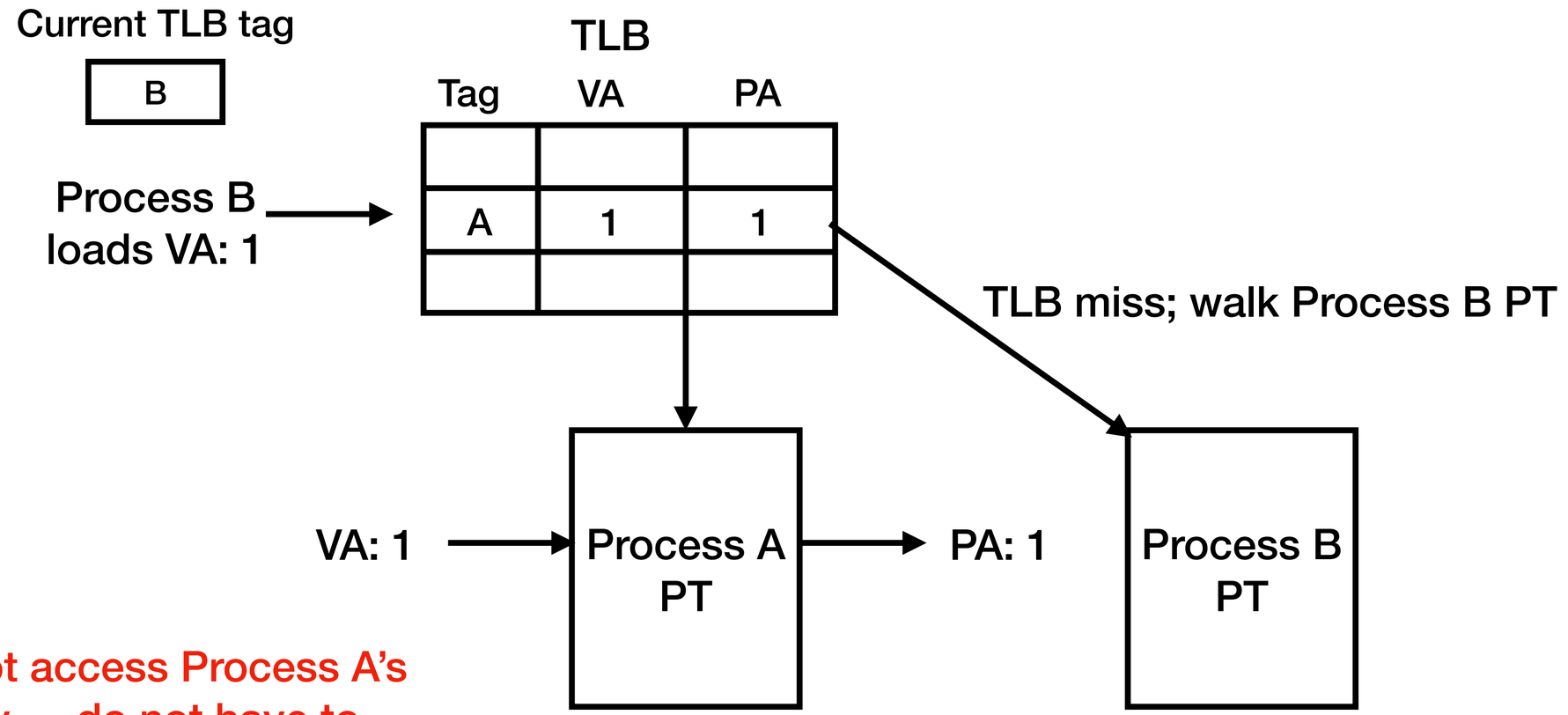
Solution: flush all cached TLB entries when switching from Process A to B, any issues?

Virtual Memory



Better solution: associate a TLB tag to VAs

Virtual Memory



Process B cannot access Process A's tagged TLB entry — do not have to flush TLBs in process switches

TLB (Translation Lookaside Buffer)

- A cache of the page tables to speed up MMU's address translation
  - MMU first walks the TLB; a miss causes a hardware page table walk
- TLB maintenance:
  - Different processes that use the same VA may cause TLB conflict
    - Solution: flush TLBs during process switches; what if this happens frequently?
    - Solution: Hardware support TLB tagging to disambiguate VAs
  - Can you name the other cases in which TLB maintenance is necessary?

Page Faults (1)

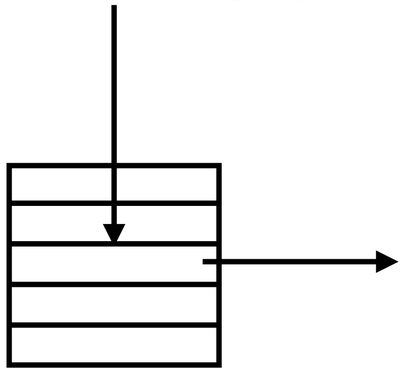
- What causes page faults?

Page Faults (2)

- Access to a virtual address that has invalid mapping causes a page fault
  - Happens when the MMU walks the page table; reasons:
    - The virtual address is unmapped
    - Permission for the virtual address is unmatched
- The hardware sends a signal to the CPU to notify the page fault; in an OS environment:
  - Moves the PC to the exception vectors provided by the OS
  - Switches CPU mode to system mode so the kernel can handle the fault

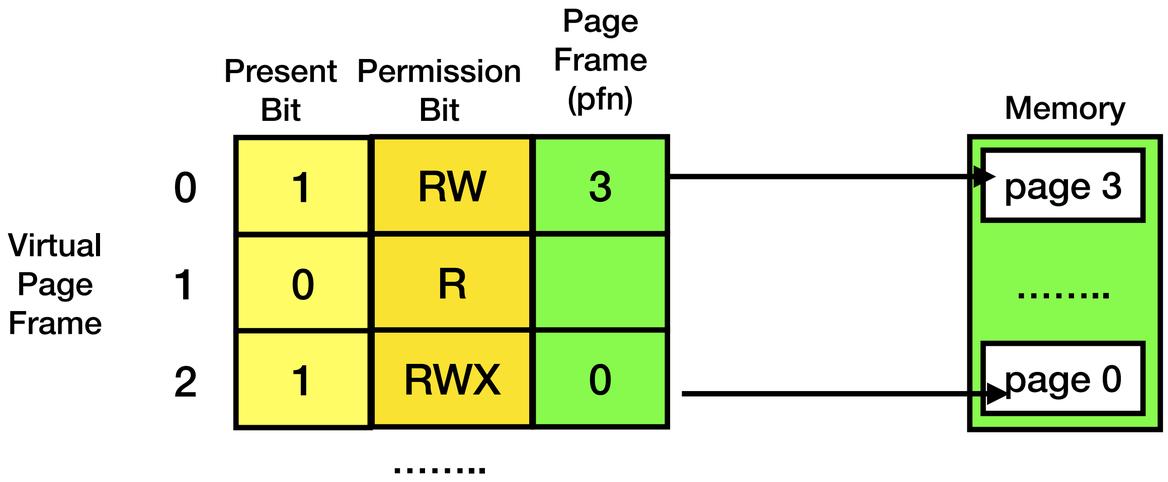
Page Faults (2)

CPU accesses virtual page frame 1 causes a page fault



Exception Vector

Calls the OS page fault handler



How does the handler know which address causes a fault?

CPU that executes code in page frame 0 also causes a page fault — why?

Agenda

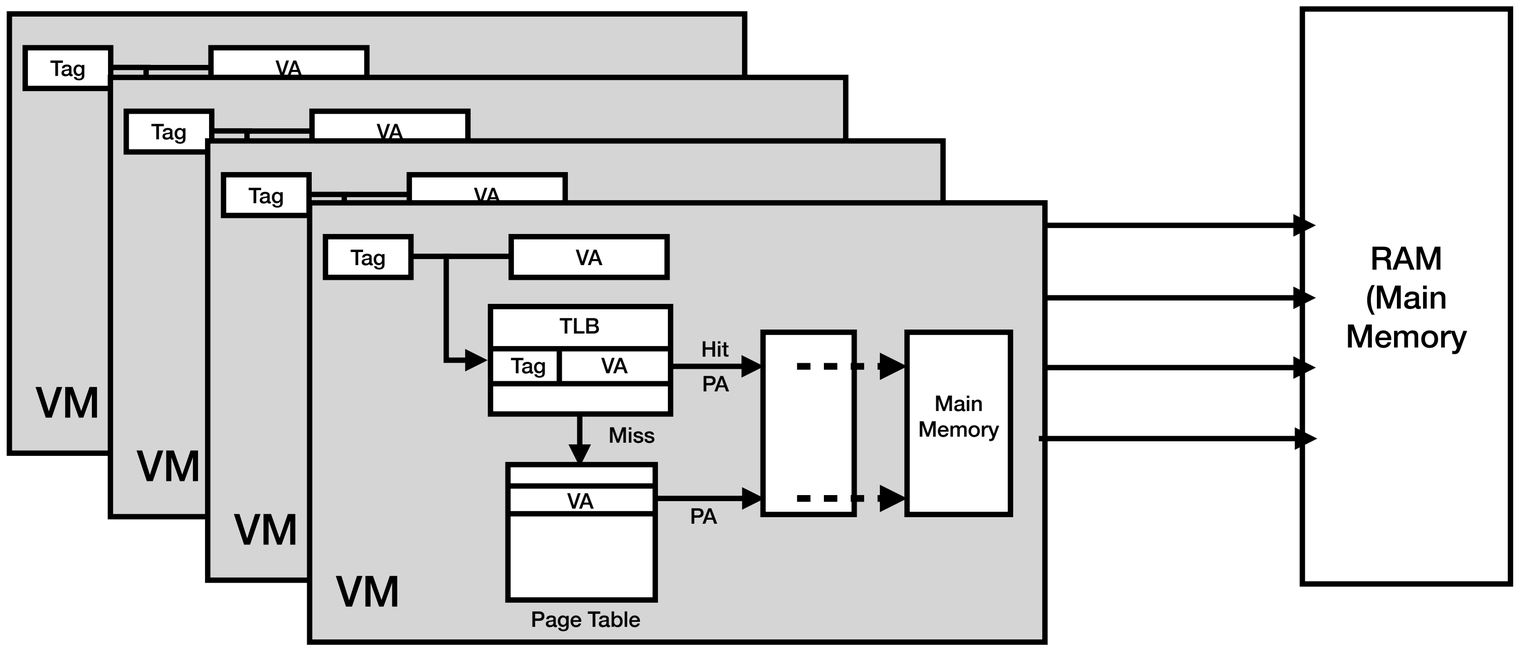
- **Memory Virtualization**
  - Review on MMU
  - **Goals and Approaches**

Goals: Memory Virtualization

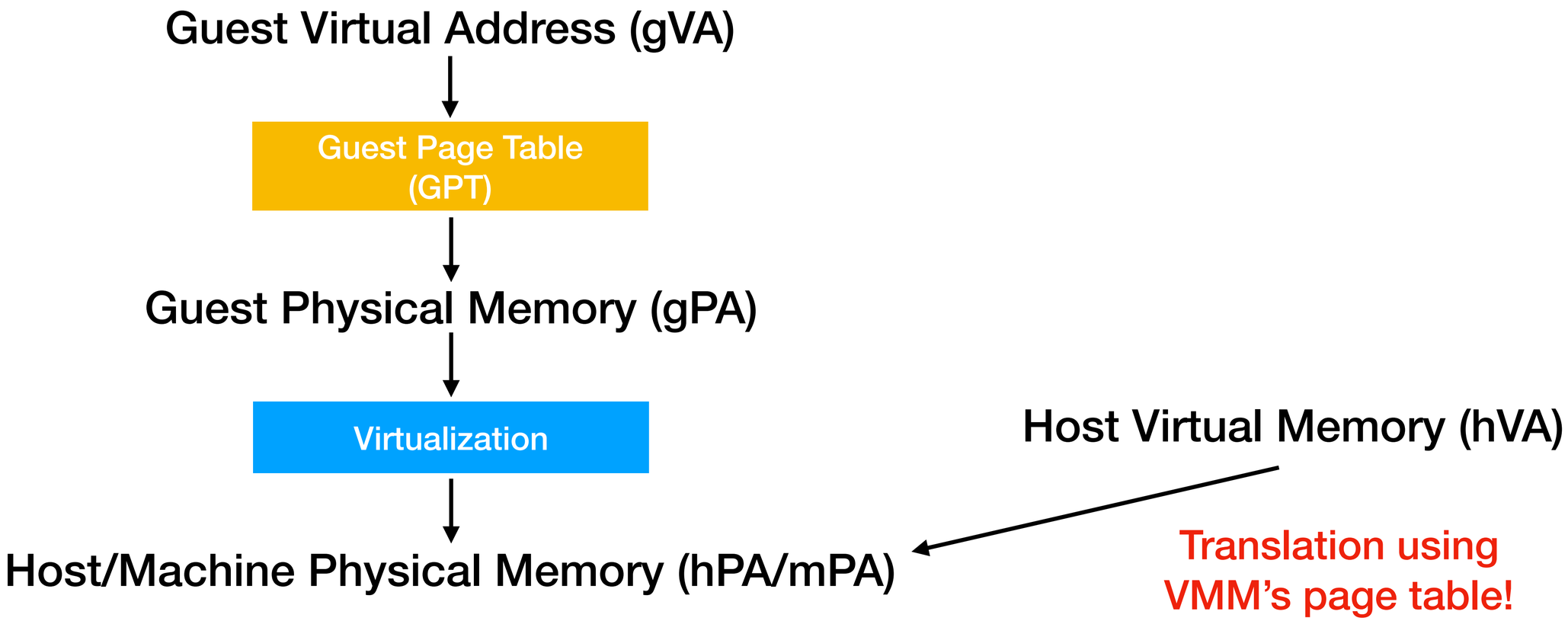
- Support programs in VMs running in physical/virtual memory
- Isolating VM memory accesses
  - VMs cannot access the VMM/hypervisor’s memory
  - VMs cannot affect the hypervisor’s or other VMs’ memory operations

Goal: Memory Virtualization

- Virtualize memory such that guest thinks that it manages real memory



Terminology

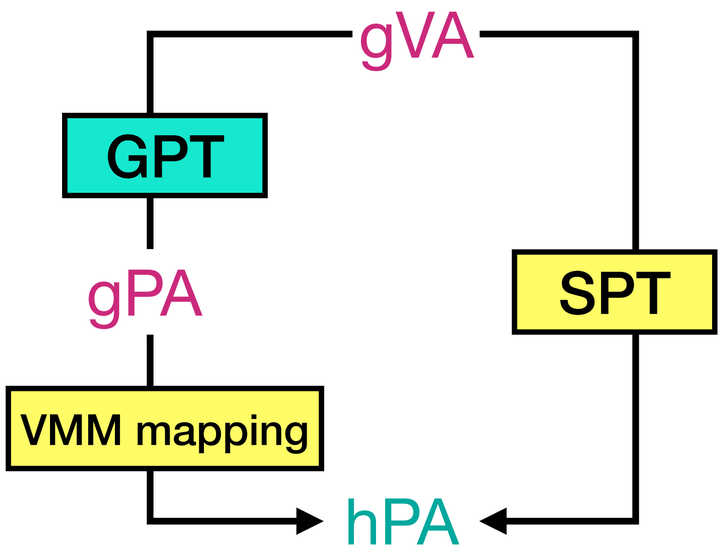


Statically Partitioning Memory

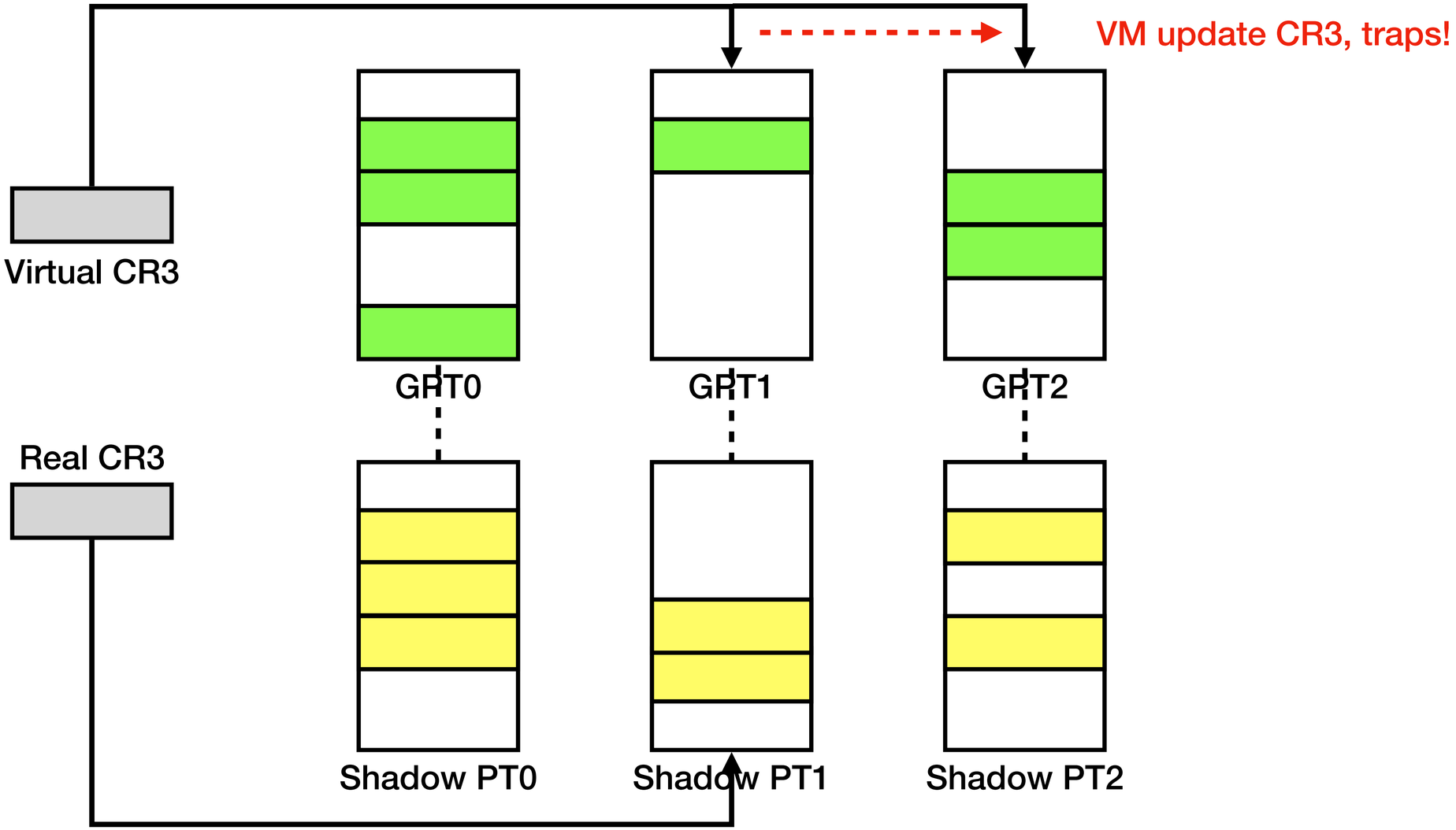
- Allocate contiguous physical memory regions to different VMs
  - Ex: Allocate hPA[0x200000:0x600000] to the VM
- Simple but inefficient — what if there are lots of VMs?
  - Cannot leverage the MMU for flexible partitioning
  - Causing fragmentation; why?

Shadow Page Tables

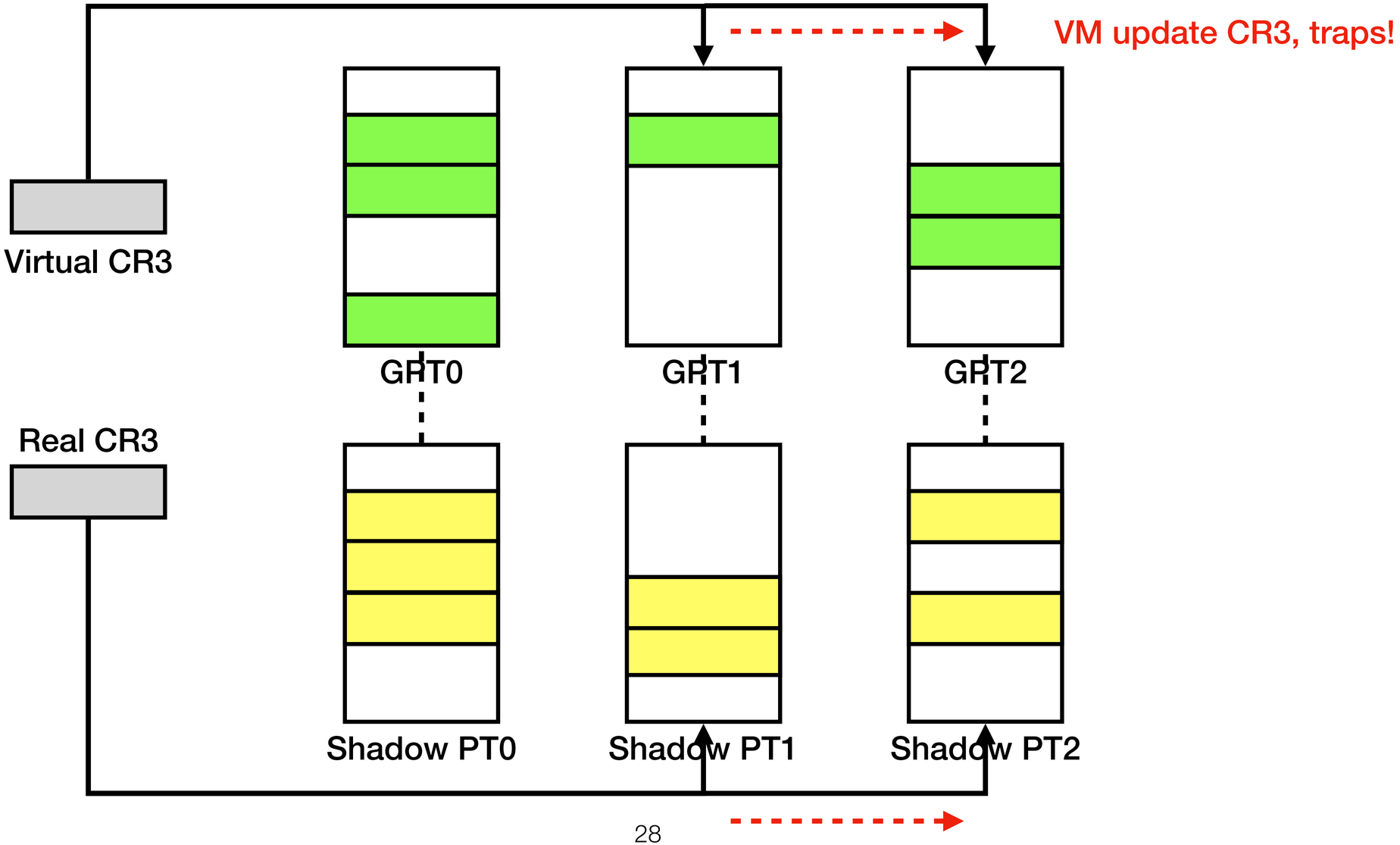
- Hypervisors maintain ***shadow page tables (SPT)*** to map gVA to hPA:
  - Invented at a time when hardware virtualization support was unavailable to leverage the existing MMU hardware
  - Extends gVA -> gPA (from GPTs) with gPA -> hPA
- SPT management:
  - VMMs allocate and maintain the SPTs
  - VMs have no access to SPTs
  - VMs cannot modify SPTs but will modify GPTs!



Shadow Page Tables: Switching GPT (1)



Shadow Page Tables: Switching GPT (2)



Shadow Page Tables: Synchronization (1)

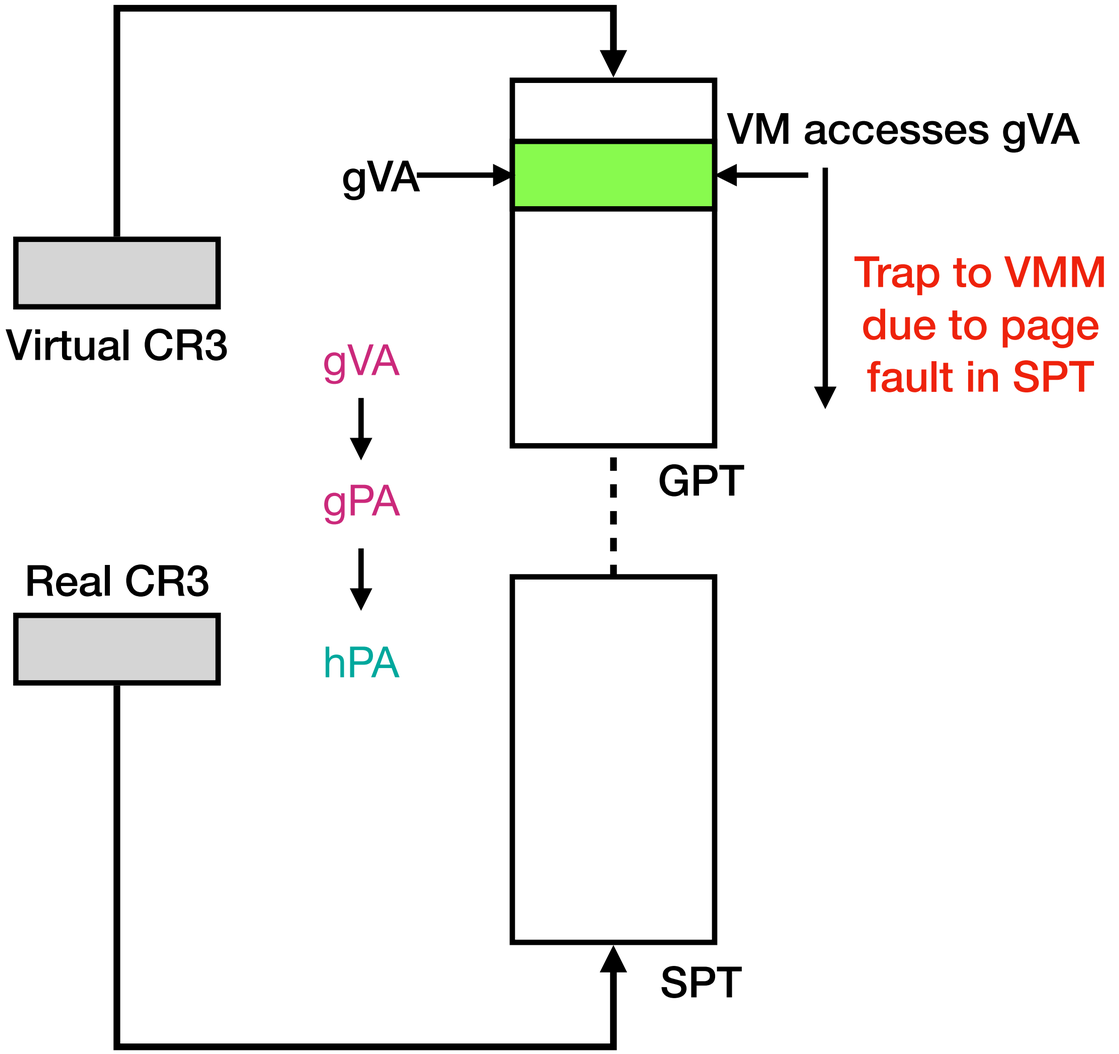
- VMMs must synchronize GPT updates to SPTs
  - A guest OS can update GPT anytime; adding new mappings is okay, but what about updating existing mappings?

Shadow Page Tables: Synchronization (2)

- VMMs must synchronize GPT updates to SPTs
  - A guest OS can update GPT anytime to
    - Add new mappings
    - Updating existing mappings?
- Strategy:
  - Make GPTs ***read-only*** so guest OS' write to GPTs (memory trace), resulting in page faults
    - How to do this?
  - VMMs handle the faults and make corresponding changes to the SPTs

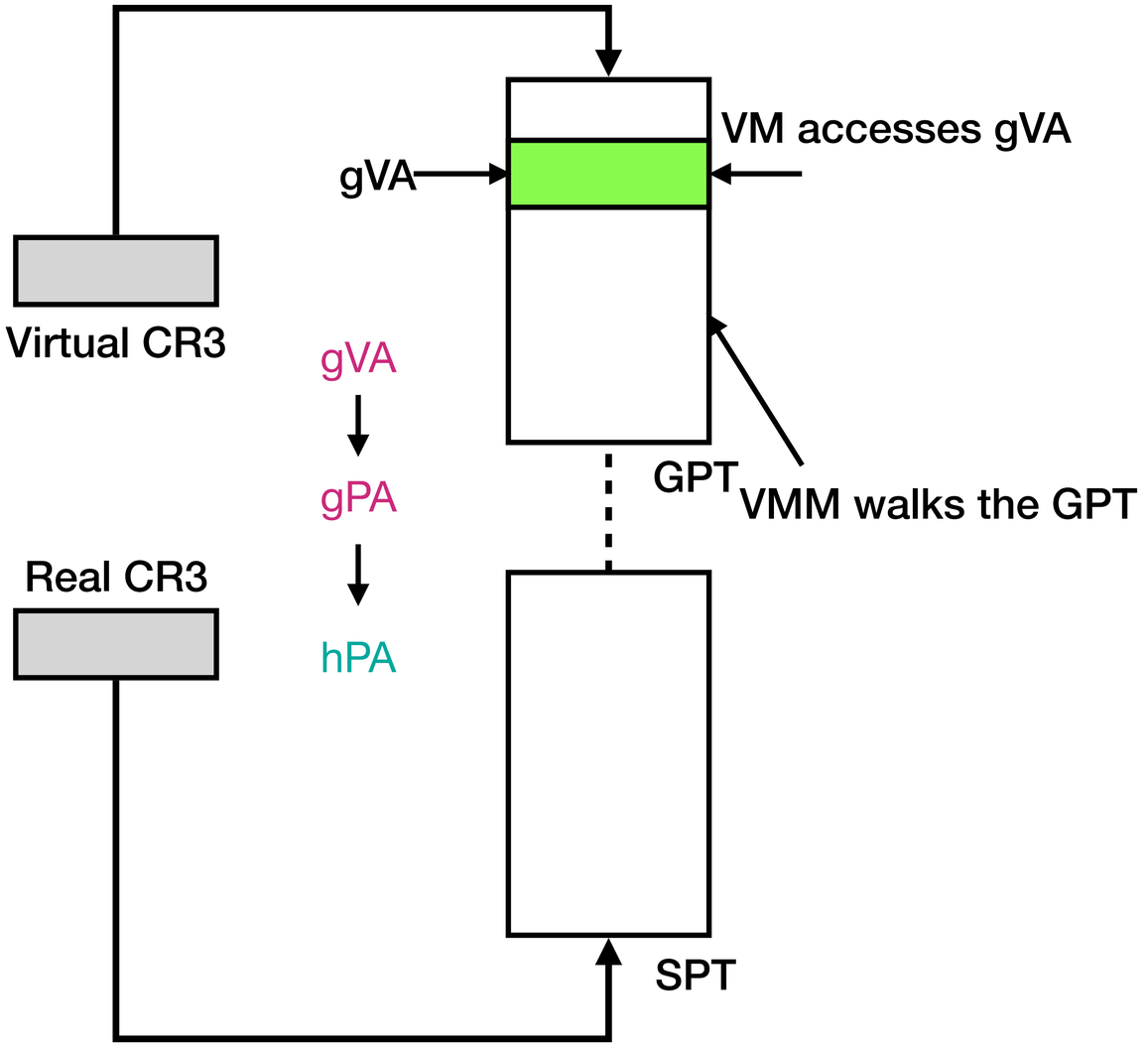
Handling shadow page faults (1)

- Similar to OS kernels, hypervisors take the demand paging approach — only allocate memory when used
- That entries may exist in GPT but not in SPT
  - VM memory accesses could cause page faults in SPT
- Hypervisors handle SPT page faults
  - Must distinguish the faults caused by virtualization from the VM's program executions; why?



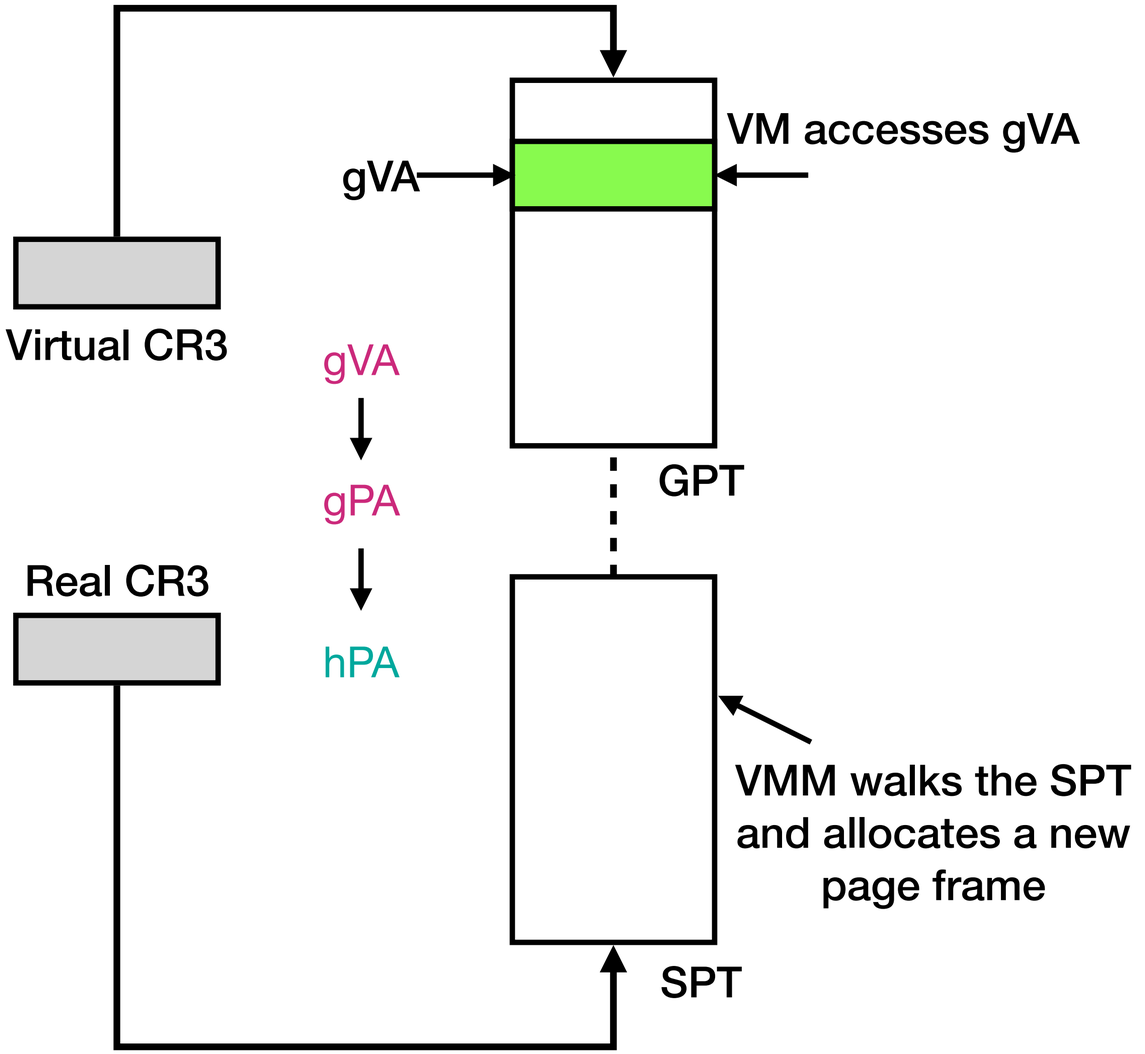
Handling shadow page faults (2)

- An SPT page fault causes a trap to the hypervisor first walks the GPT
- The hypervisor first walks the GPT to check if the gVA access is valid
  - Valid: gVA to gPA mapping exists, or the VM has the right permission to access the gVA
  - **“Inject” a page fault** to VM if the access is invalid
  - How does page fault injection work?
    - The hypervisor sets VM state to emulate what the hardware does on a page fault



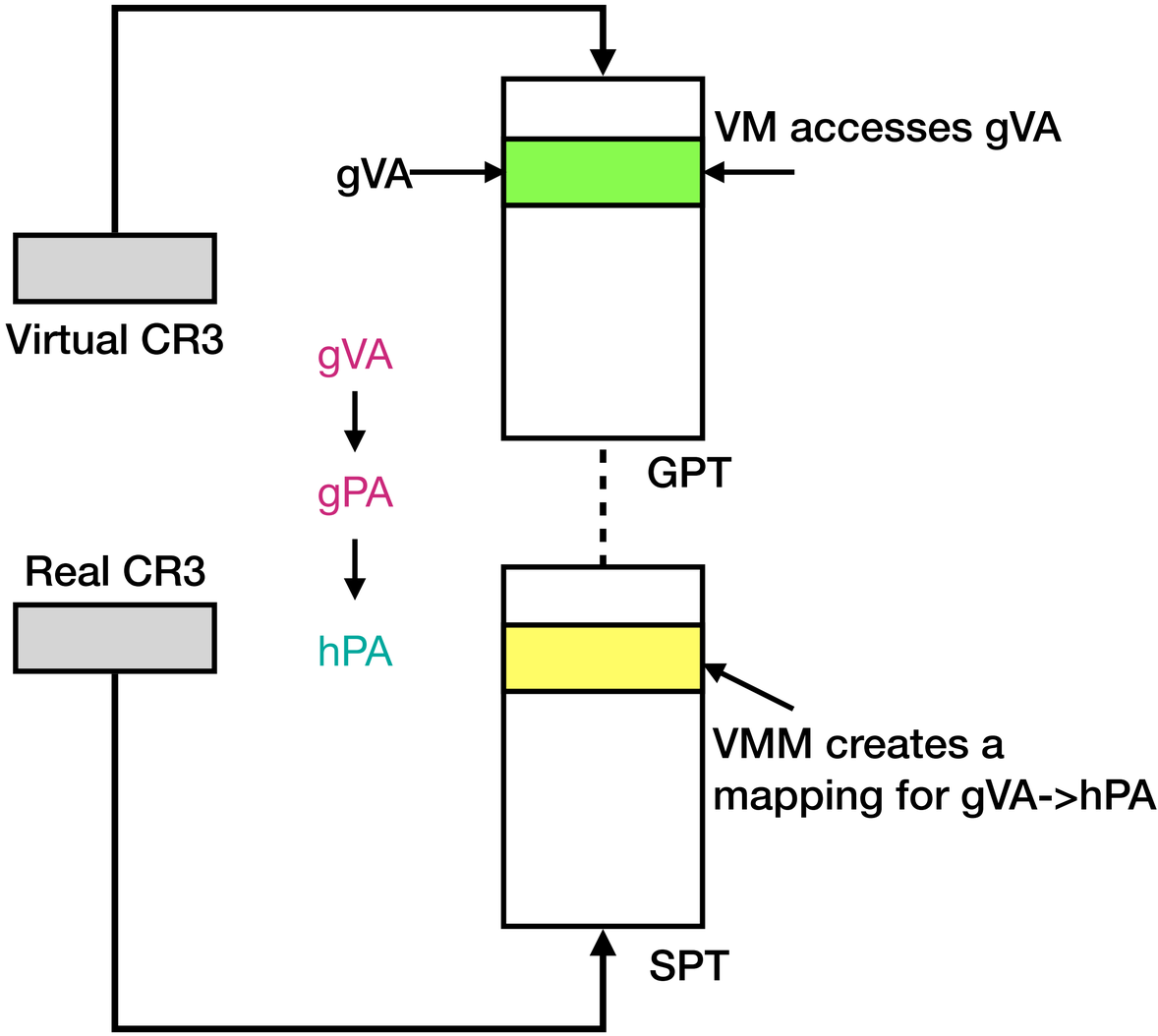
Handling shadow page faults (3)

- If gVA access is valid; the hypervisor then walks the SPT
  - In this case, the mapping for gVA does not exist in the SPT; the hypervisor adopts demand paging
  - The hypervisor allocates a new physical page frame (specified by hPA) for the gVA
  - The hypervisor maintains a gPA to hPA mapping internally

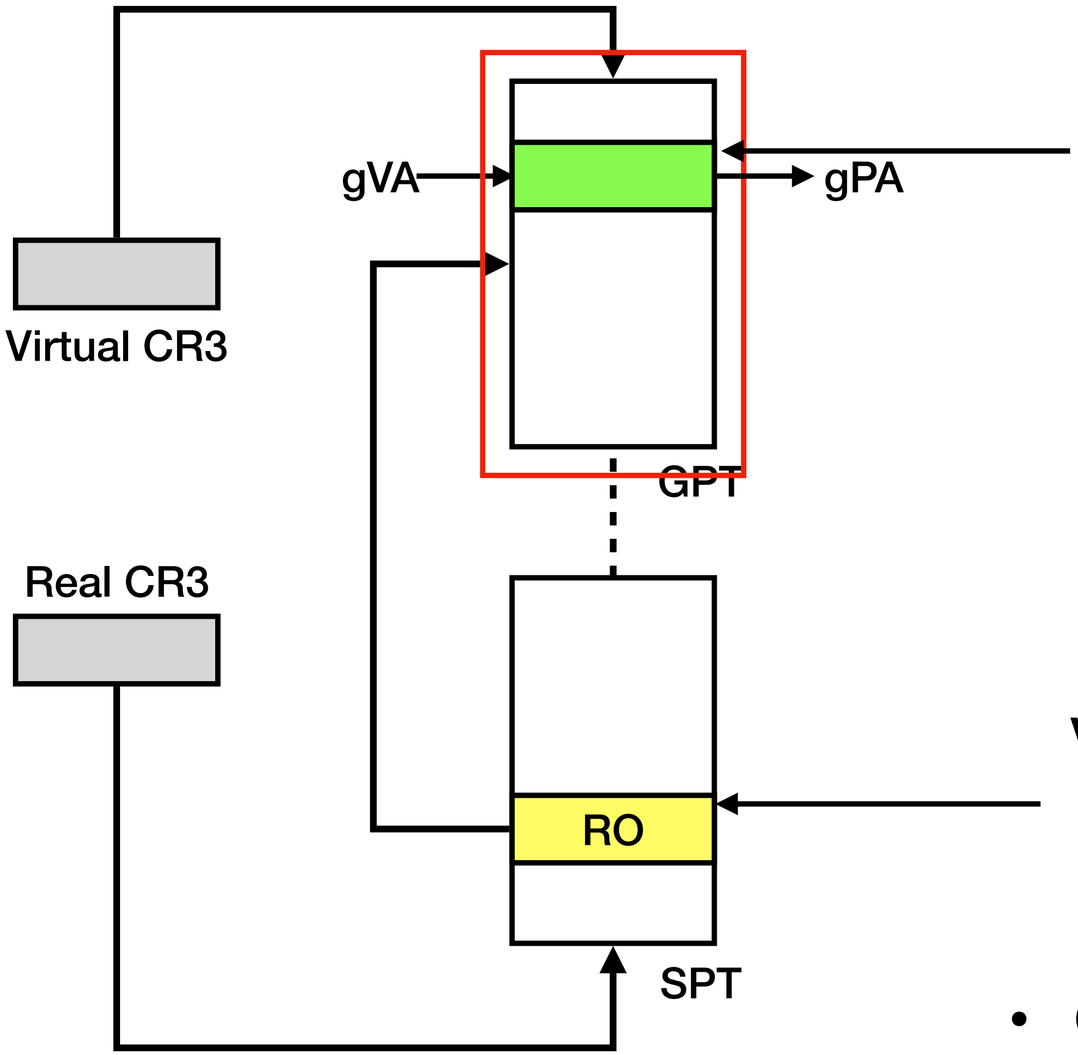


Handling shadow page faults (4)

- The hypervisor creates a mapping for gVA->hPA in the SPT
- Q: What about TLB management when handling shadow page faults?



Handling shadow page faults (5)



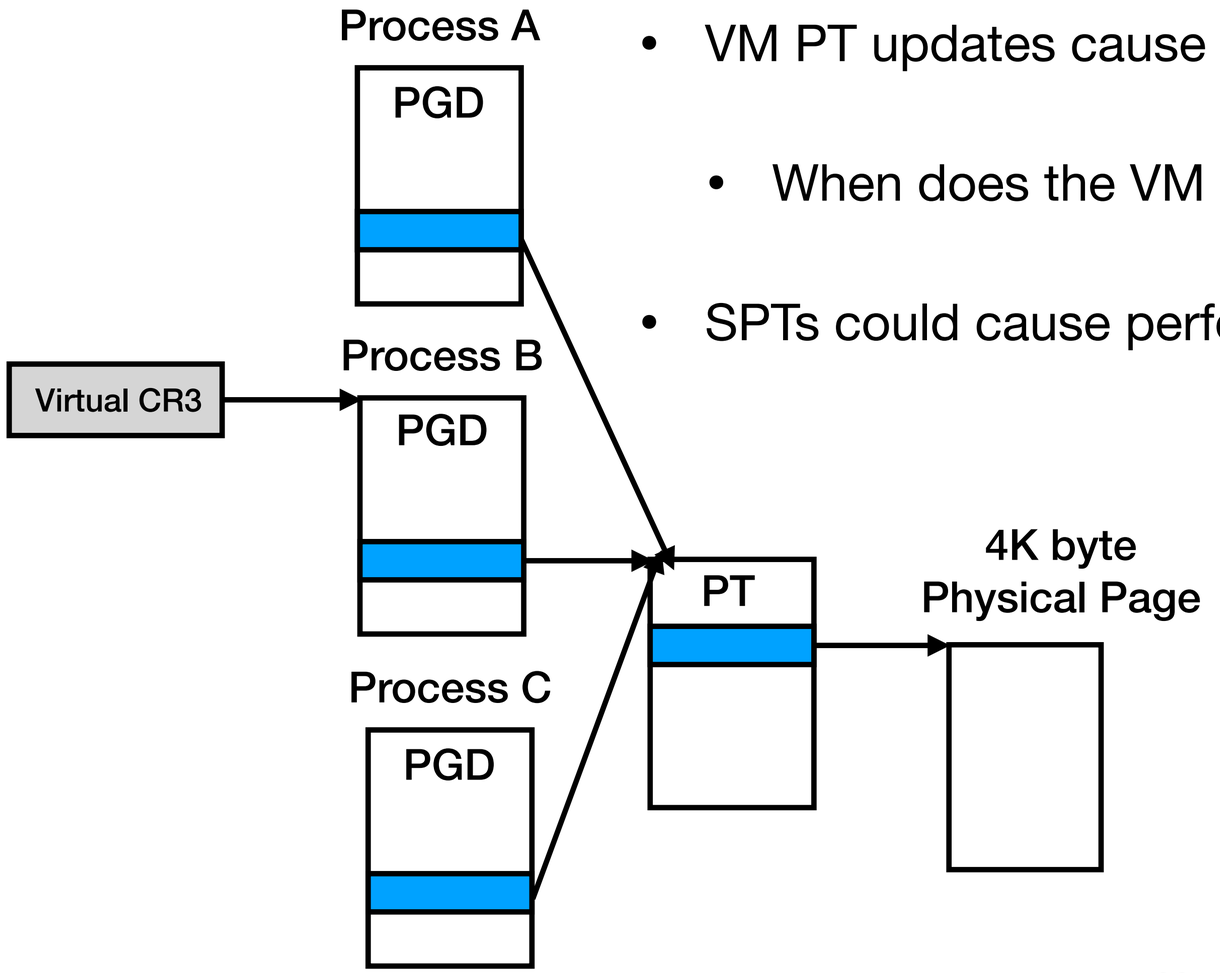
VM's write to GPT (modifies an existing gVA mapping) traps to VMM

VMM updates SPT to reflect the PT update

VMM invalidates TLB entry for gVA to hPA translations that use the GPT

- Q: TLB maintenance?

Shadow Page Tables: Issues (1)



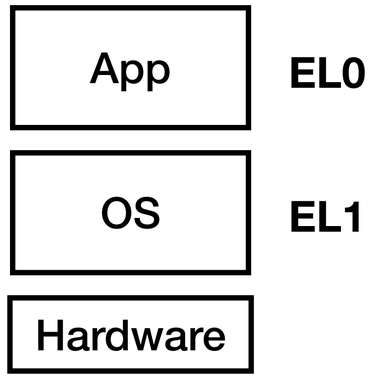
- VM PT updates cause page faults to the hypervisor
  - When does the VM update PTs?
- SPTs could cause performance overhead for processes sharing PTs; why?

Shadow Page Tables: Issues (2)

- Memory usage of shadow page tables:
  - GPTs are per process, the hypervisor manages one SPT for each GPT
  - OS manages a set of PTs for each different process — Imagine there are 10 VMs running on the machine and each runs 100 processes.

Shadow Page Tables: Issues (3): Access Permissions for SPTs

- MMU supports **user/supervisor** access permissions in page tables
  - User: user mode access; supervisor: kernel mode access



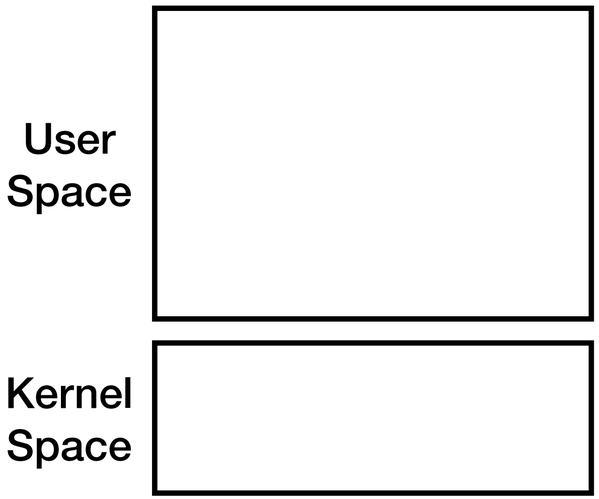
AP[2:1]	Access from EL1	Access from EL0
00	Read/write	None
01	Read/write	Read/write
10	Read-only	None
11	Read-only	Read-only

Example: Arm's memory access permissions for EL0/EL1 (from Arm Arm v8)

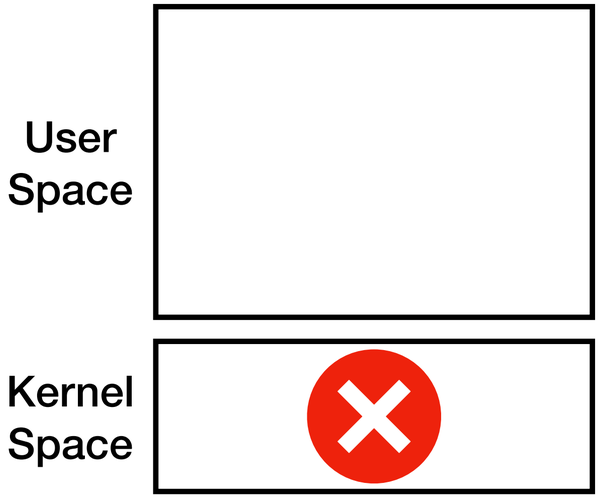
Shadow Page Tables: Issues (4): Access Permissions for SPTs

- Case Study: the VM ***physically*** runs in user mode but ***virtually*** runs in both (virtual) user and (virtual) kernel mode
  - Hypervisors must support the VM's user and kernel separation — ensure VM in user mode cannot access kernel memory
  - Solution: Updates SPT entries in VM user/kernel mode switches
    - Problem: significant performance overhead
  - Solution: For a given GPT maintains 2 SPT versions: kernel and user SPTs
    - Each SPT has different and appropriate permissions
    - Problem: lots of memory is needed for SPTs

Shadow Page Tables: Issues (5): Access Permissions for SPTs



Guest Kernel is running



Guest User is running

Shadow Page Tables: Issues (6)

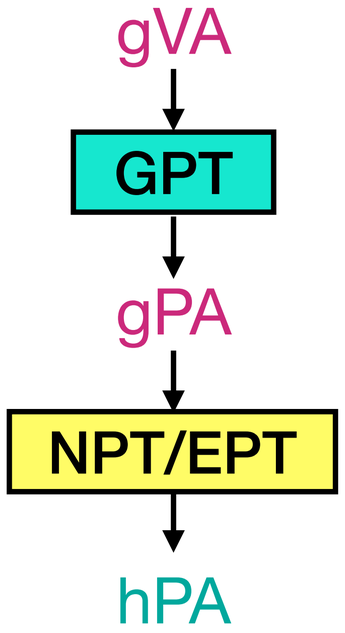
- SPTs complicate the hypervisor design:
  - How does the hypervisor know which gVAs are mapped to page tables?
  - What happens if a guest process is terminated?

Shadow Page Tables: Summary

- Pros:
  - When SPTs are built, memory translation is fast
- Cons:
  - Performance overhead due to GPTs and SPTs synchronization
  - Resource overhead
  - Hypervisor complexity: multiplex access permissions, tracking GPTs, and recycling SPTs

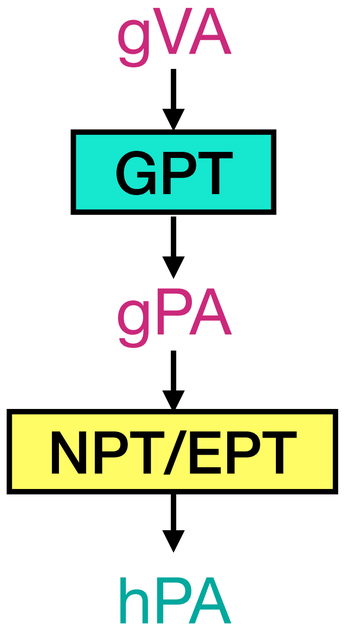
Hardware support for memory virtualization

- x86 and Arm CPUs add hardware support for memory virtualization:
  - AMD introduced Nested Page Tables (NPTs) in 2007
  - Intel introduced Extended Page Tables (EPTs) in 2008
  - Arm introduced stage 2 page tables in Arm VE

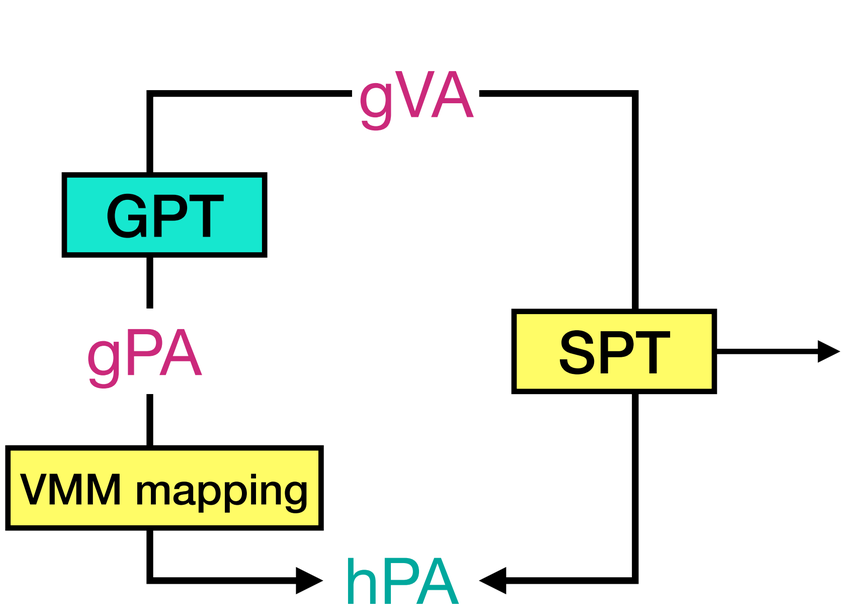


Hardware support for memory virtualization

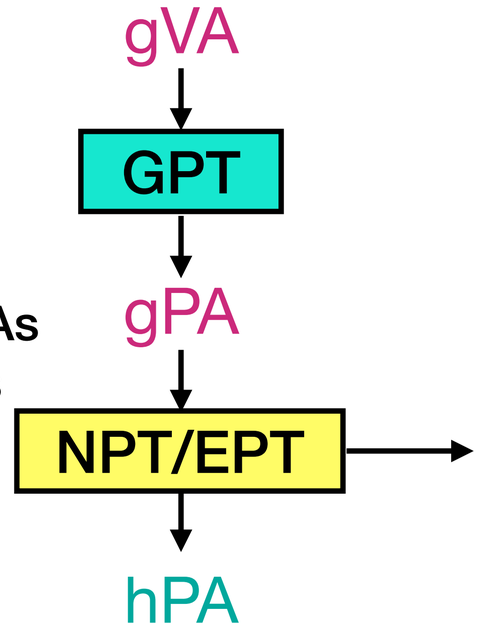
- The hardware supports Secondary Level Address Translation (SLAT):
  - SLAT incorporates a nested-level page table
  - The MMU walks both the GPT and the nested-level page tables:
    - GPT: gVA -> gPA
    - Nested-level page table: gPA -> hPA
  - Hypervisors manage one nested-level page table (NPT/EPT) for each VM



Shadow PT vs NPT/EPT



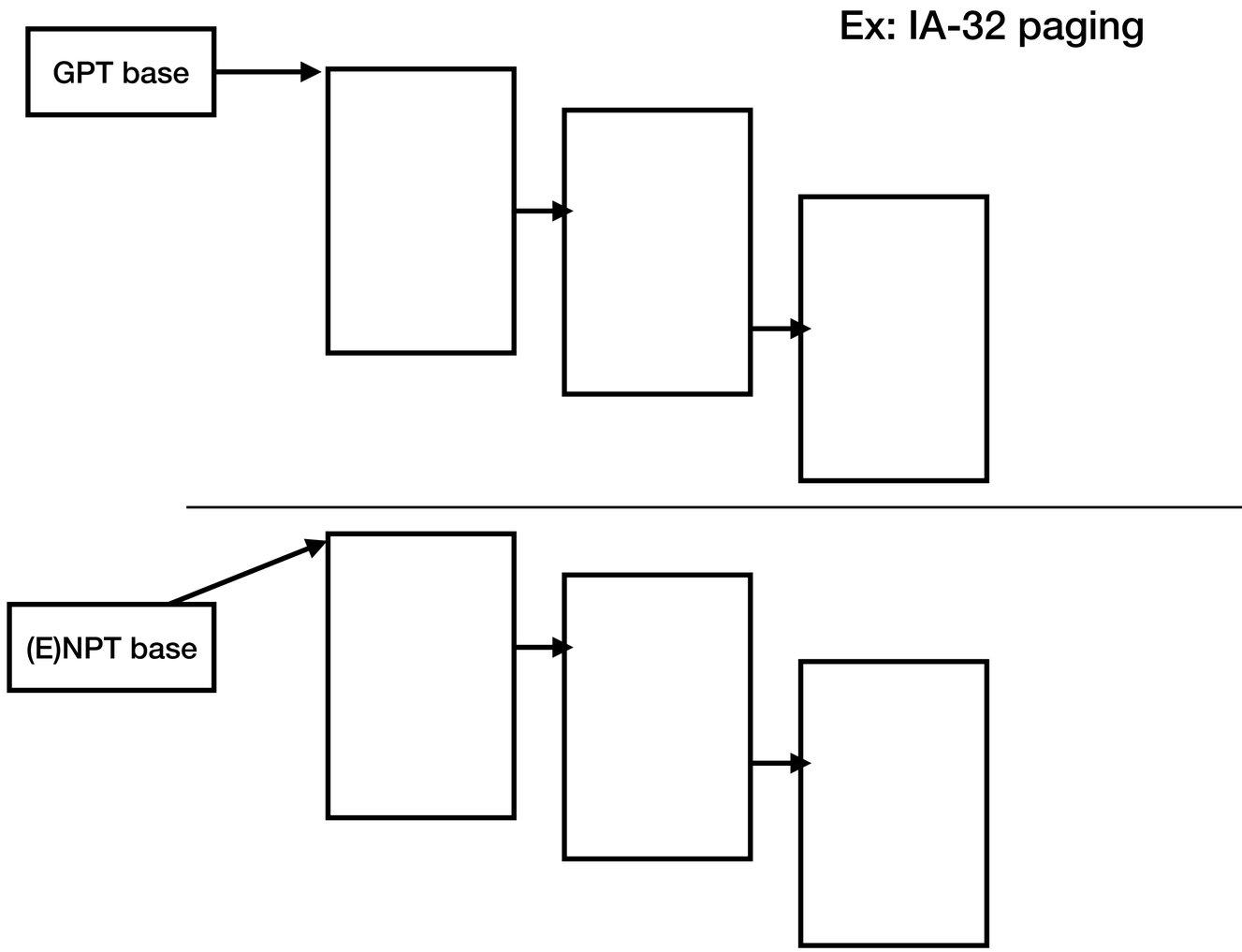
Example: map gVAs in [0:4G] to hPAs



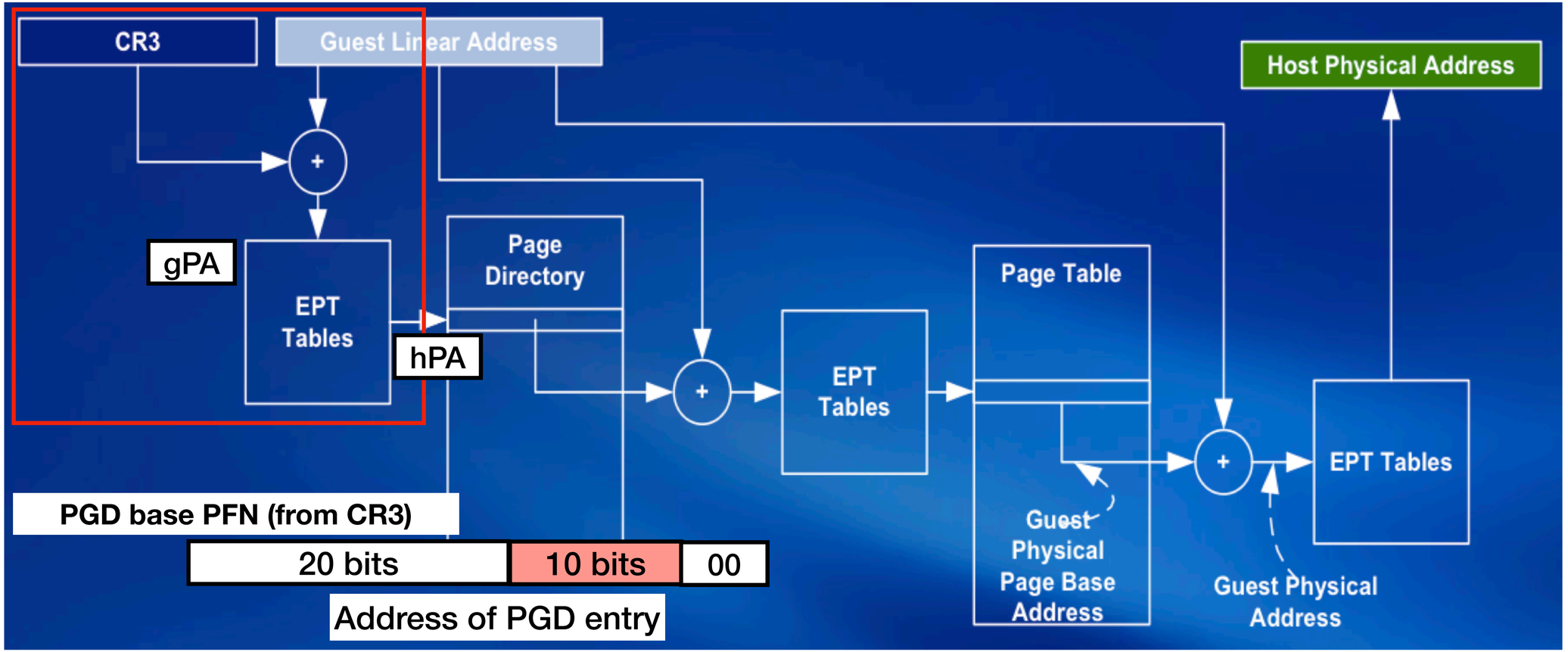
Example: map gPAs in [0:1G] to hPAs

Nested Paging

- The MMU does not understand gPAs; it only understands hPAs
  - GPTs are built with gPAs
  - EPTs/NPTs are built with hPAs
- How does the MMU walk the GPTs?

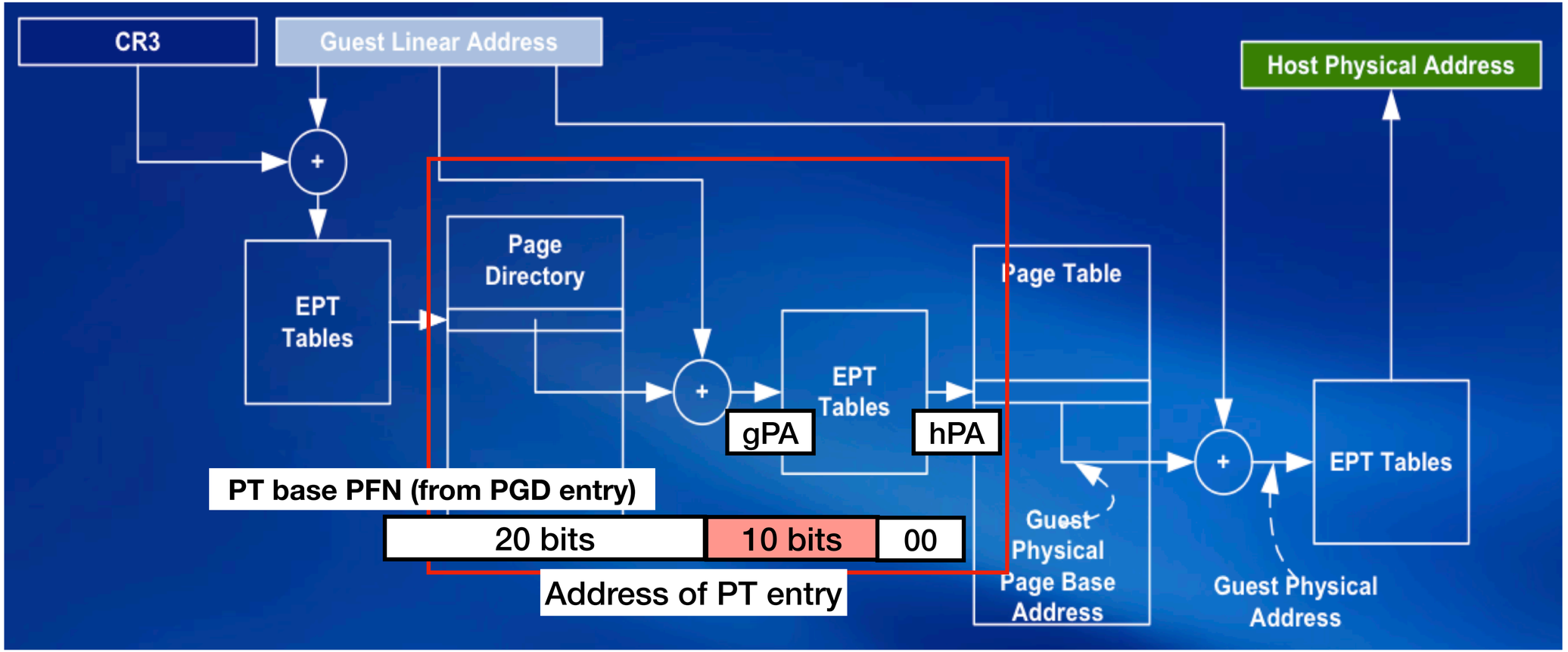


Case Study: EPT Translation (1)



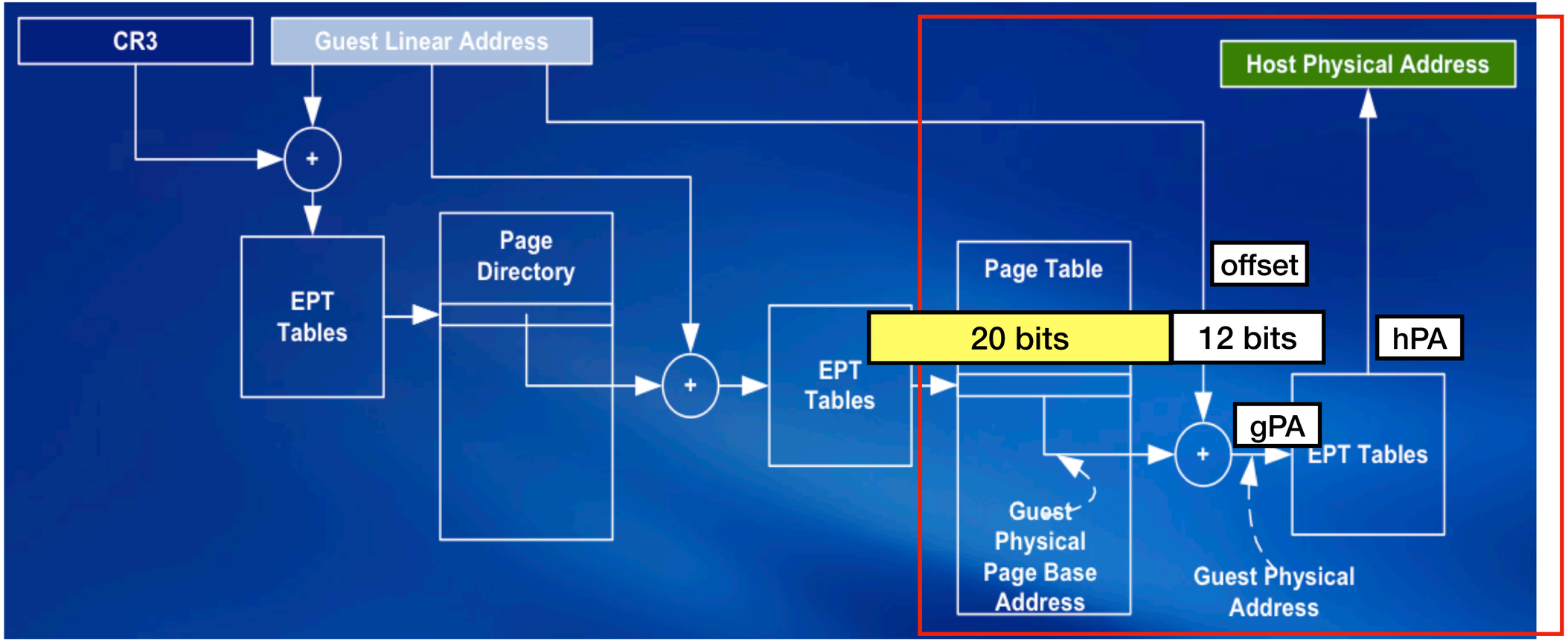
Modified from Yiying Zhang's slides
https://cseweb.ucsd.edu/~yiying/cse291j-winter20/reading/Virtualize-Memory.pdf

Case Study: EPT Translation (2)



Modified from Yiying Zhang's slides
https://cseweb.ucsd.edu/~yiying/cse291j-winter20/reading/Virtualize-Memory.pdf

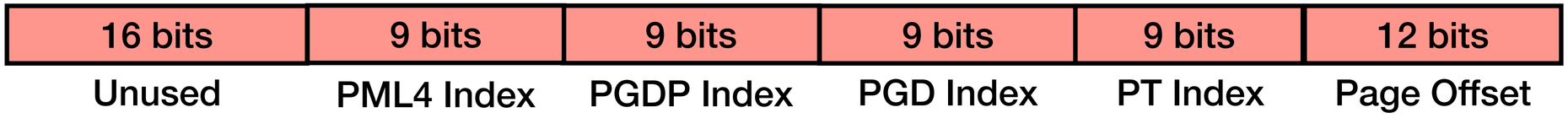
Case Study: EPT Translation (3)



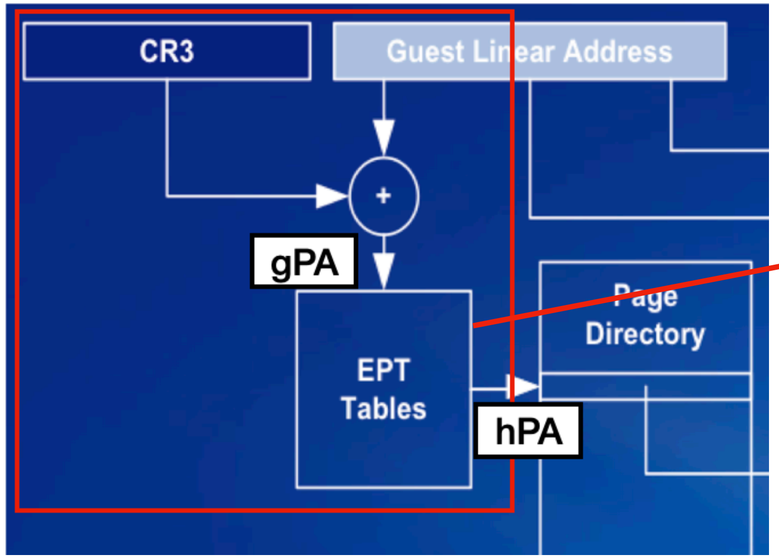
Modified from Yiying Zhang's slides
https://cseweb.ucsd.edu/~yiying/cse291j-winter20/reading/Virtualize-Memory.pdf

Case Study: IA-64 Page Table

- IA-64 page tables include 4 levels
  - Each page table is 4KB
- Supports 1GB, 2MB, and 4KB page table mappings
  - Page table entries are 64-bit, includes 512 entries
  - Use 2 levels for 1GB, 3 levels for 2MB, and 4 levels for 4K mappings



Nested paging with IA-64 page tables (1)



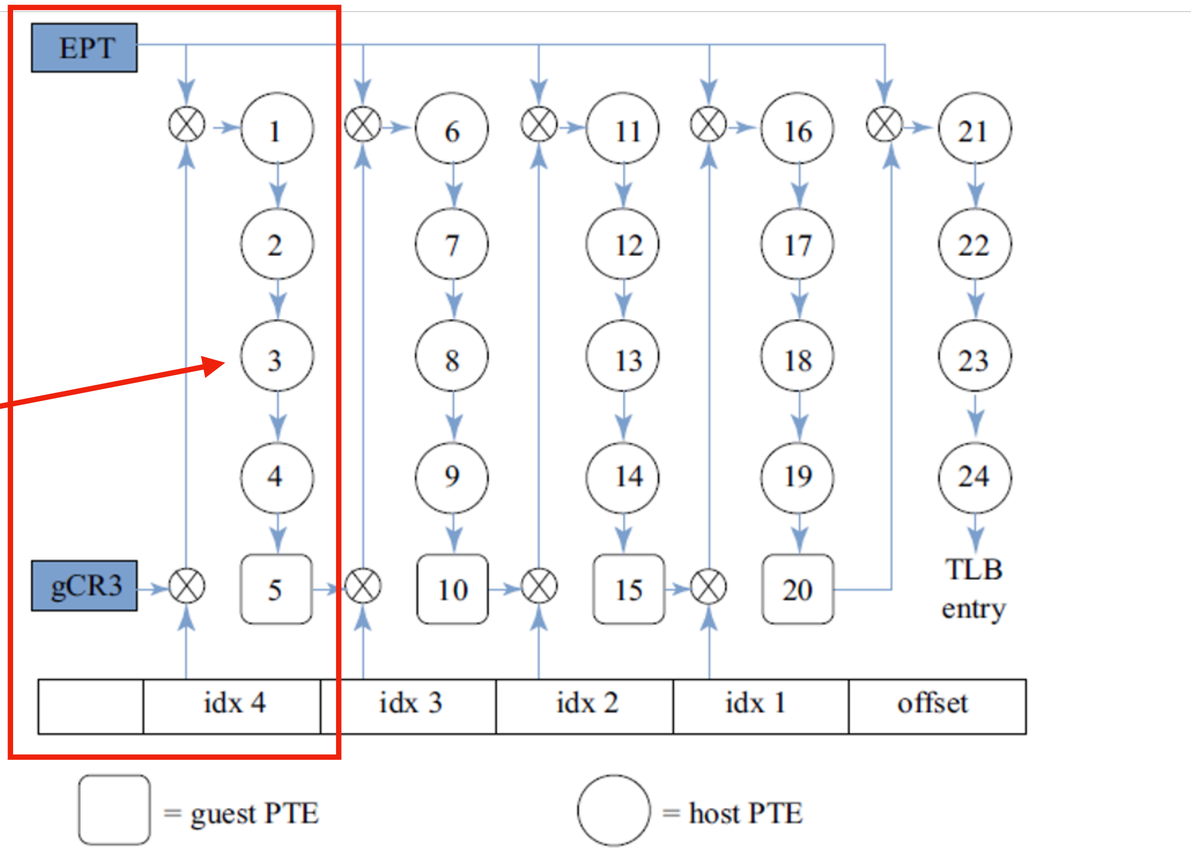


Figure 5.1: Sequence of architecturally-defined memory references to load a TLB entry with extended page tables ($m = 4$ and $n = 4$).

Nested paging with IA-64 page tables (2)

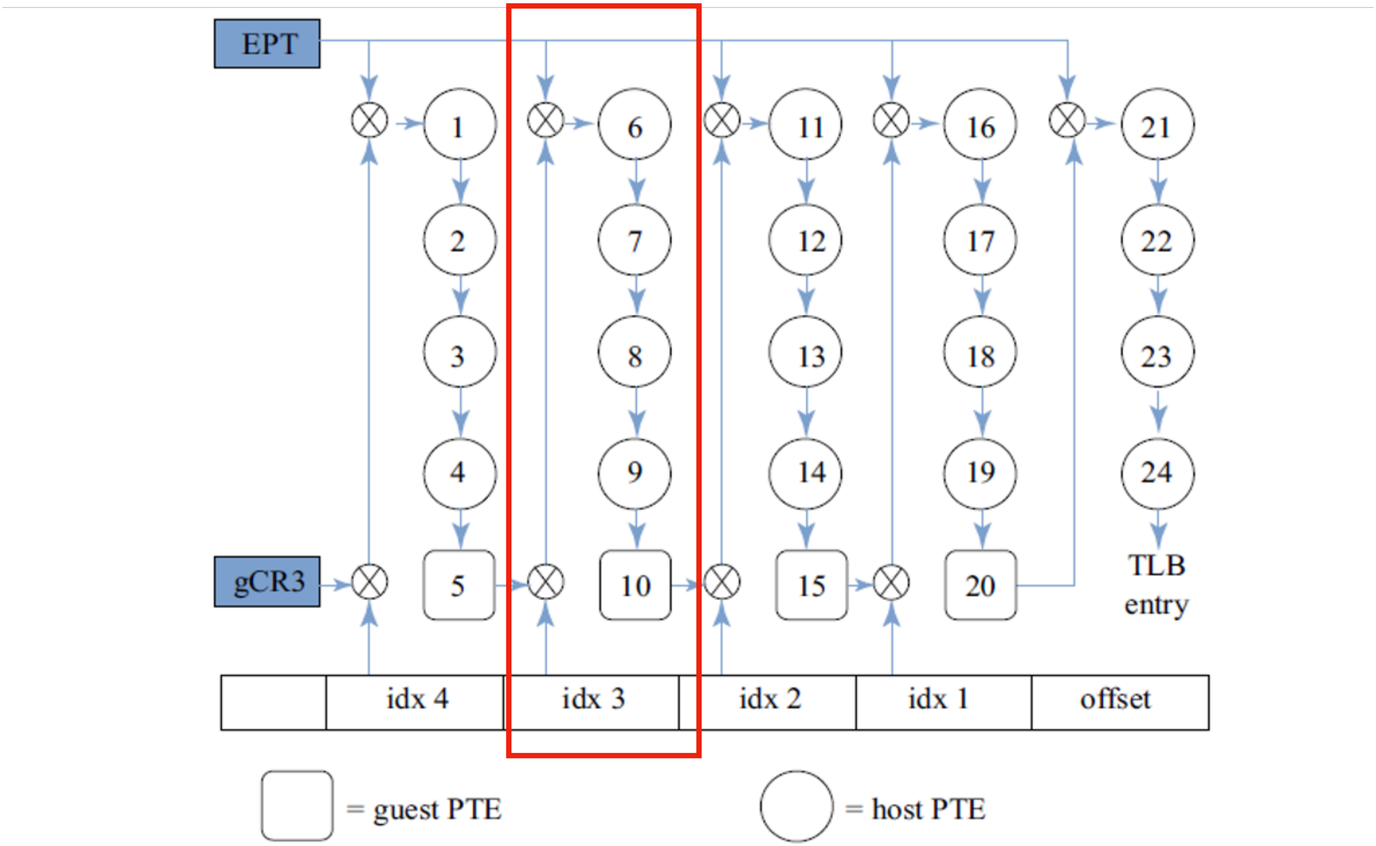


Figure 5.1: Sequence of architecturally-defined memory references to load a TLB entry with extended page tables ($m = 4$ and $n = 4$).

Nested paging with IA-64 page tables (3)

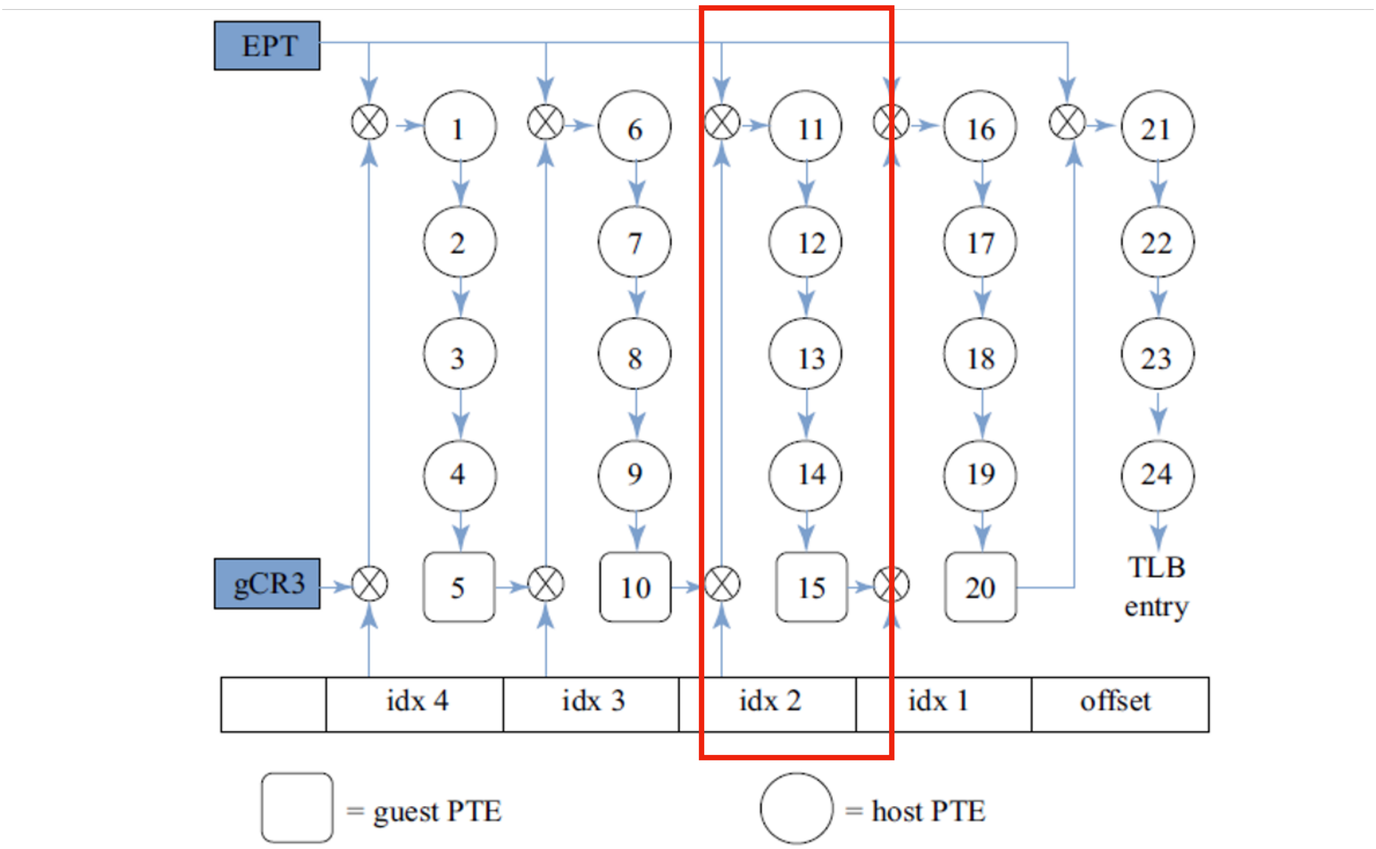


Figure 5.1: Sequence of architecturally-defined memory references to load a TLB entry with extended page tables ($m = 4$ and $n = 4$).

Nested paging with IA-64 page tables (4)

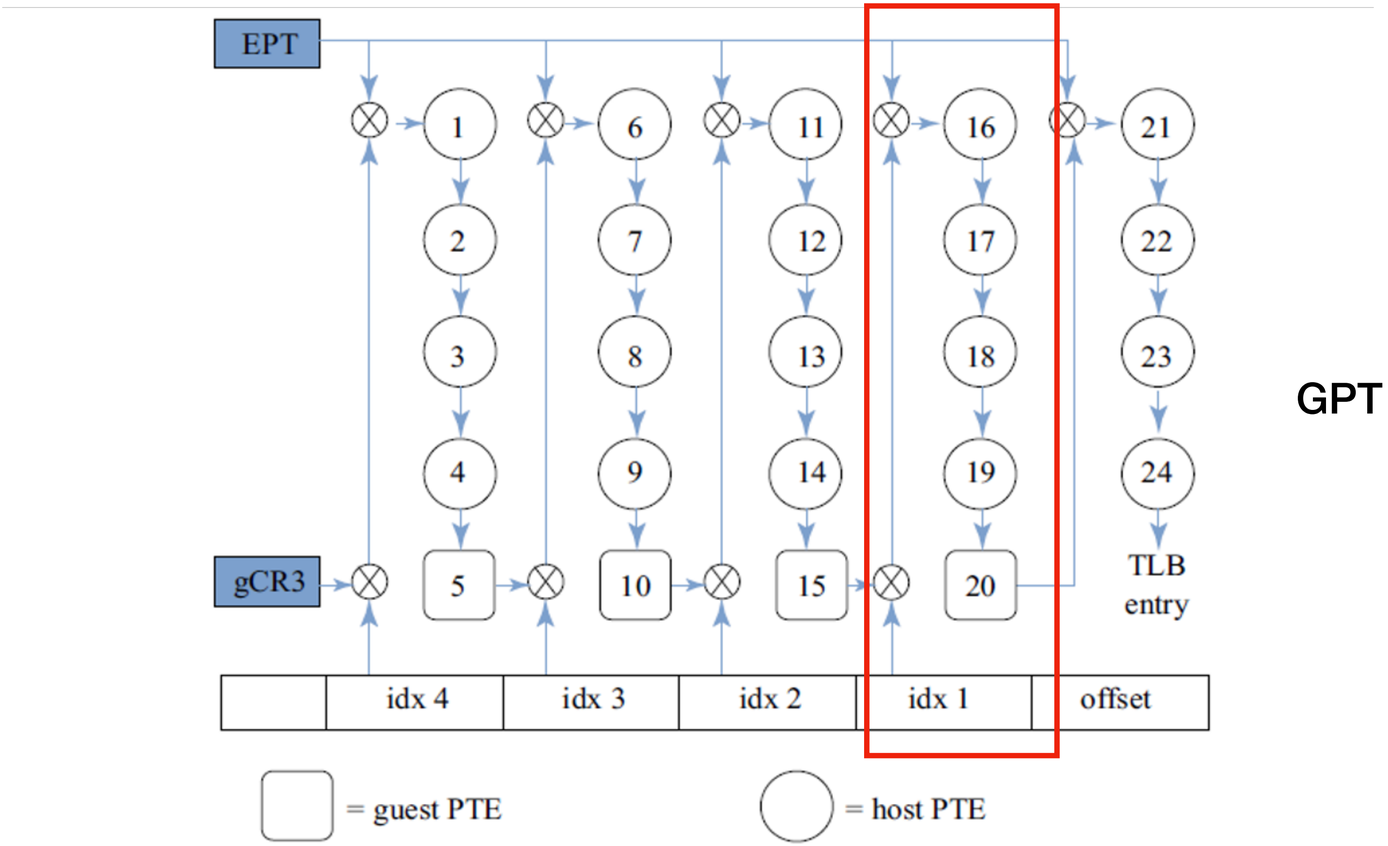


Figure 5.1: Sequence of architecturally-defined memory references to load a TLB entry with extended page tables ($m = 4$ and $n = 4$).

GPT walk is done; we got the gPA; Are we done?

Nested paging with IA-64 page tables (5)

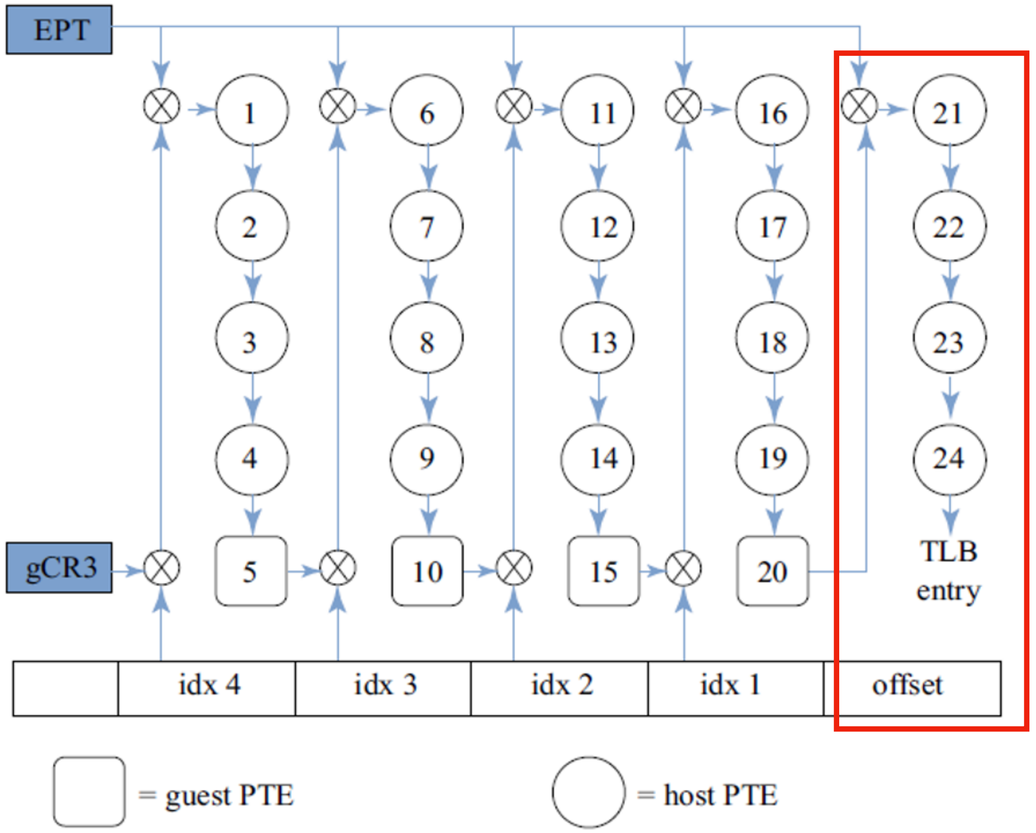


Figure 5.1: Sequence of architecturally-defined memory references to load a TLB entry with extended page tables ($m = 4$ and $n = 4$).

GPT walk is done; we got the gPA; Are we done? No, we still need to walk the EPT to translate gPA => hPA

TLB miss when using NPRs/EPTs

- TLB caches the gVA to hPA mapping, on a TLB miss:
  - The hardware MMU walks each level of a GPT starting from the GPT root
  - MMU walks each level of the GPT to translate the gPA of the page table to hPA before dereferencing page table entries
  - Assume GPT has **N** levels; EPT has **M** levels
  - **N** * **M** + **N** + **M** memory accesses are required — why?
  - In contrast, only **X**(SPT level #) memory accesses are needed for SPT

Memory accesses during EPT Walk (1)

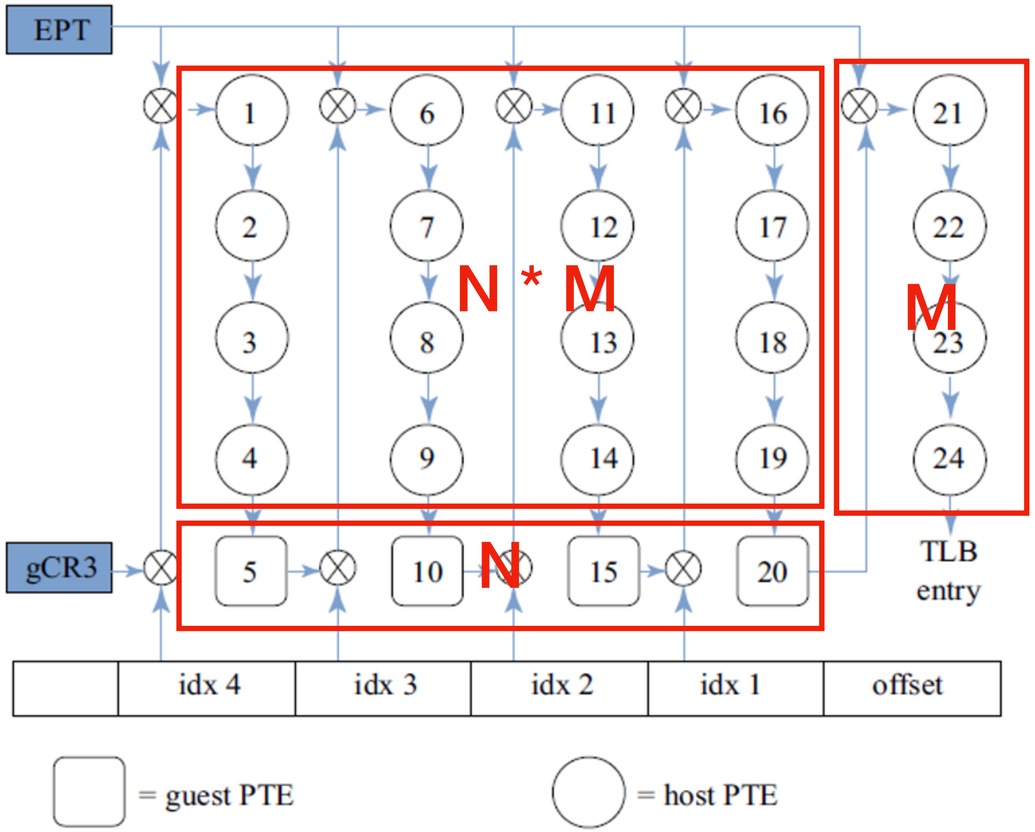
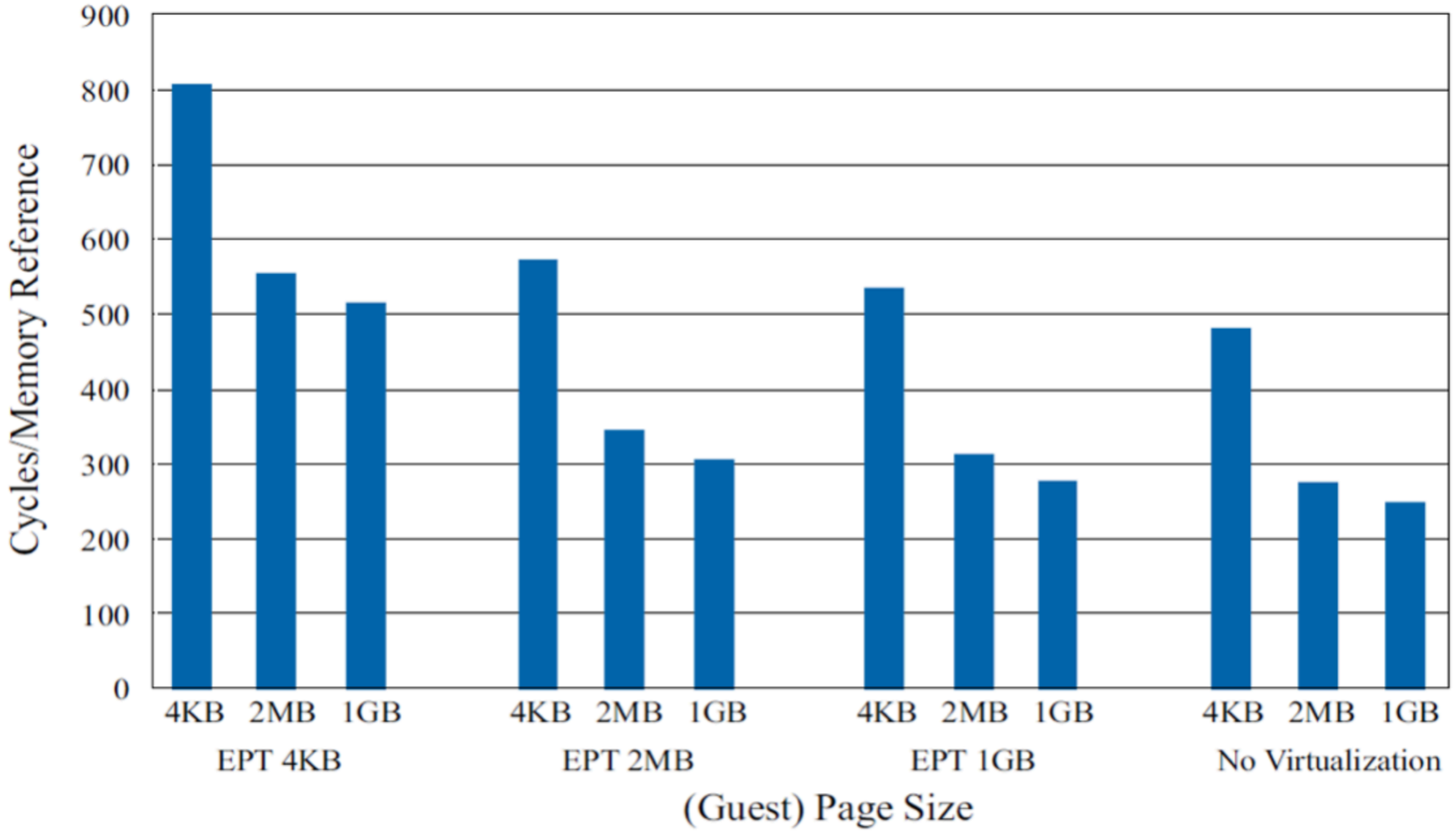
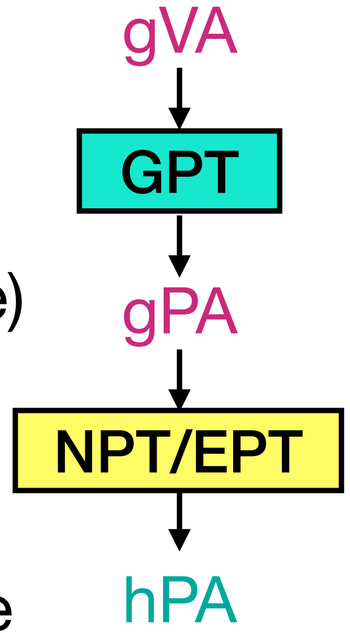


Figure 5.1: Sequence of architecturally-defined memory references to load a TLB entry with extended page tables ($m = 4$ and $n = 4$).

EPT Walk Cost on Sandy Bridge

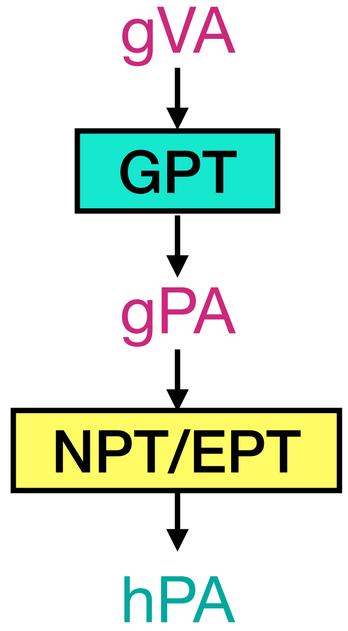


Summary: Hardware support for memory virtualization (1)



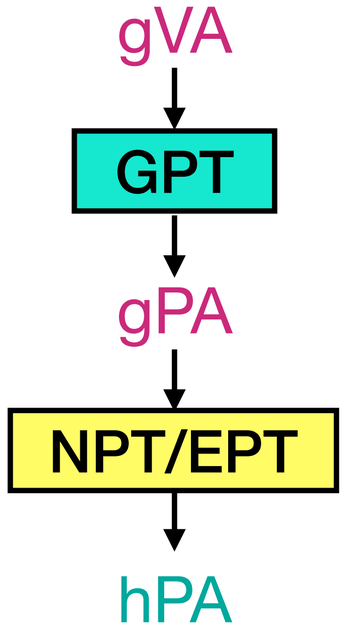
- The MMU hardware walks two sets (GPT + NPT) of page tables
- VMMs manage one NPT for each VM (one set of gPA address space)
- NPT simplifies the VMM design significantly:
  - Avoid switching page tables when VM updates to page table base register — only have to do so when switching VMs
  - No GPT walk is required when handling NPT page faults
  - No need to synchronize GPTs with NPT

Summary: Hardware support for memory virtualization (2)



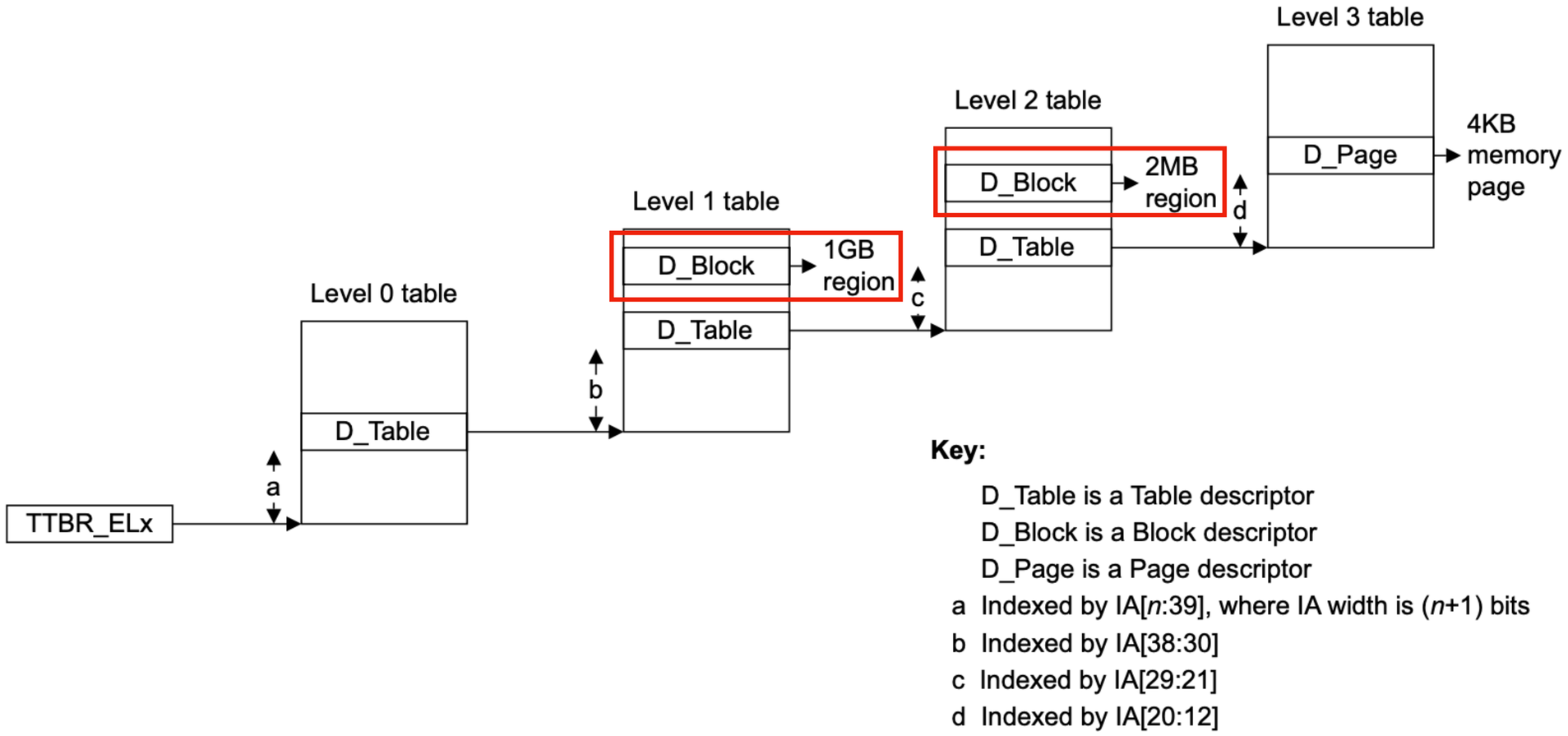
- Pros
  - NPTs provide better resource efficiency and performance
  - NPTs simplify hypervisor complexity
- Cons:
  - TLB miss could be costly due to the extra page table walks

Summary: Hardware support for memory virtualization (3)

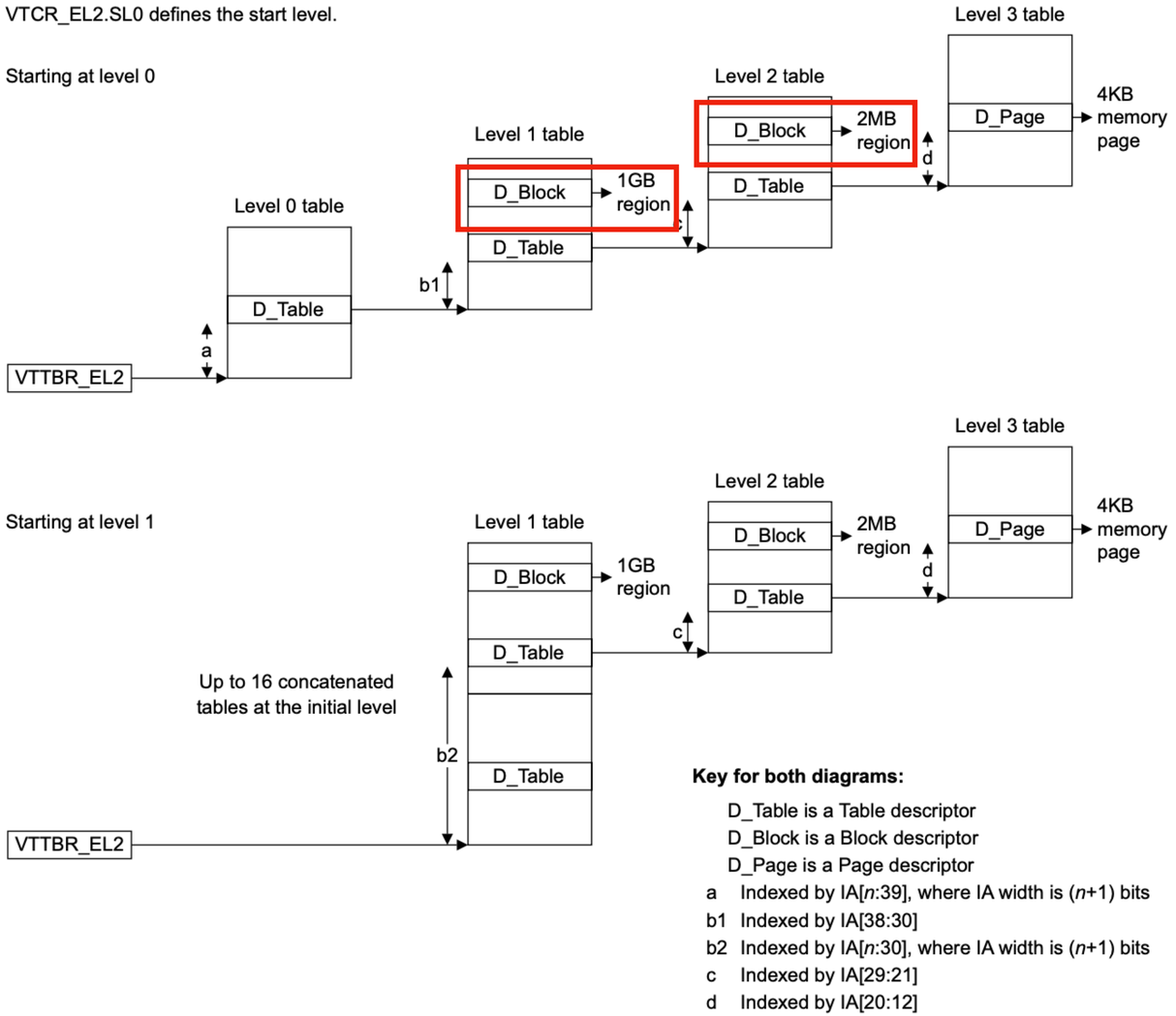


- Question: when are NPT/EPT updates necessary?

Case Study: Arm page tables (stage 1)



Case Study: Arm page tables (stage 2)



Case Study: Arm page tables - summary

- NPTs in Arm are called stage-2 page tables (or translation tables)
  - Support nested paging similar to NPTs/EPTs
- Stage-2 page tables were introduced as part of Arm VE
  - Arm added a new register VTTBR_EL2 to points to the root of the stage-2 page tables

NPT Walk Cost Complication

- Using larger page size in EPTs result in better performance
  - Why? Fewer page table levels involved during page table walk
  - Better TLB performance — can cache more large page entries
- Hypervisor and OS adopt huge page (2MB) mappings
  - Strategy: use 2MB pages in EPT by default; only use 4KB under memory pressure
  - Do you need to dissolve the mappings?

Architected Page Tables v.s. Architected TLB

- Architected Page Tables: the hardware translates memory using both TLBs and page tables
  - TLBs are used to cache page table mappings (like the example we discussed earlier)
  - Included in IA-32/64, Arm; commonly used by VMs nowadays
  - TLBs are transparent to software and managed by hardware
  - Software manages page tables; flushes TLBs to ensure coherence
- Architected TLBs: the hardware translates memory only using the TLBs
  - Included in MIPS
  - Software flushes and reloads the TLBs; maintains page tables for bookkeeping

Virtualizing Memory — TLBs (1)

- We focus on architected page tables
- TLBs cache page table translations
  - Caches translations for first, second, or both level
- Challenges: the TLB must distinguish translations between different VMs
  - How does the TLB distinguish one VM's translations from others'?

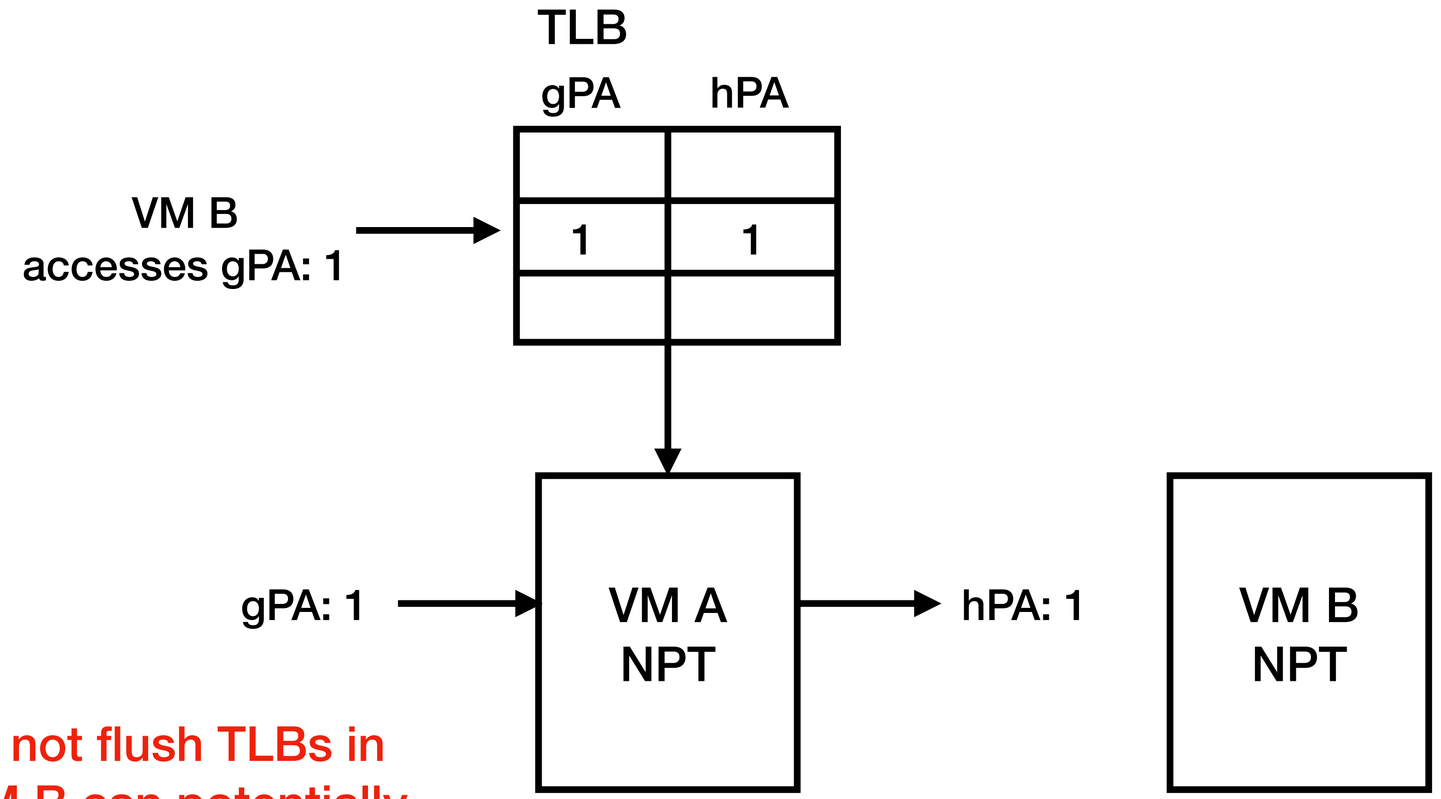
Virtualizing Memory — TLBs (2)

- Naive approach: flush all TLB entries when context-switching VMs
- Better approach: disambiguate TLB entries using TLB tags
  - Shadow page tables: the hypervisor assigns a unique ASID for each different shadow page table
    - Problem: hardware ASIDs are limited:
      - Intel's PCID has 12 bits, and Arm's ASID has 8 bits
      - What if hardware ASIDs are used up?
  - How about SLATs (EPTs/NPTs/Stage-2 page tables)?

Tagging TLBs for nested paging (1)

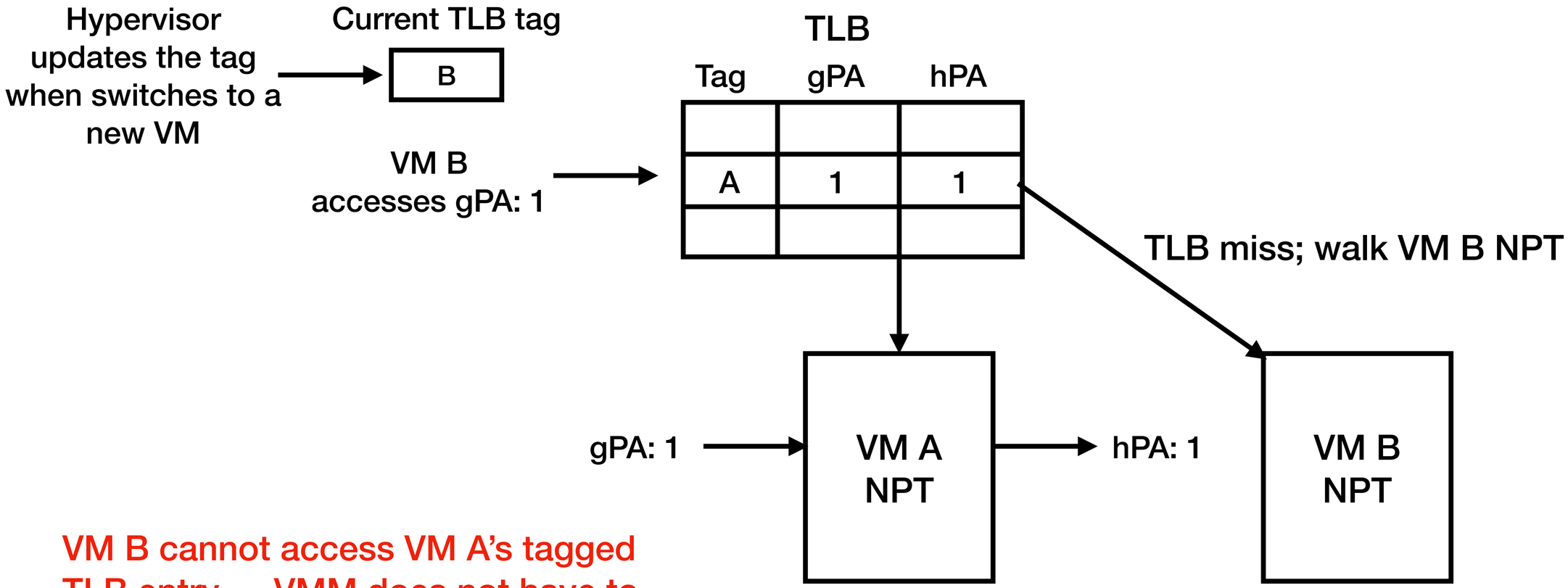
- Intel VT-x includes Virtual Processor Identifier (***VPID***):
  - Without VPID, first generation of Intel VT-x forces TLB flushes on each VMX transitions (VM enter/exit) — costly!
  - Includes 16-bit VPID field in VMCS
  - Tag translations using the EPT to avoid TLB conflicts
- Arm VE provides ***VMID*** to tag stage 2 page tables

Tagging TLBs for nested paging (2)



If the hypervisor does not flush TLBs in VM switches; here, VM B can potentially access VM A's cached TLB entry

Tagging TLBs for nested paging (3)



VM B cannot access VM A's tagged TLB entry — VMM does not have to flush TLBs in VM switches

Managing NPTs and the TLB

- For a newly created VM, the hypervisor:
  - Allocates a new NPT root and VM identifier (ID); the latter is to tag the TLBs
- The new NPT contains few mappings initially (to firmware/boot loader)
  - Why? Hypervisors usually employ demand paging
    - Trap NPT faults to the hypervisor when accessing unallocated gPAs
    - Hypervisors handle nested page faults similarly to shadow page faults
- gPA to hPA mappings might not be static; why?

Shadow PTs and NPT/EPT

	Shadow PT	Nested/Extended PT
Complexity in Software Implementation	High	Low
Required Memory Resource	High	Low
Page Table Walk Overhead	Low	High