CS 423
Operating System Design: Virtualizing CPU and Memory

Tianyin Xu

The Simplest Idea

- To run a virtual machine on top of a hypervisor, the basic technique that is used is **limited direct execution** – when we wish to “boot” a new OS on top of the VMM, we simply jump to the address of the first instruction and let the OS begin running.
- **What are the problems you can think about?**

Privileged Operations

- What if a running application or OS tries to perform privileged operations?
  - Update TLB (assuming a SW-managed TLB)
  - (Guest) OS is no longer the boss anymore.
  - VMM must somehow intercept attempts to perform privileged operations and thus retain control of the machine.

Privileged Operations

- Privileged Operations are supposed to be done through **System Calls**
  - Interrupt/trap
- **Interrupt/trap handlers**
  - OS, when it is first starting up, establishes the address of such a routine with the hardware.

Normal Case

Process	Hardware	Operating System
1. Execute instructions (add, load, etc.)		
2. System call: Trap to OS		
	3. Switch to kernel mode; Jump to trap handler	
		4. In kernel mode; Handle system call; Return from trap
	5. Switch to user mode; Return to user code	
6. Resume execution (@PC after trap)		

Virtualized Case

- What should happen?
  - VMM should controls the machine
  - VMM should install a trap handler that will first get executed in kernel mode.
- **VMM need handle this system call?**
  - The VMM doesn’t really know how to handle the call; after all, it does not know the details of each OS that is running and therefore does not know what each call should do.

How to handle System Call?

- What should happen?
  - VMM should controls the machine
  - VMM should install a trap handler that will first get executed in kernel mode.
- **VMM need handle this system call?**

How to handle System Call?

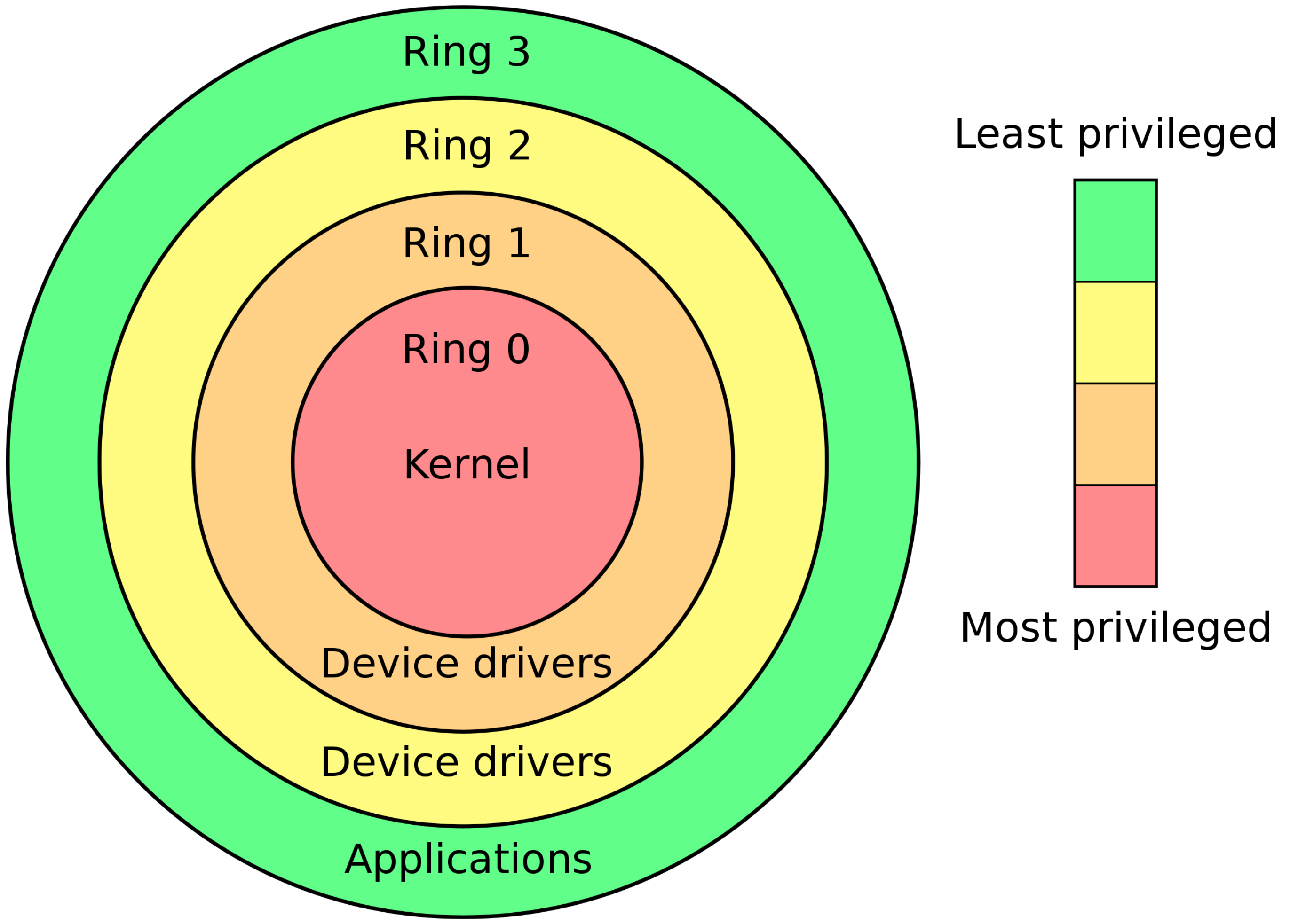
- What the VMM does know, however, is where the OS's trap handler is.
  - When the OS booted up, it tried to install its own trap handlers;
  - It is privileged, and therefore trapped into the VMM;
  - The VMM recorded the necessary information (i.e., where this OS's trap handlers are in memory).

Process	Operating System
1. System call: Trap to OS	
	2. OS trap handler: Decode trap and execute appropriate syscall routine; When done: return from trap
3. Resume execution (@PC after trap)	

Process	Operating System	VMM
1. System call: Trap to OS		
		2. Process trapped: Call OS trap handler (at reduced privilege)
	3. OS trap handler: Decode trap and execute syscall; When done: issue return-from-trap	
		4. OS tried return from trap: Do real return from trap
5. Resume execution (@PC after trap)		

How about protection?

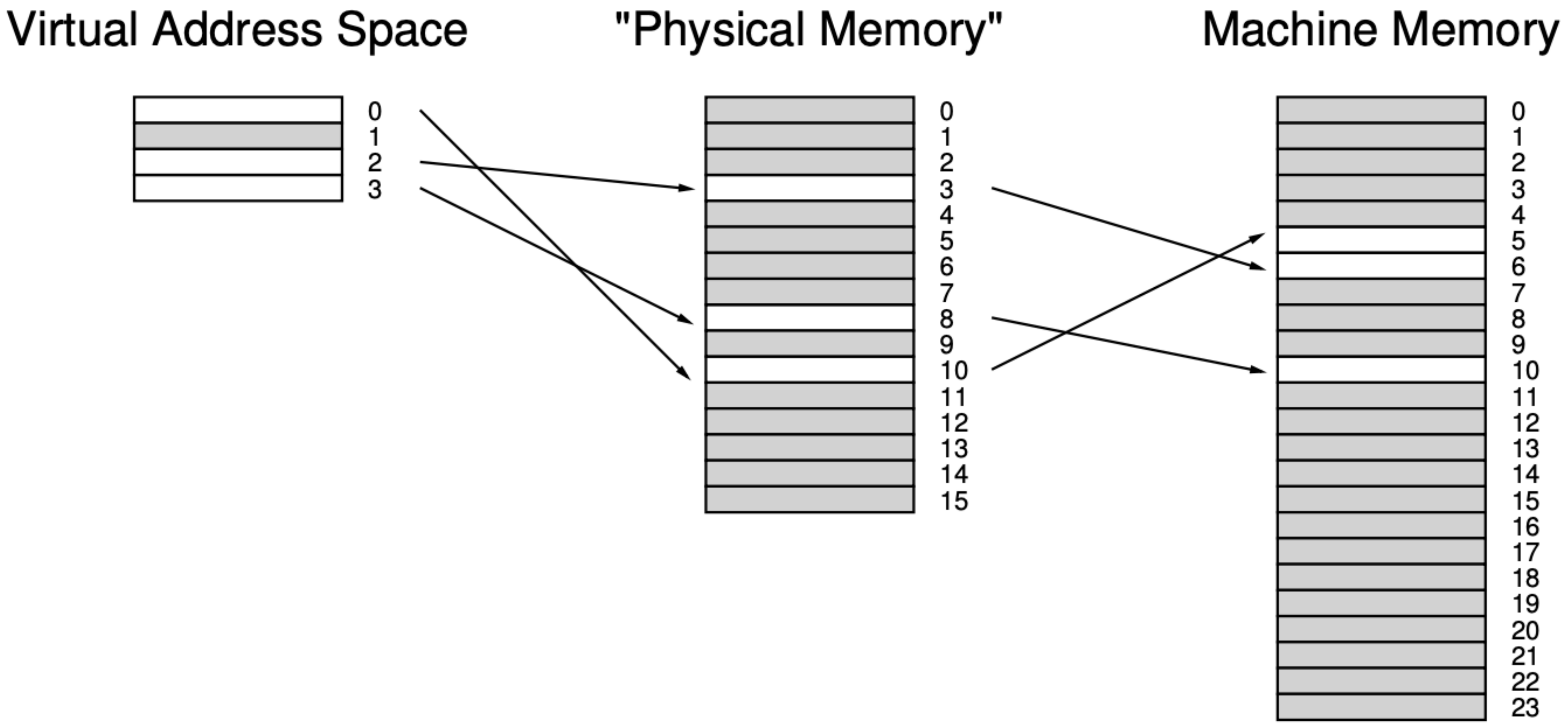
- **Normal Case**
  - Kernel mode
  - User mode
- **Virtualized Case**
  - User mode
  - Kernel mode
  - Hypervisor mode



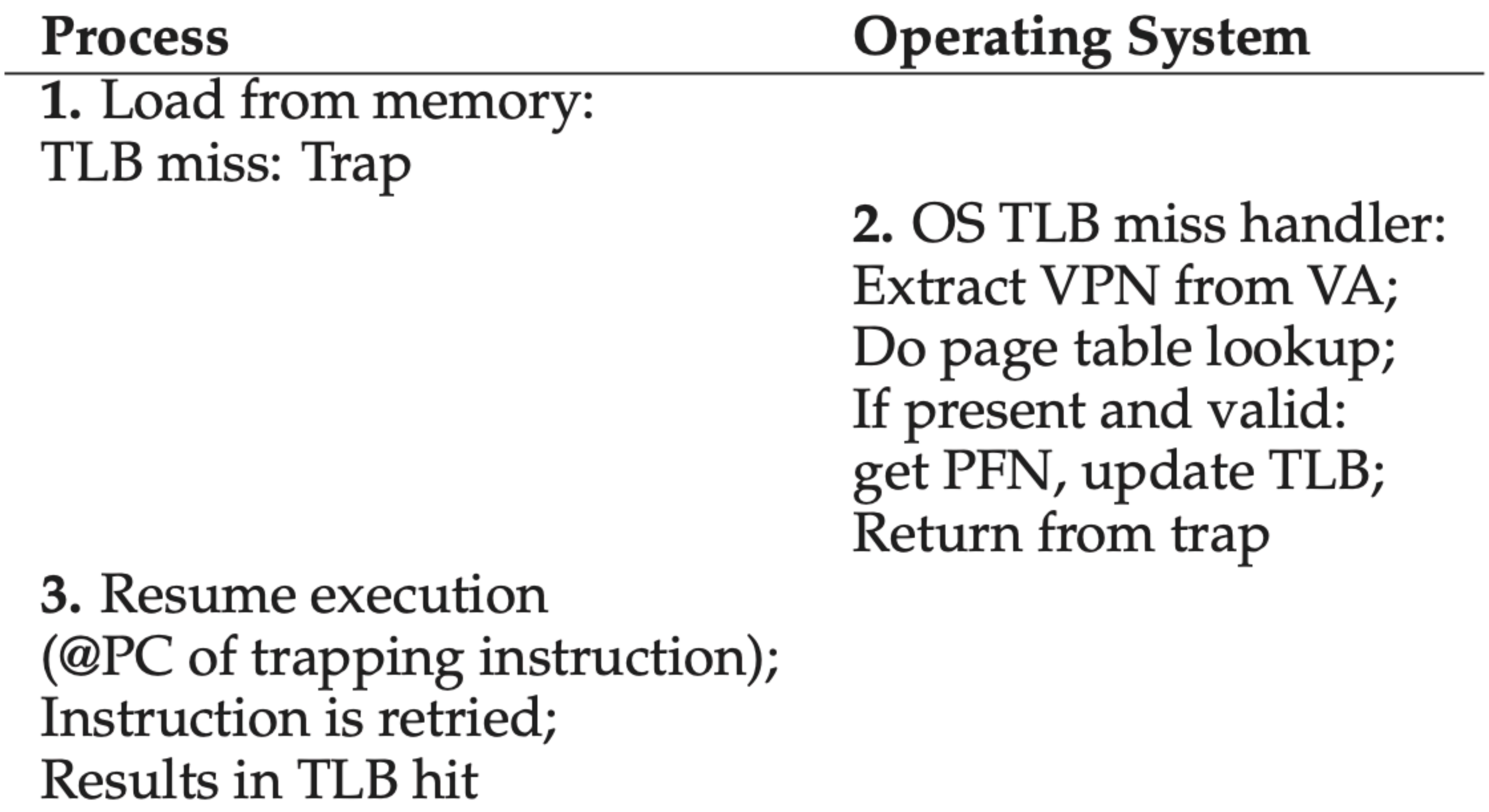
Virtualizing Memory

OS Page Table
VPN 0 to PFN 10
VPN 2 to PFN 03
VPN 3 to PFN 08

VMM Page Table
PFN 03 to MFN 06
PFN 08 to MFN 10
PFN 10 to MFN 05



A Recap of Virtual Memory



Process	Operating System
1. Load from memory: TLB miss: Trap	
	2. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid: get PFN, update TLB; Return from trap

3. Resume execution
(@PC of trapping instruction);
Instruction is retried;
Results in TLB hit	

Process	Operating System	Virtual Machine Monitor
1. Load from mem TLB miss: Trap		
		2. VMM TLB miss handler: Call into OS TLB handler (reducing privilege)
	3. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid, get PFN, update TLB	
		4. Trap handler: Unprivileged code trying to update the TLB; OS is trying to install VPN-to-PFN mapping; Update TLB instead with VPN-to-MFN (privileged); Jump back to OS (reducing privilege)
	5. Return from trap	
		6. Trap handler: Unprivileged code trying to return from a trap; Return from trap

7. Resume execution
(@PC of instruction);
Instruction is retried;
Results in TLB hit		

TLB miss handler?

- We have been assuming a software-managed TLB – so the OS is handling TLB misses
- **What about HW-managed TLBs (x86)?**
    - The hardware walks the page table on each TLB miss and updates the TLB as need be, and thus the VMM doesn’t have a chance to run on each TLB miss to sneak its translation into the system

Shadow Page Tables

- VMM must closely monitor changes the OS makes to each page table and keep a shadow page table that instead maps the virtual addresses of each process to the VMM’s desired machine pages.

Shadow Page Tables

- VMM maintains shadow page tables that map guest virtual pages (V) directly to host physical pages (GP).
- Guest modifications to V->GP tables synced to VMM V->HP shadow page tables.
  - Guest OS page tables marked as read-only.
  - Modifications of page tables by guest OS -> trapped to VMM.
  - Shadow page tables synced to the guest OS tables

Drawbacks: Shadow Page Tables

- Need to handle trap on all page table updates (and context switches)
  - Processor moves from vmx non-root (guest mode) to vmx root (host mode)
  - Similar to a CPU context switch, but actually more expensive
- Maintaining consistency between guest page tables and shadow page tables leads to frequent traps if guest has frequency switches or page table updates
- Loss of performance due to TLB flush on every “world-switch”
- Memory overhead due to shadow copying of guest page tables

Nested Page Tables

- Extended page-table mechanism (EPT) used to support the virtualization of physical memory.
- Guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory.
  - **The hardware gives us a 2nd set of page tables to do the translation without needing VMM intervention**
  - Of course, the VMM is still responsible for setting up the EPT, but this generally only needs to be done once at guest boot time

Address Translation

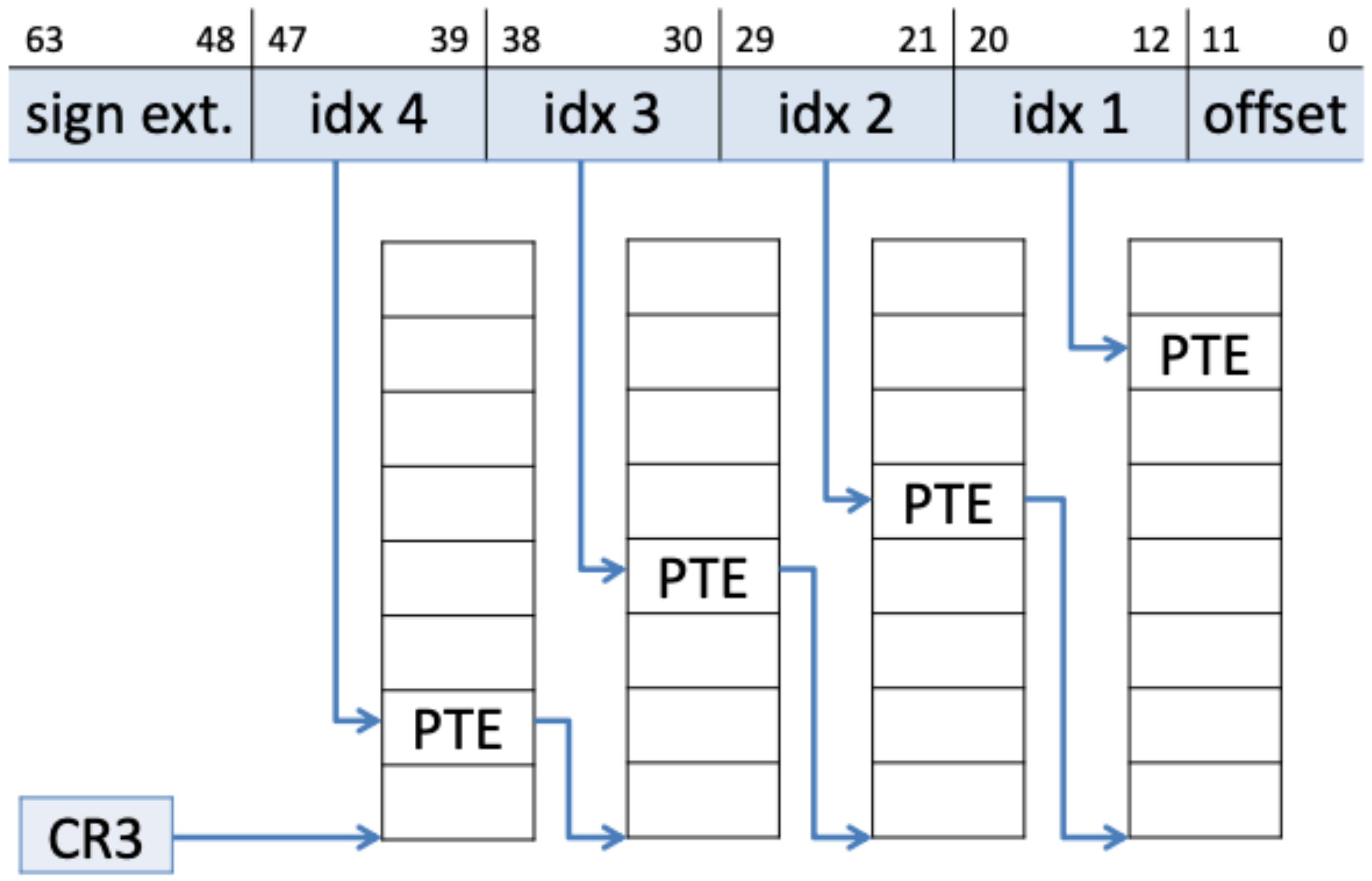
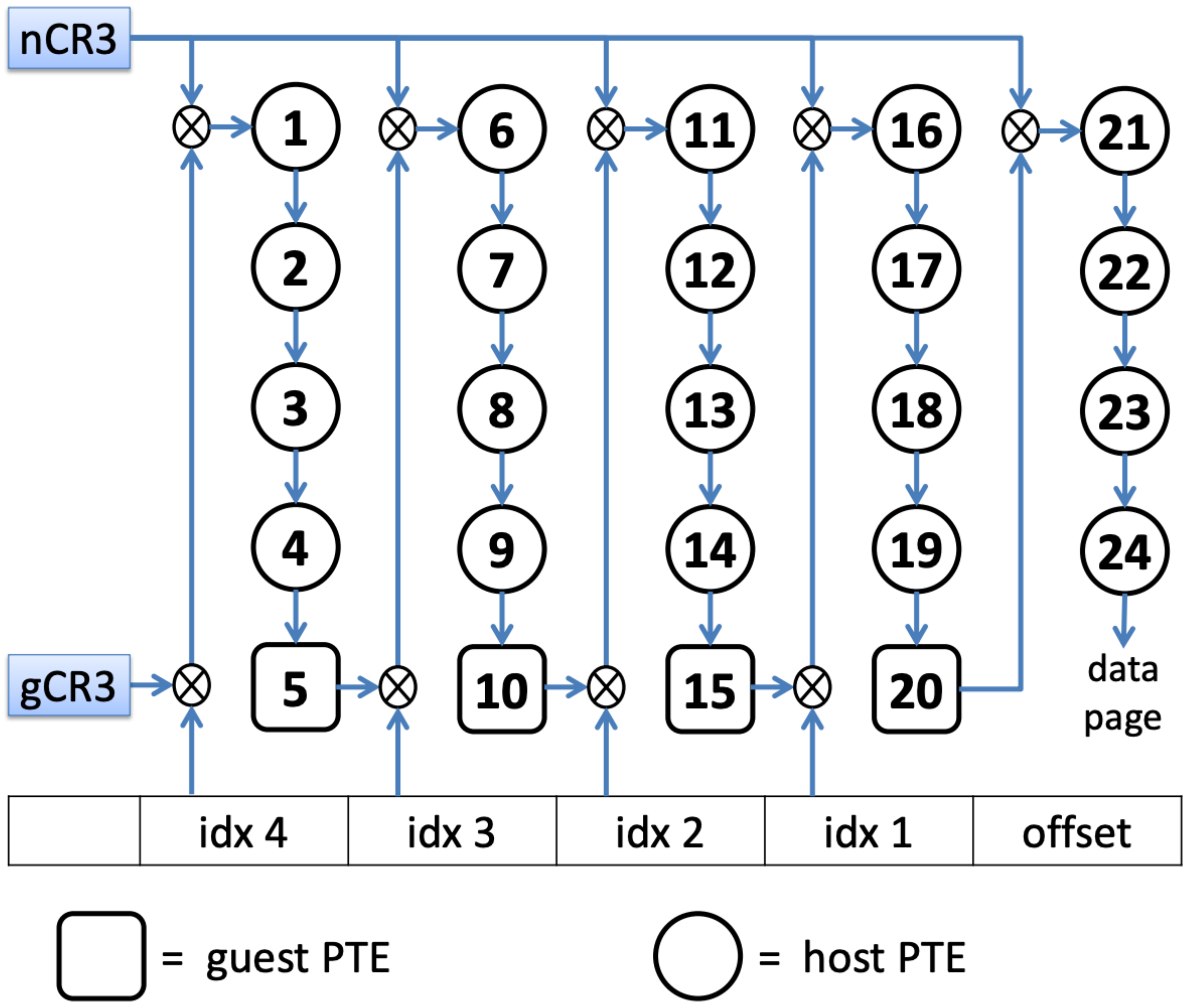


Figure 1: Bare-metal radix page table walk.

63 52	51 12	11 0
0's padding	PPN	attributes

Table 1: Radix PTE structure.

Virtualized Address Translation



Advantages: EPT

- Simplified VMM design (no need to maintain any "shadow" state or complex software MMU structures)
- Guest page table modifications need not be trapped, hence VM exits reduced.
- Reduced memory footprint compared to shadow page table algorithms.

Disadvantages: EPT

- TLB miss is very costly since guest-physical address to machine address needs an extra EPT walk for each stage of guest-virtual address translation.