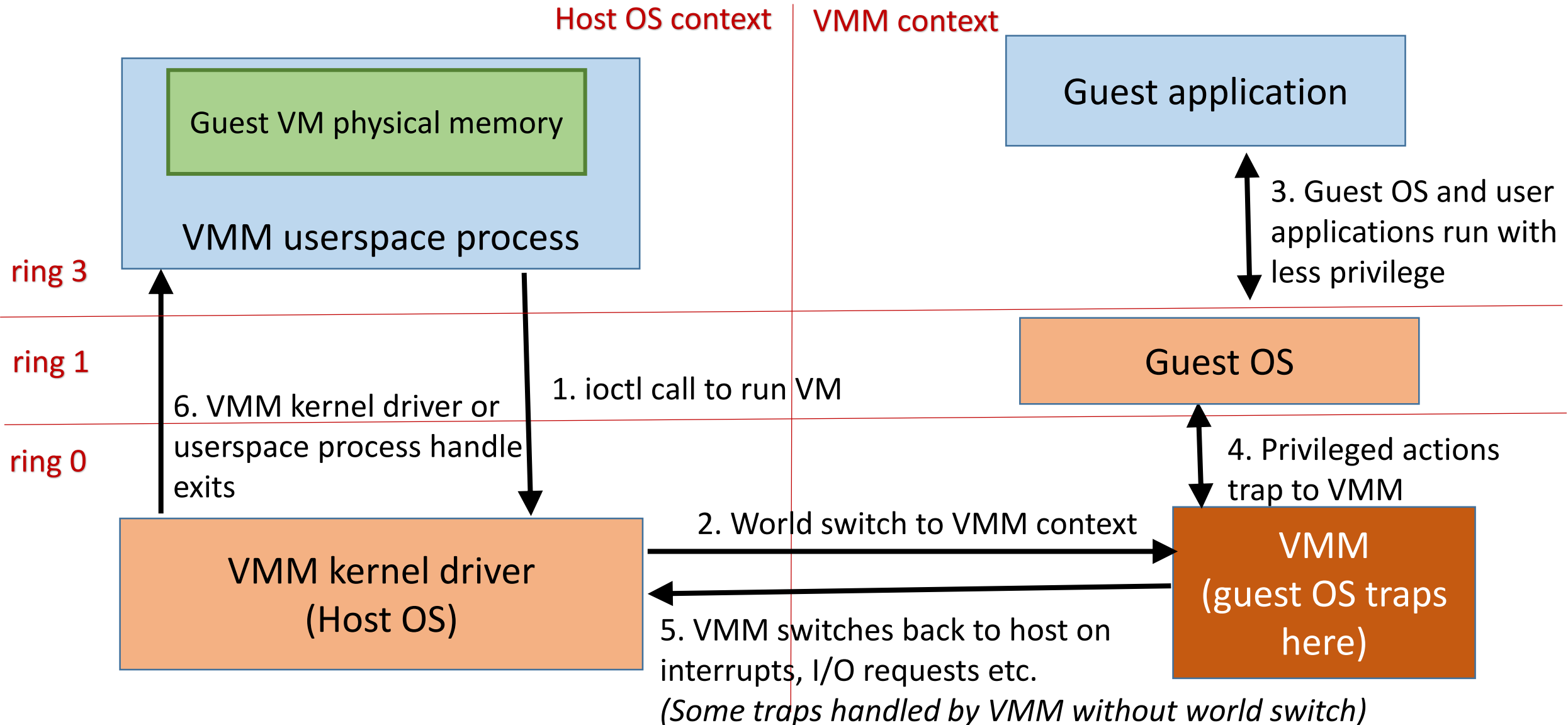


Full Virtualization

Full virtualization

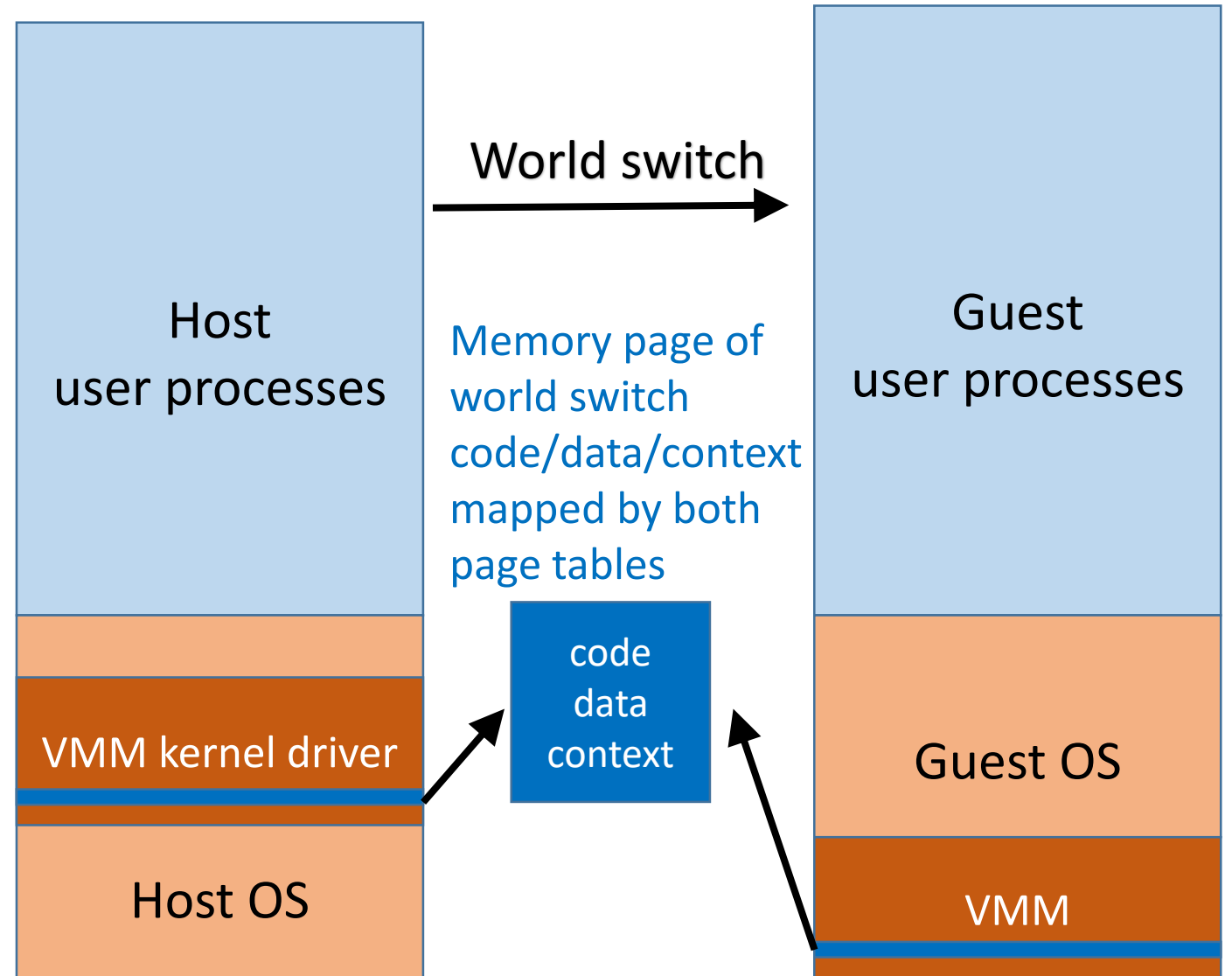
- x86 and other hardware lacked virtualization support
 - But cloud computing increased demand for virtualization
- VMWare workstation first to solve the problem of virtualization existing operating systems on x86 (basis for this lecture)
 - Type 2 hypervisor based on trap-and-emulate approach
- Key idea: **dynamic** (on a need basis) **binary** (not source) **translation** of OS instructions
 - Problematic OS instructions translated before execution
- Subsequently, hardware support for virtualization (previous lecture)
 - Binary translation is higher overhead than hardware-assisted virtualization
 - Used when hardware support not available

Full virtualization VMM architecture



Host and VMM contexts

- Each context has separate page tables, CPU registers, IDTs and so on
- VMM context: VMM occupies **top 4MB** of address space
- Memory page containing code/data of world switch mapped in both contexts
 - Host/VMM context saved/restored in this special “cross” page by VMM

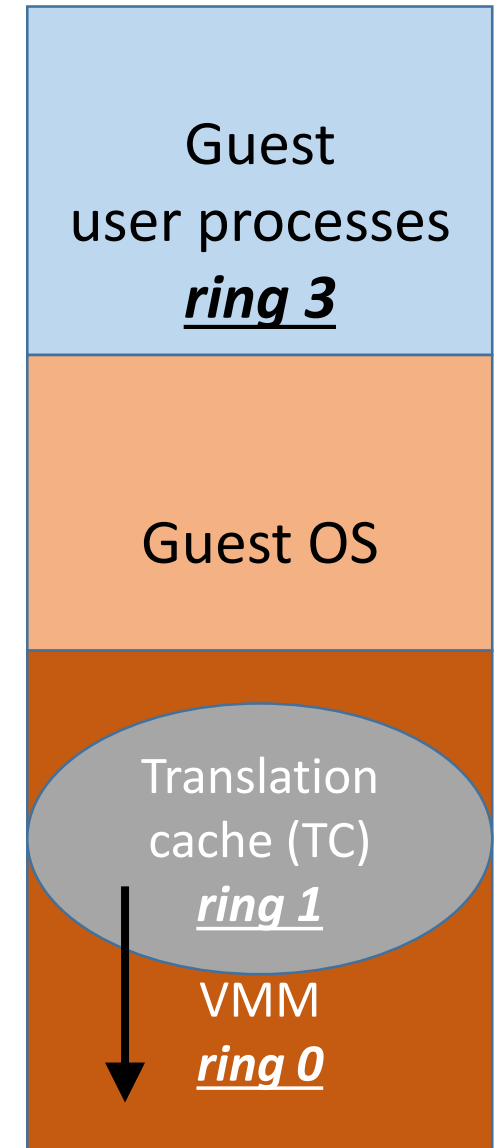


Understand difference with QEMU/KVM

- Where is context saved?
 - Common cross page mapped into both host and guest address spaces
 - KVM: Common memory (VMCS) accessible by CPU in both contexts via special instructions
- Privilege level of guest OS?
 - Guest OS runs in ring 1 (lower privilege). Instructions that do not run correctly at lower privilege level are suitably translated to trap to VMM
 - KVM: Guest OS runs in VMX ring 0. Some privileged instructions trigger exit to KVM
- How to trap to VMM?
 - VMM is located in top 4MB of guest address space , guest OS traps to VMM for privileged ops. World switch to host if VMM cannot handle trap in guest context
 - KVM: VMM is not in guest context, guest traps to VMM in host via VM exit

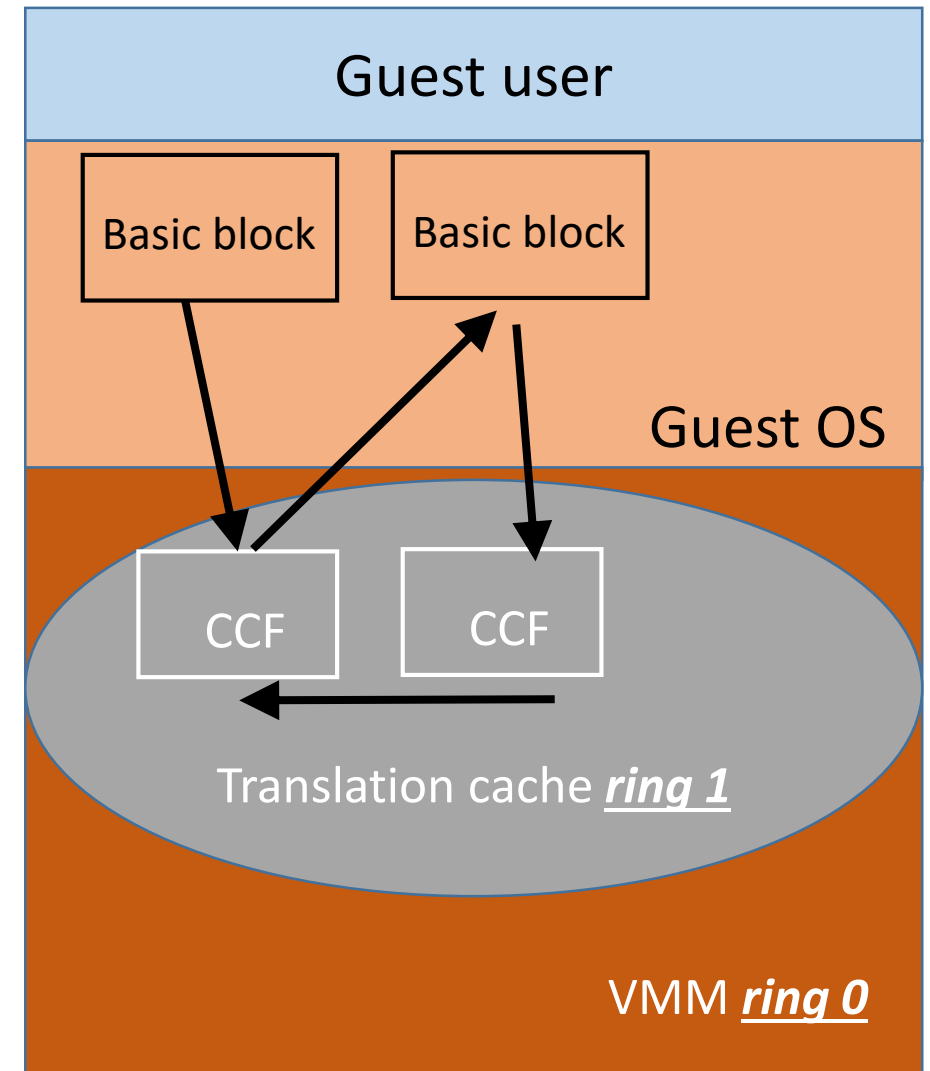
Binary translation

- Guest OS binary is translated instruction-by-instruction and stored in **translation cache (TC)**
 - Part of VMM memory
 - Most code stays same, unmodified
 - OS code modified to work correctly in ring 1
 - Sensitive but unprivileged instructions modified to trap
- Guest OS code executes from TC in ring 1
- Privileged OS code traps to VMM
 - E.g., I/O, set IDT, set CR3, other privileged ops
 - Emulated in VMM context or by switching to host
 - VMM sets sensitive data structures like IDT etc. (maintains **shadow** copies)



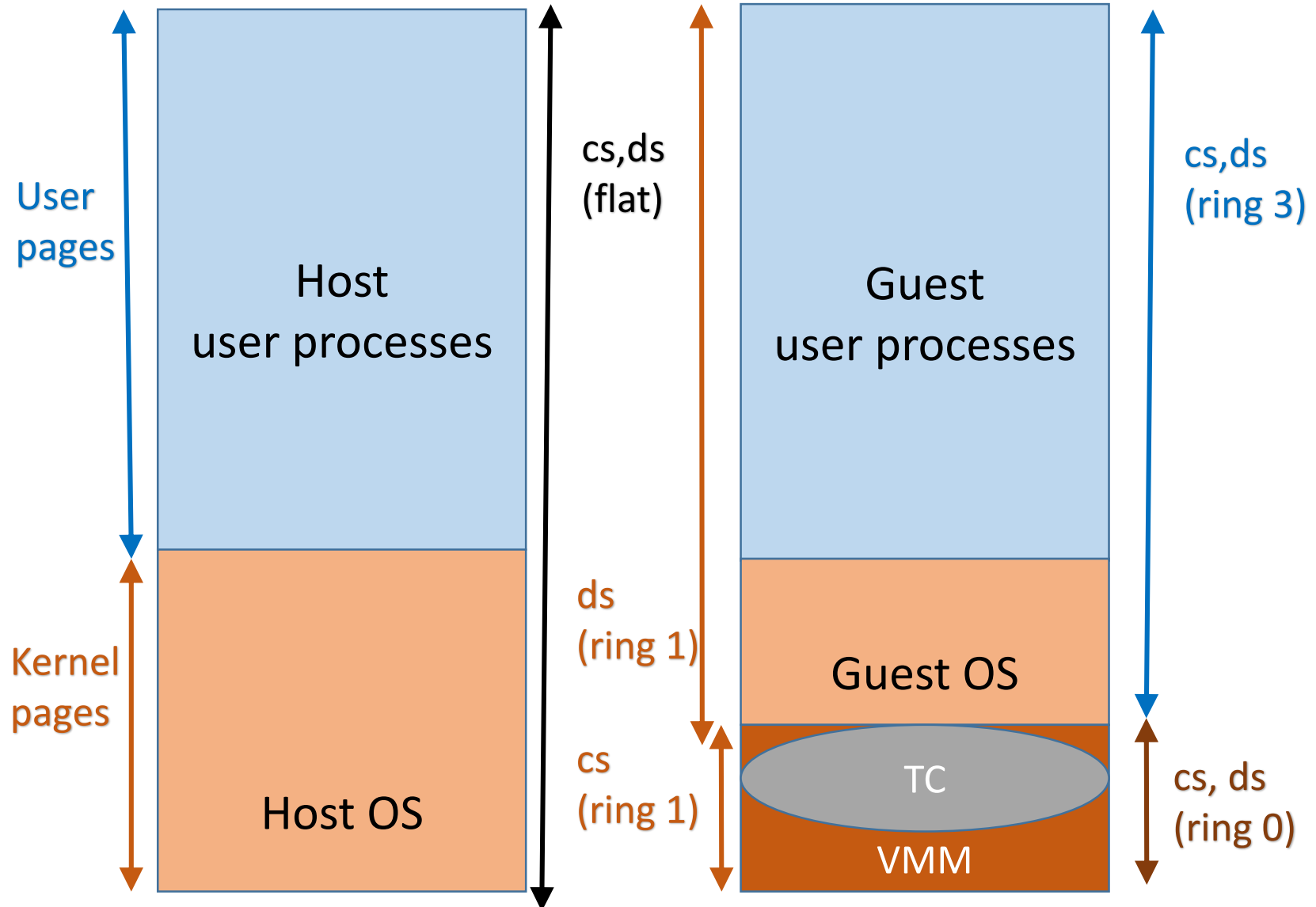
Dynamic binary translation

- VMM translator logic (ring 0) translates guest code one **basic block** at a time to produce a **compiled code fragment (CCF)**
 - Basic block = sequence of instructions until a jump/return
- Once CCF is created, move to ring 1 to run translated guest code
- Once CCF ends, “call out” to VMM logic, compute next instruction to jump to, translate, run CCF, and so on
- If next CCF present in TC already, then directly jump to it without invoking VMM translator logic
 - Optimization called **chaining**



Use of segmentation for protection

- Paging protects user code from kernel code via bit in page table entry
 - Segments are "flat"
 - Separate flat segments for user and kernel modes
- Segmentation is used to protect VMM from guest
 - Flat segments truncated to exclude VMM
 - CS of guest OS (ring 1) points to VMM
 - VMM (ring 0) segments point to top 4MB



Special case: GS segment (optional)

- Sometimes, translated guest code (ring 1) needs to access VMM data structures like saved register values, program counters and so on
- In such cases, memory accesses are rewritten to use the GS segment, e.g., virtual address “GS:someAddress”
- GS register points to the 4MB VMM area in ring 1
 - Ensures that the translated guest OS code can selectively access VMM data structures
- Original guest code that uses GS (which is rare) is rewritten to use another segment like %fs

Summary

- VMWare workstation is example of full virtualization, where unmodified OS is run on x86 hardware via dynamic binary translation
 - VMM user process and kernel driver on host trigger world switch from host OS context to VMM context
 - World switch code/data is part of both host and VMM contexts, special cross page accessible in both modes has saved contexts
 - VMM is in top 4MB of address space in VMM context
 - Translated guest code runs in ring 1, traps to VMM in ring 0 for privileged operations (trap-and-emulate)
 - Traps handled by VMM in ring 0, or VMM exits to host OS for emulation
 - Segmentation used to protect VMM from guest OS
- “Bringing Virtualization to the x86 Architecture with the Original VMware Workstation”, Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, Edward Y. Wang.
- “A Comparison of Software and Hardware Techniques for x86 Virtualization”, Keith Adams, Ole Agesen.