Ch 10: Advanced Memory Management

Bill Cheng

http://merlot.usc.edu/william/usc/

Main Points

- Applications of memory management
  - what can we do with the ability to trap on memory references to individual pages?
- File systems and persistent storage
  - goals
  - abstractions
  - interfaces

Address Translation Uses

- Address translation is a powerful tool
  - protection
  - fill-on-demand/zero-on-demand
  - copy-on-write
  - memory-mapped files
  - demand paged virtual memory

More Address Translation Uses

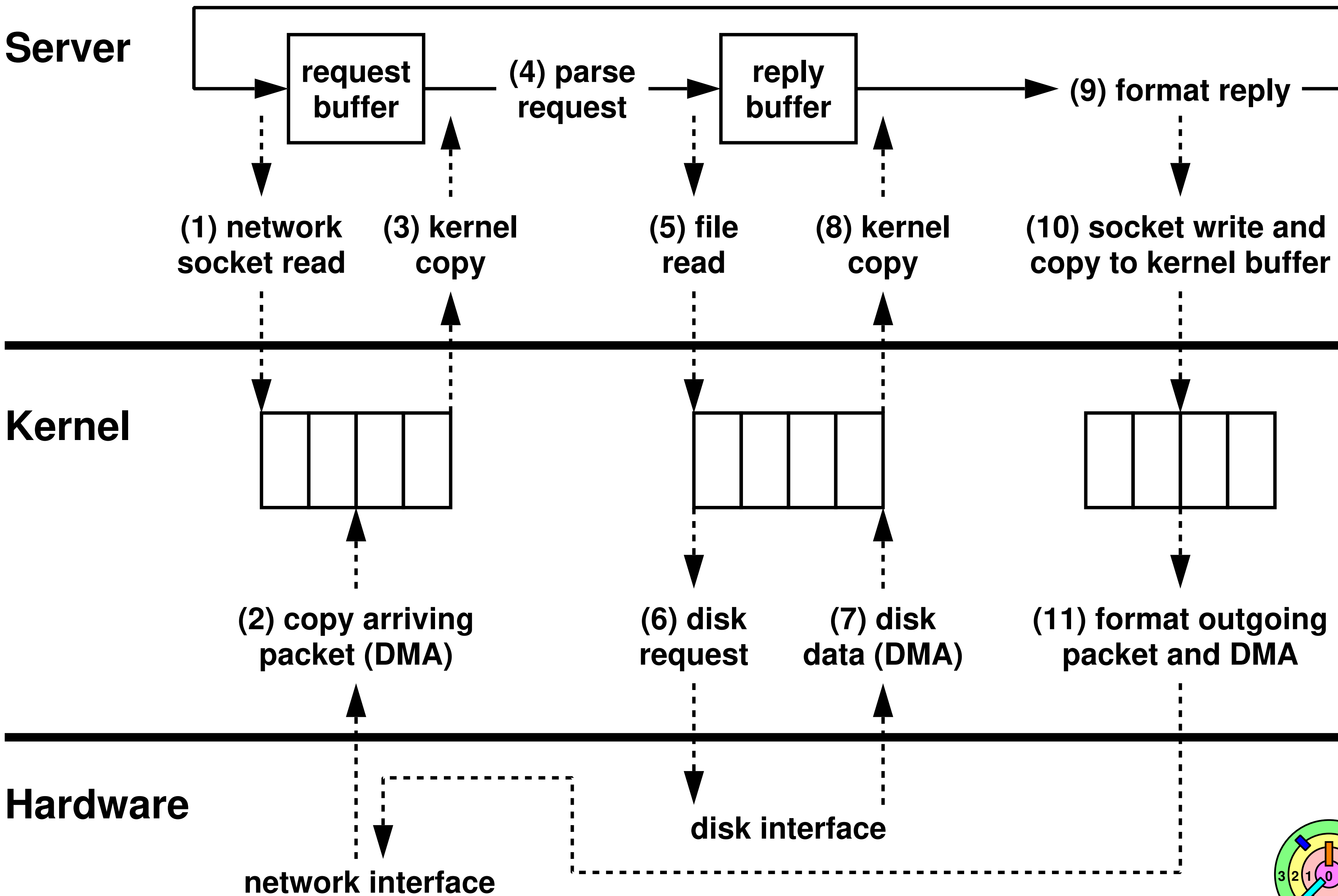
- *Zero-copy I/O:* directly from I/O device into/out of user memory
- *Checkpoint/restart:* transparently save a copy of a process, without stopping the program while the save happens
- *Process migration:* transparently move processes between machines
- *Recoverable virtual memory:* implement data structures that can survive system reboots
- *Program debugging:* data breakpoints when address is accessed
- *Cooperative Caching:* demand page to memory on a different machine
- *Distributed shared memory:* illusion of memory that is shared between machines

(10.1) Zero-Copy I/O

Data Streaming

- Many applications stream data between user-level programs and physical devices (such as disks and network hardware)
  - web server (disk $\rightarrow$ application buffer $\rightarrow$ network)
  - web client (upload)
  - online video/music service
  - network file system
  - file sharing service (such as BitTorrent)
  - etc.
- Problem: too much data copying from one buffer in kernel (or user space) to another buffer in user space (or kernel)

Web Server


Server
request buffer
(4) parse request
reply buffer
(9) format reply
(1) network socket read
(3) kernel copy
(5) file read
(8) kernel copy
(10) socket write and copy to kernel buffer
Kernel
(2) copy arriving packet (DMA)
(6) disk request
(7) disk data (DMA)
(11) format outgoing packet and DMA
Hardware
disk interface
network interface

Zero-Copy I/O

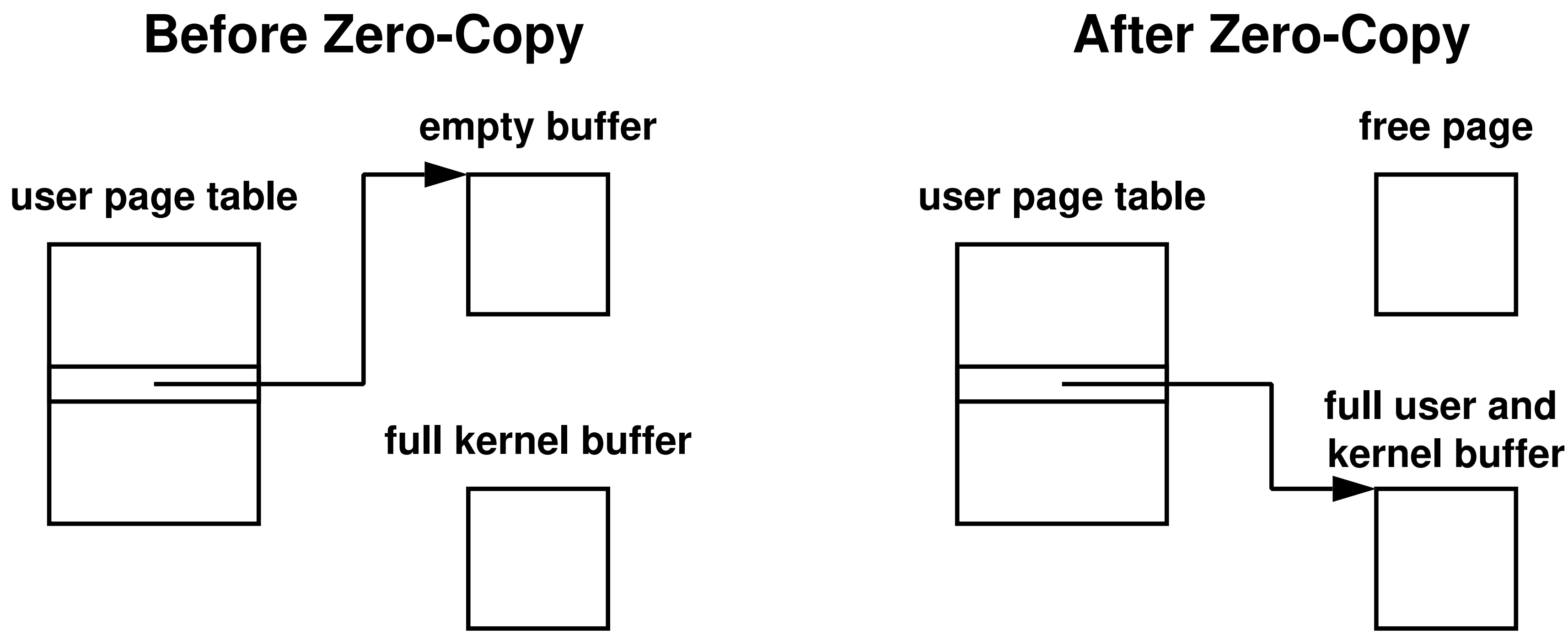
- When copying large amount of data across user-space / kernel boundary, don't copy data, copy pointers (in page table entry)
  - it's fine to copy actual data when copying small amount of data

- Requirements
  - application must first align its user-level buffer to page boundary

- When copying from user-space to kernel, the kernel must make the page R/O to prevent the page being modified
  - the kernel must also *pin* the page to prevent it from being evicted by the virtual memory manager (since this page is considered a kernel buffer now)
  - if the user-space writes to the page, it will trap into the kernel and the kernel can make a copy of the page (and unpin the original page and make the original page R/W)

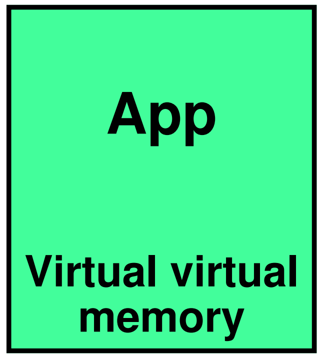
Zero-Copy I/O

- When copying from kernel to user-space, the kernel can just simply change the page table entry (and reclaim the physical memory behind the empty buffer)



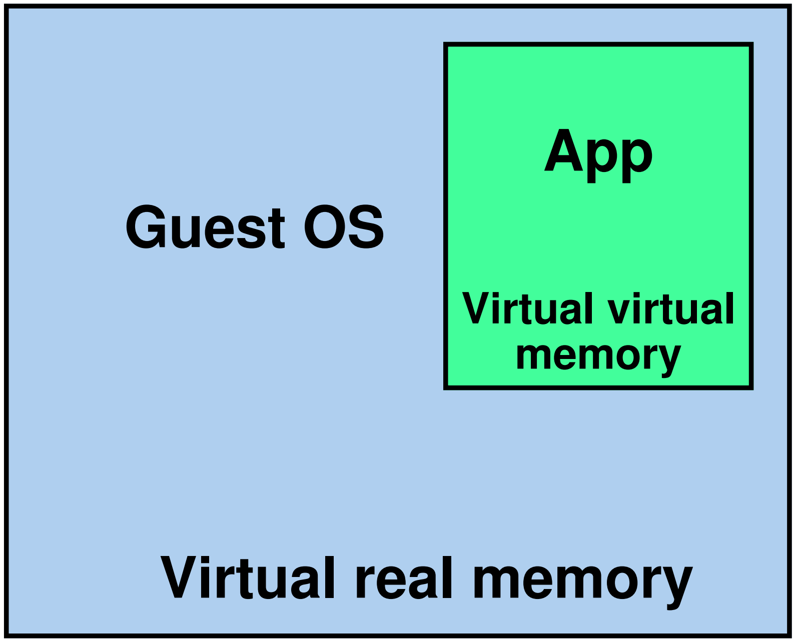
(10.2) Virtual Machines

How Do You Virtualize Virtual Memory?



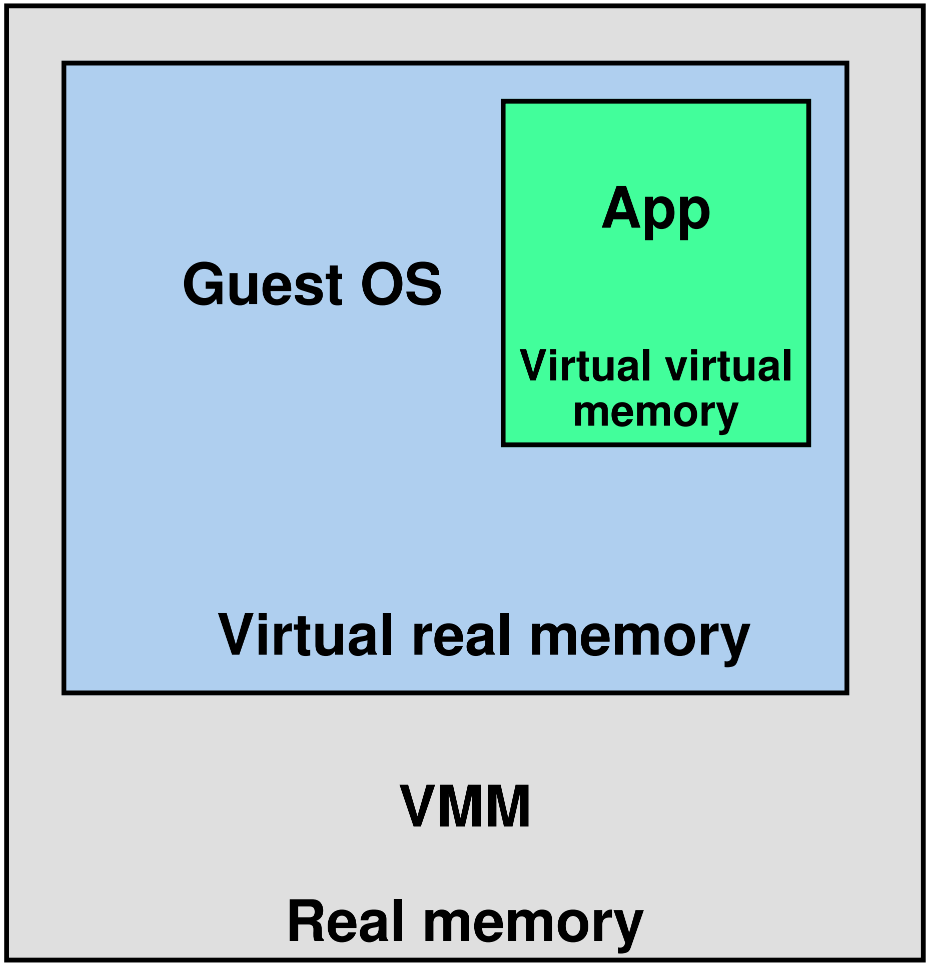
- A user process running inside a virtual machine thinks it's accessing virtual memory
  - but it's really dealing with ***virtual virtual memory*** **(i.e., guest virtual memory)**

How Do You Virtualize Virtual Memory?



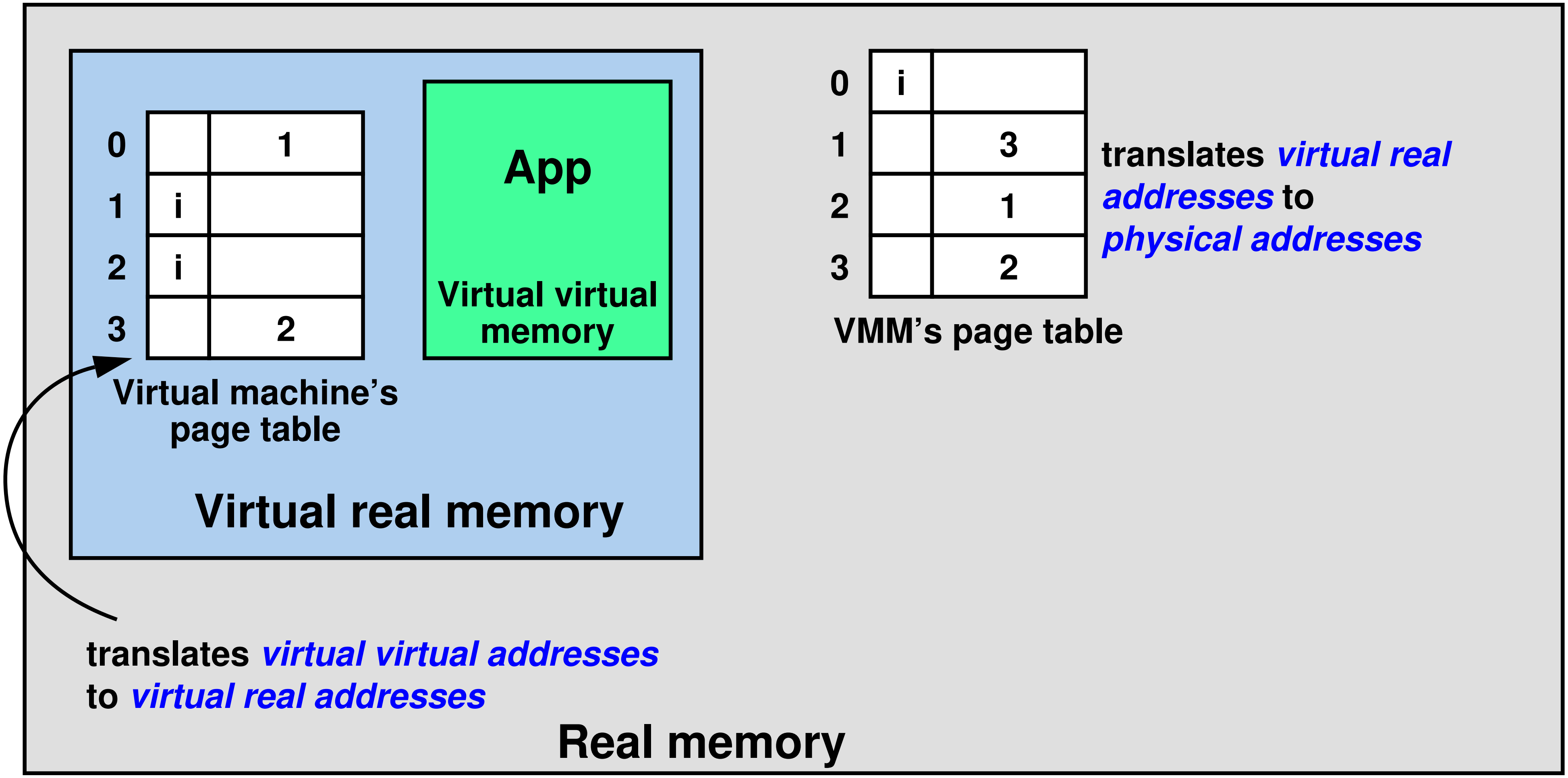
- A user process running inside a virtual machine thinks it's accessing virtual memory
  - but it's really dealing with *virtual virtual memory* (i.e., guest virtual memory)
- The guest OS thinks it's managing real memory
  - but it's really dealing with *virtual real memory* (i.e., guest physical memory)

How Do You Virtualize Virtual Memory?



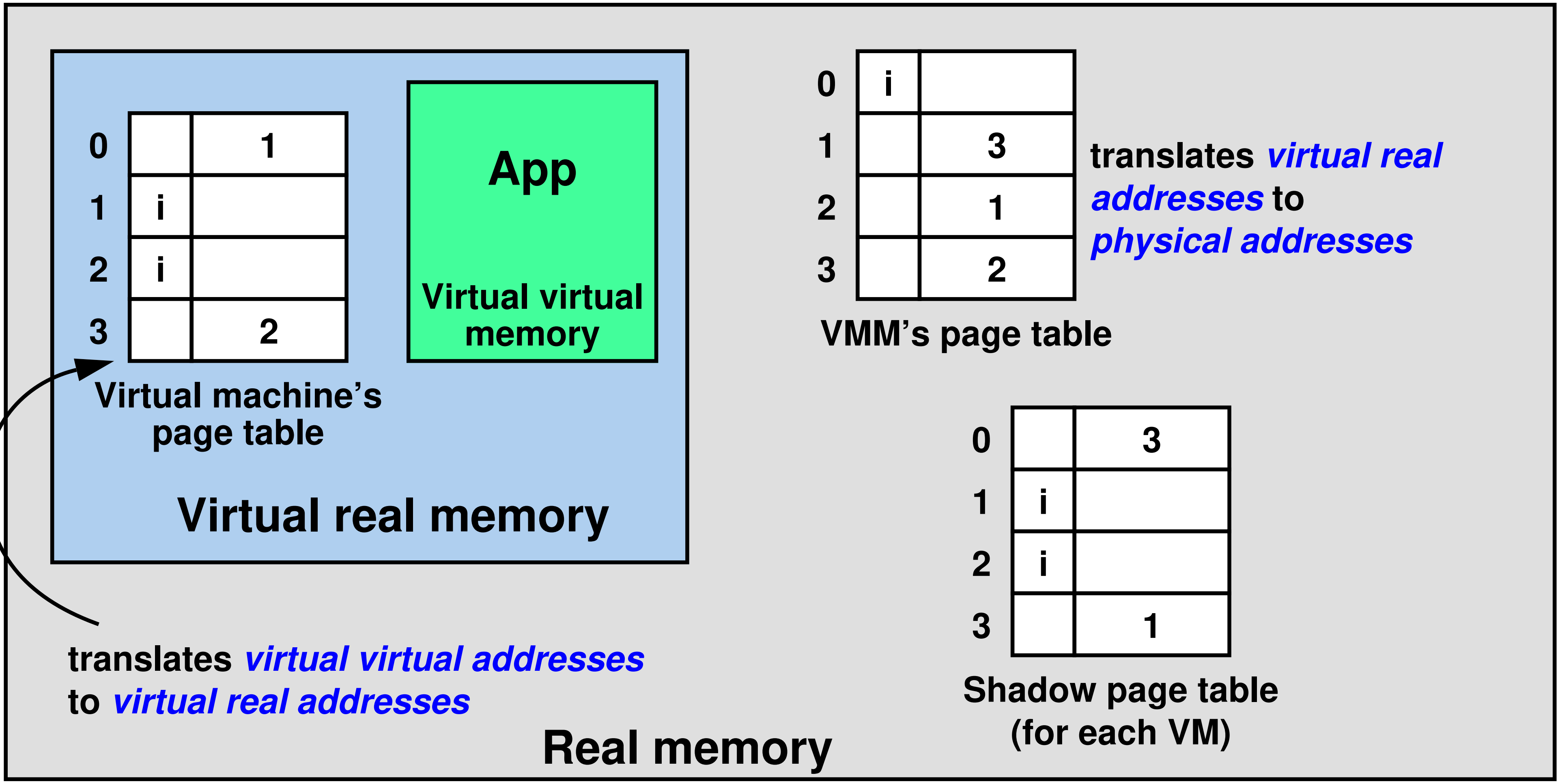
- A user process running inside a virtual machine thinks it's accessing virtual memory
  - but it's really dealing with *virtual virtual memory* (i.e., guest virtual memory)
- The guest OS thinks it's managing real memory
  - but it's really dealing with *virtual real memory* (i.e., guest physical memory)
- VMM needs to manage real memory (host physical memory)
  - how can we *virtualize virtual memory*?

How Do You Virtualize Virtual Memory?



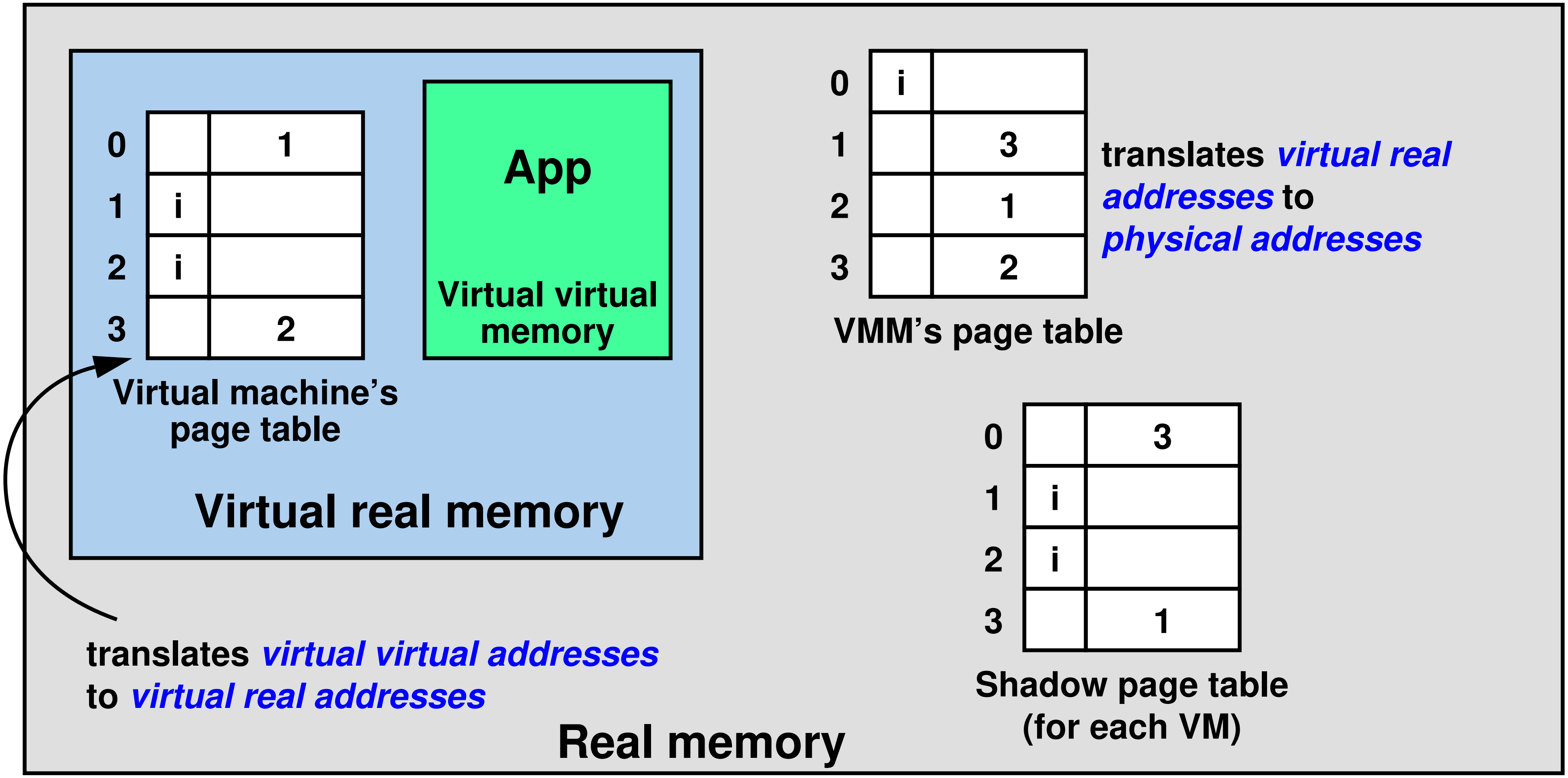
- VMM cannot use either page tables directly
  - must combine these two page tables into one ***shadow page table*** and use that in VMM to perform address translation

Shadow Page Table



- VMM cannot use either page tables directly
  - must combine these two page tables into one *shadow page table* and use that in VMM to perform address translation

Shadow Page Table

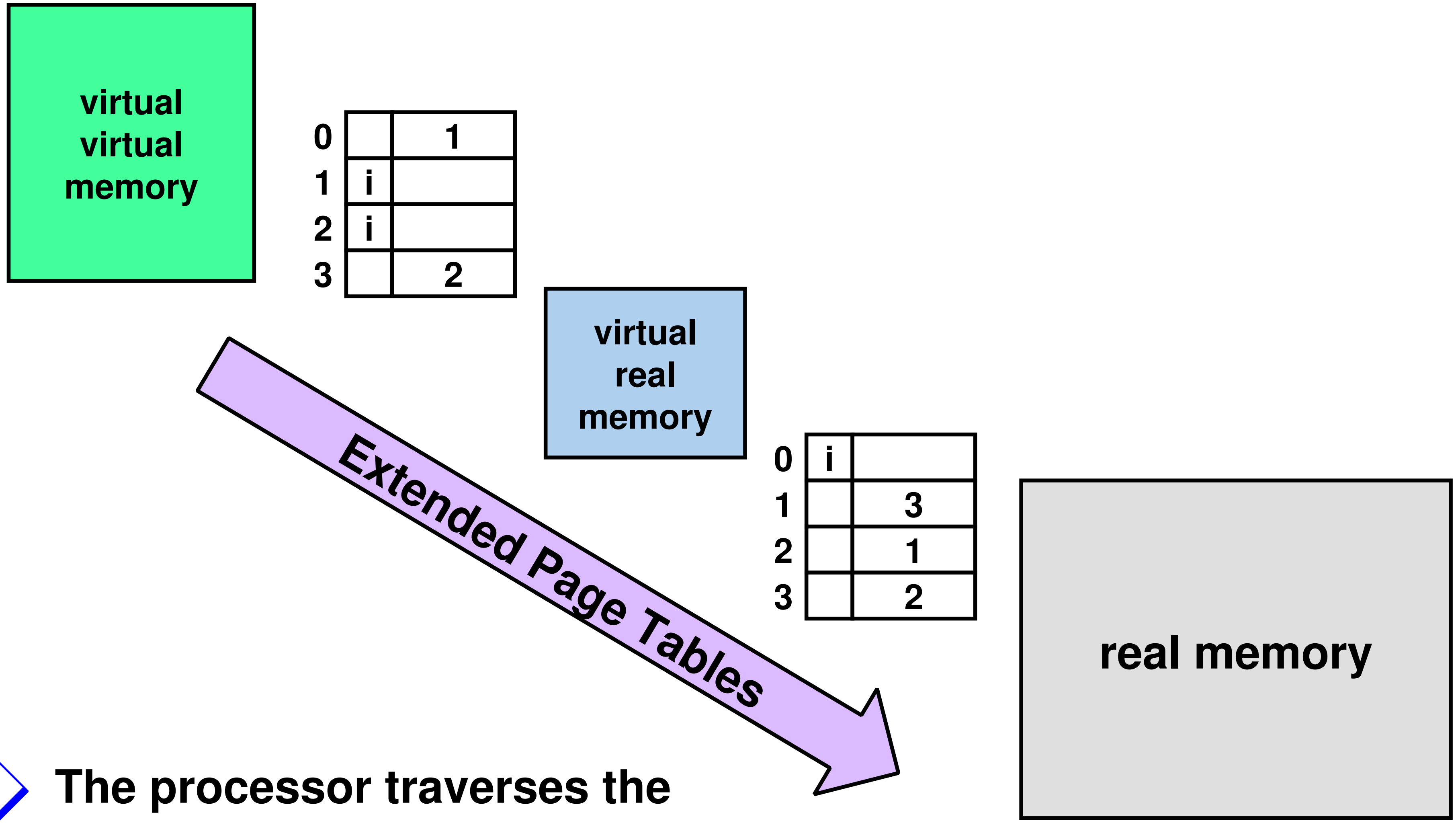


- When a VM changes its page table, VMM must update the corresponding *shadow page table*
  - main problem: poor performance

Hardware Support for Virtual Machines

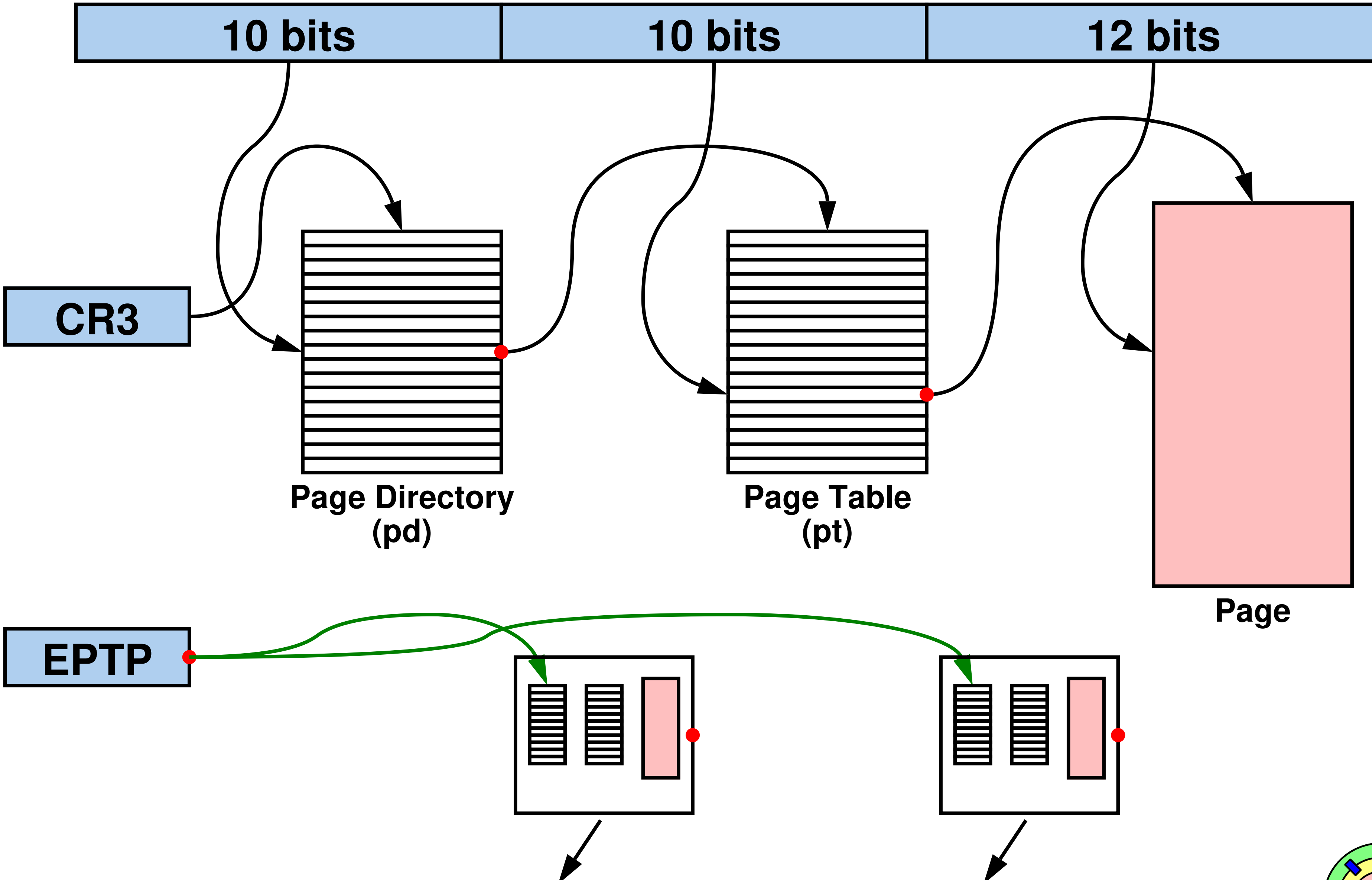
- x86 recently added hardware support for running virtual machines at user level called *Extended Page Table*
- Operating system kernel initializes two sets of translation tables
  - one for the guest OS
  - one for the host OS
- Hardware translates address in two steps
  - first using guest OS tables, then host OS tables
  - TLB holds composition

Extended Page Table



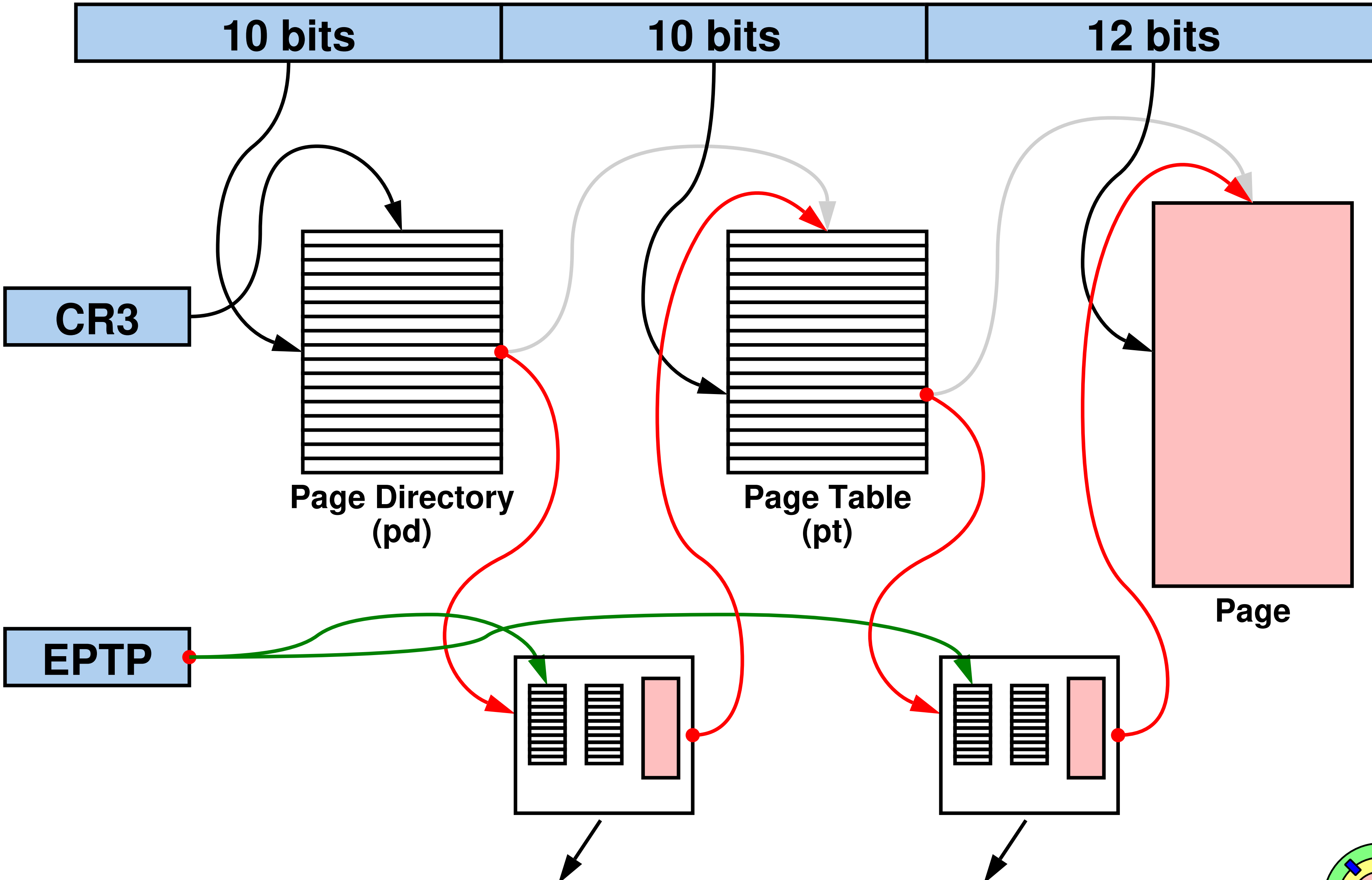
- The processor traverses the two tables in sequence and does the page table combining all by itself

x86 Paging with EPT



these translates *virtual real addresses*
to *physical addresses*

x86 Paging with EPT

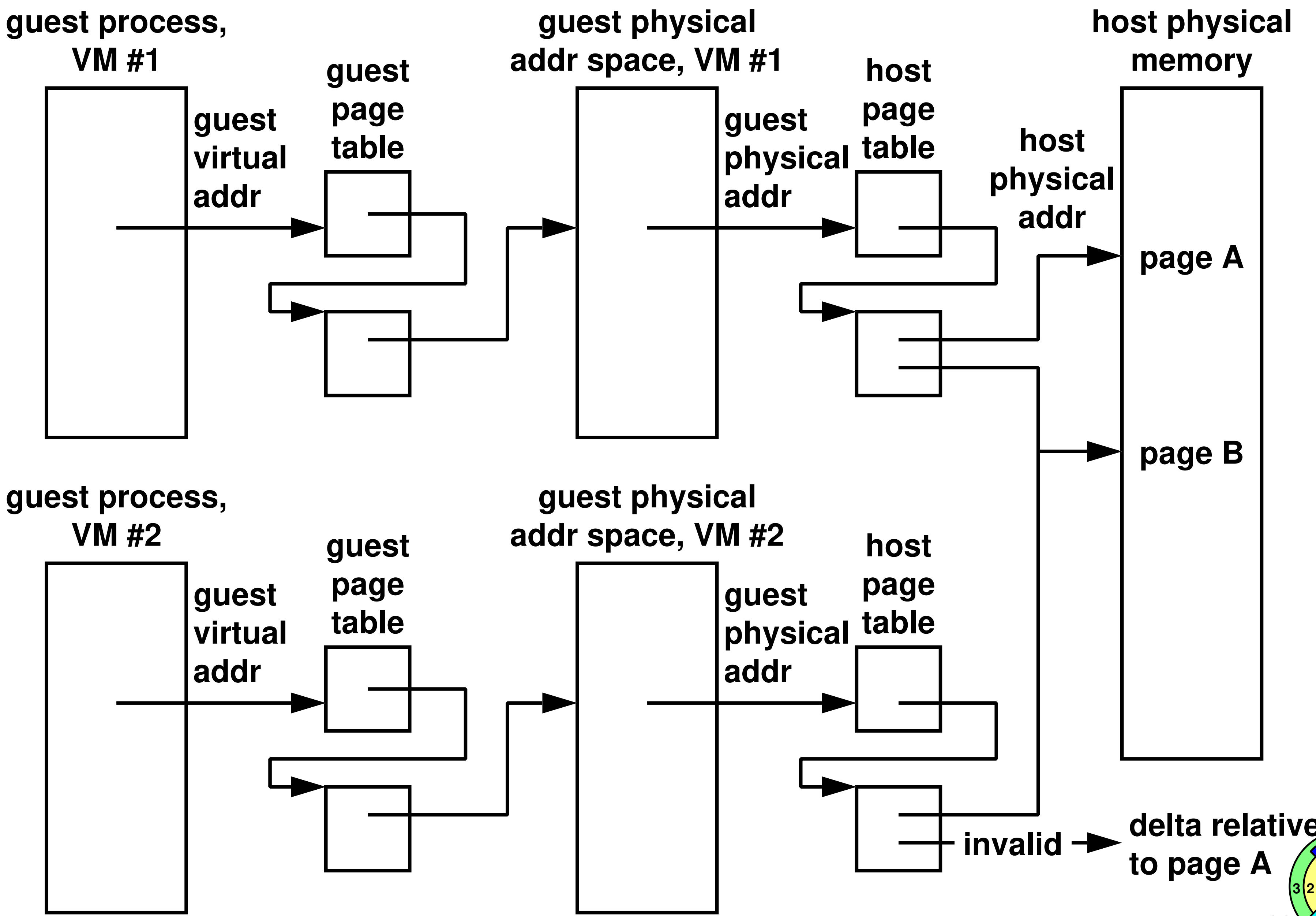


these translates *virtual real addresses*
to *physical addresses*

Transparent Memory Compression

- For VMs in a data center, pages from different VMs can be shared
  - e.g., zero pages
    - make page table entries invalid, on page fault, create a new page of zeroes
  - it's also possible to shared pages that are almost the same
    - can compute the difference (delta) between two pages then compress and store the difference on disk and set the corresponding page table entry to be invalid
      - on a page fault, bring the compressed delta from disk, uncompress it and apply the difference
  - if page size is big (such as 2MB), this trick can save a lot of physical memory
- Can run a scavenger task in the VMM to look for identical or similar pages across different VMs to perform the *compression*

Transparent Memory Compression

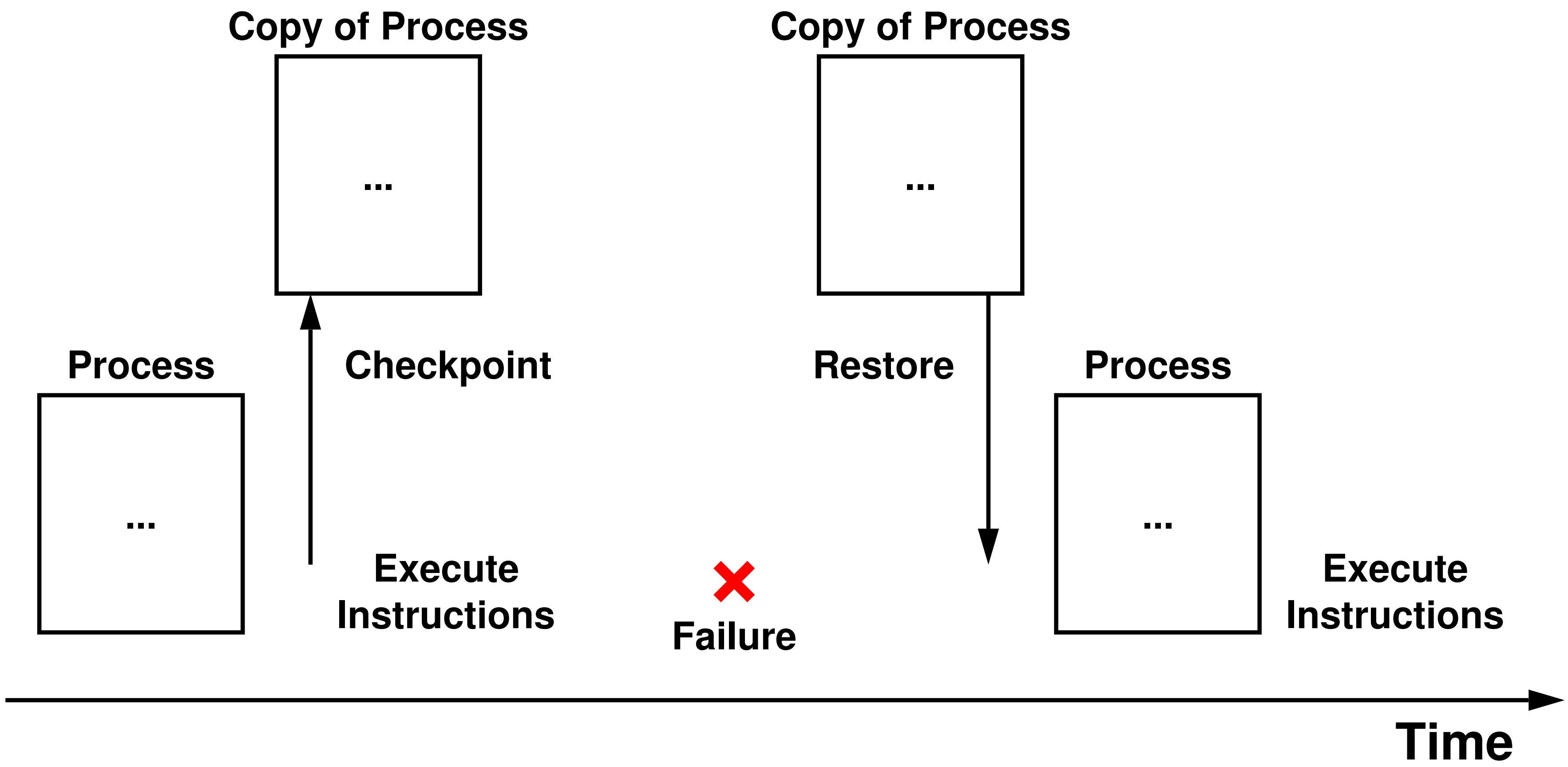


(10.3) Fault Tolerance

Fault Tolerance

- For a long running program (that runs for days, weeks, months), it would be nice to be able to recover from power glitches and temporary hardware errors
  - the likelihood of this happening in a data center is not negligible
- Application should be doing checkpointing itself and write results to disk periodically
  - can the OS help?
  - we want to be able to restart a process whenever the power fails, exactly where it left off, without the user's knowledge

Transparent Checkpoint



- First we need to suspend all threads executing in the process and save its state (i.e., register values)
  - then save memory
  - this is called a checkpoint or a snapshot

Transparent Checkpoint

- Can we perform checkpointing while the threads are running?
  - after the registers are saved, we can do copy-on-write by setting all valid page table entries to R/O
  - when a page is saved on disk, change the page to R/W
  - when copy-on-write kicks in, make a copy of the page
    - original page goes to disk
    - let the running program use the copy of the page
- We can check point the OS if the OS is running inside a virtual machine using the same approach

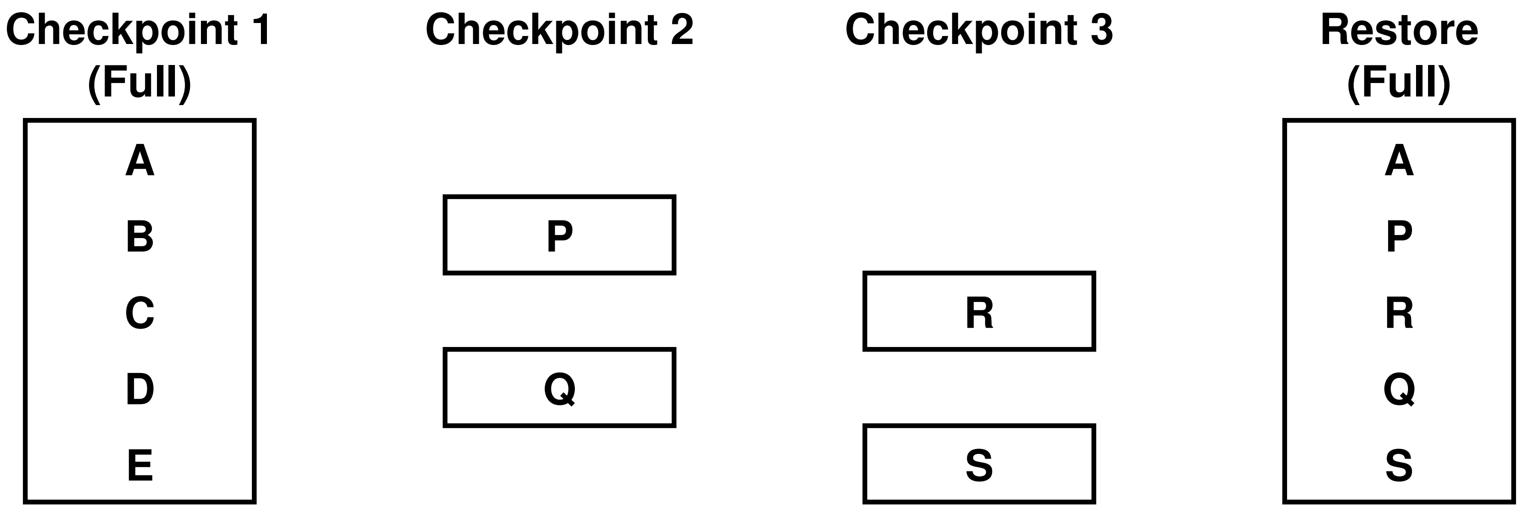
Process Migration

- What if we checkpoint a process and then restart it on a different machine?
  - *process migration:* move a process from one machine to another
  - this is often done inside a data center to migrate an entire virtual machine for load balancing

Recoverable Virtual Memory

- *Recoverable Virtual Memory:* if there is a failure (e.g., power lost or system crash), restore the contents of memory to a pointer not long before the failure
  - using the same copy-on-write trick as before, this can be done, but can be quite slow to checkpoint the entire process every time you write into memory
- Solution: don't checkpoint the entire process, instead, do it *incrementally*
  - only save copy of any pages that have been modified since the last incremental checkpoint
  - if there is a crash, we can recover the most recently memory by reading in the first checkpoint, then applying each of the incremental checkpoint

Incremental Checkpoint



- The idea here is very similiar to the idea of file system backup where you can recover a lost file
  - on Linux, level 0 backup is a full backup
  - other levels are incremental backups
  - if you set up a one-week daily backup schedule, you will start with a level 0 full backup
    - next day is level 6, then level 5, etc.
    - if you lose a file, you can only go back to any version up to one week ago

Deterministic Debugging

- Can we precisely replay the execution of a multi-threaded process (or an OS kernel)?
  - e.g., debug whether a process has a memory race or not
- On a uniprocessor, the execution of an OS running in a virtual machine can only be affected by three factors:
  - its initial state
  - input data provided by its I/O devices
  - the precise timing of interrupts
- Since host OS mediate each of the above factors for the virtual machine, it can record them and play them back during debugging
  - from a checkpoint, record:
    - all inputs and return values from system calls
    - all scheduling decisions
    - all synchronization operations
      - e.g., which thread acquired lock in which order

Cooperative Caching

- Can we demand page to memory on a different machine?
  - accessing remote memory over LAN is usually much faster than accessing a local hard drive
  - on page fault, look in remote memory first before fetching from disk

Distributed Virtual Memory

- Can we make a network of computers appear to be a shared-memory multiprocessor?
  - *read-write:* if page is cached only on one machine
  - *read-only:* if page is cached on several machines
  - *invalid:* if page is cached read-write on a different machine
- On read page fault:
  - change remote copy to read-only
  - copy remote version to local machine
- On write page fault (if cached):
  - change remote copy to invalid
  - change local copy to read-write

(10.4) Security

Security Through Virtual Machines

- How can virtual machines be used to limit the scope of malicious applications?
  - *Virtual Machine Honeypot:* a virtual machine constructed for the purpose of executing suspect code
    - used in commercial anti-virus software
- Creating a virtual machine honeypot may seem extravagant
  - reward is high if you can catch malicious applications before it can do any harm to your real system
- Virtual machine honeypot does not have to run as fast as the real system
  - can use these tricks to create a virtual machine honeypot:
    - shadow page tables
    - memory compression
    - efficient checkpoint and restart
    - copy-on-write

Security Through Virtual Machines

- How do you know the virtual machine honeypot has been corrupted?
  - typically, a virus would immediately install logging software or scan the disk for sensitive information
    - a smart virus may stay dormant for a while and do bad things slowly to avoid detection

- Some virus is designed to infect both the guest OS and the host kernel implementing the virtual machine
  - as a user, it's super important to keep system software up to date
    - the system is vulnerable only if the virus is able to exploit unknown weakness in the guest OS *and* a separate unknown weakness in the host kernel
    - *Defense In Depth:* improving security through multiple layers of protection

(10.5) User-Level Memory Management

User-Level Memory Management

- **How to let applications manage their own memory?**
  - **the kernel is in still charge of allocating resource between processes and in preventing access to privileged memory**
    - **once a page frame has been assigned to a process, the kernel can leave it up to the process to determine what to do with the page**

- **OS can provide applications the flexibility to decide:**
  - ***where to get missing pages:*** **e.g., from remote memory inside a data center, remote disk, local disk, local non-volatile memory, etc.**
  - ***which page can be accessed:*** **e.g., browsers and databases need to set up their own application-level sandboxes to execute untrusted code**
  - ***which page should be evicted:*** **application often knows better which page will be less likely to be referenced in the near future**

User-Level Memory Management

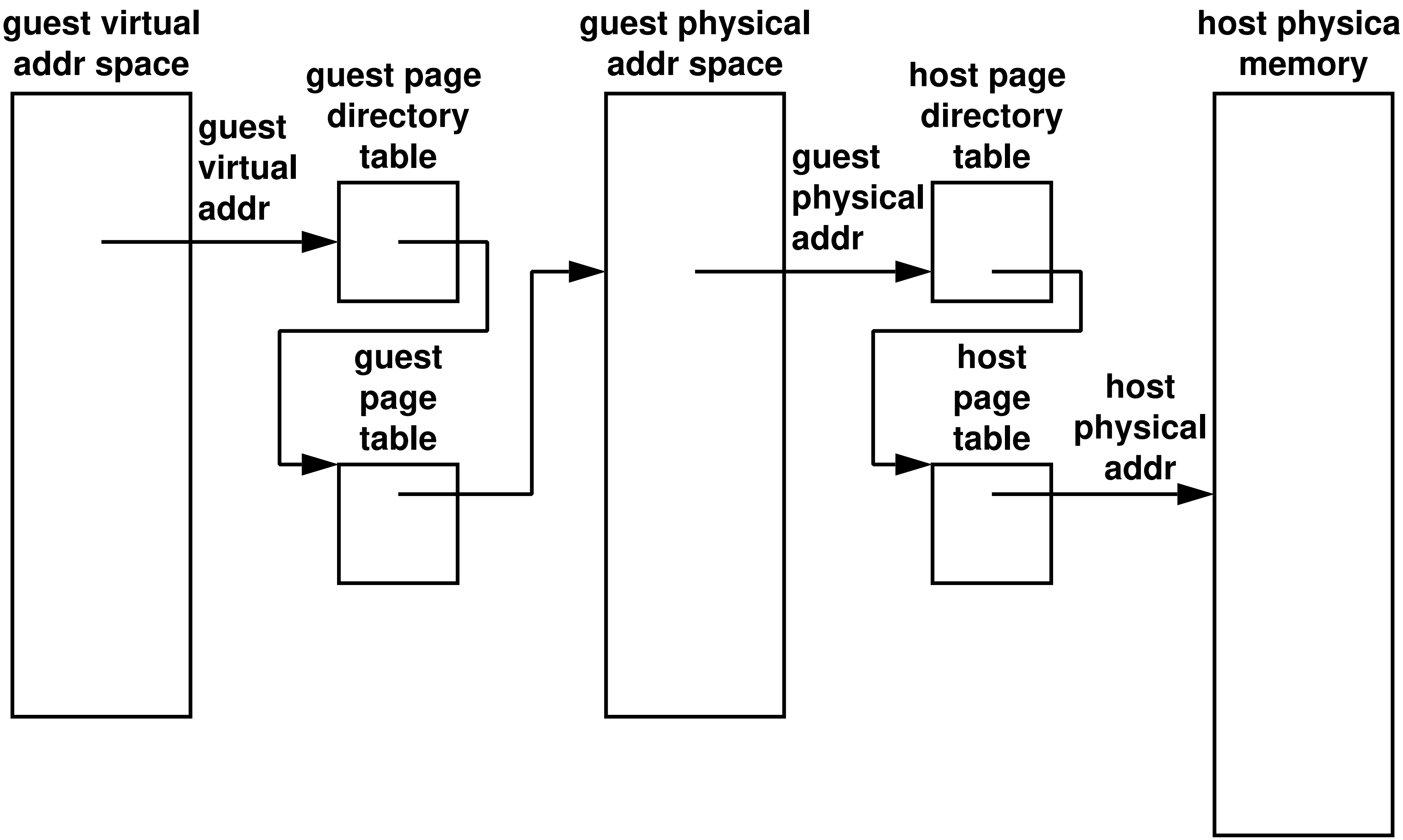
- Many applications can adapt the size of their working set to the resources provided by the kernel: better match $\rightarrow$ better performance
  - *garbage collected programs:* some programs want to do its own garbage collection
  - *databases and virtual machines:* these applications work best if they know how much physical memory is available to them

User-Level Memory Management

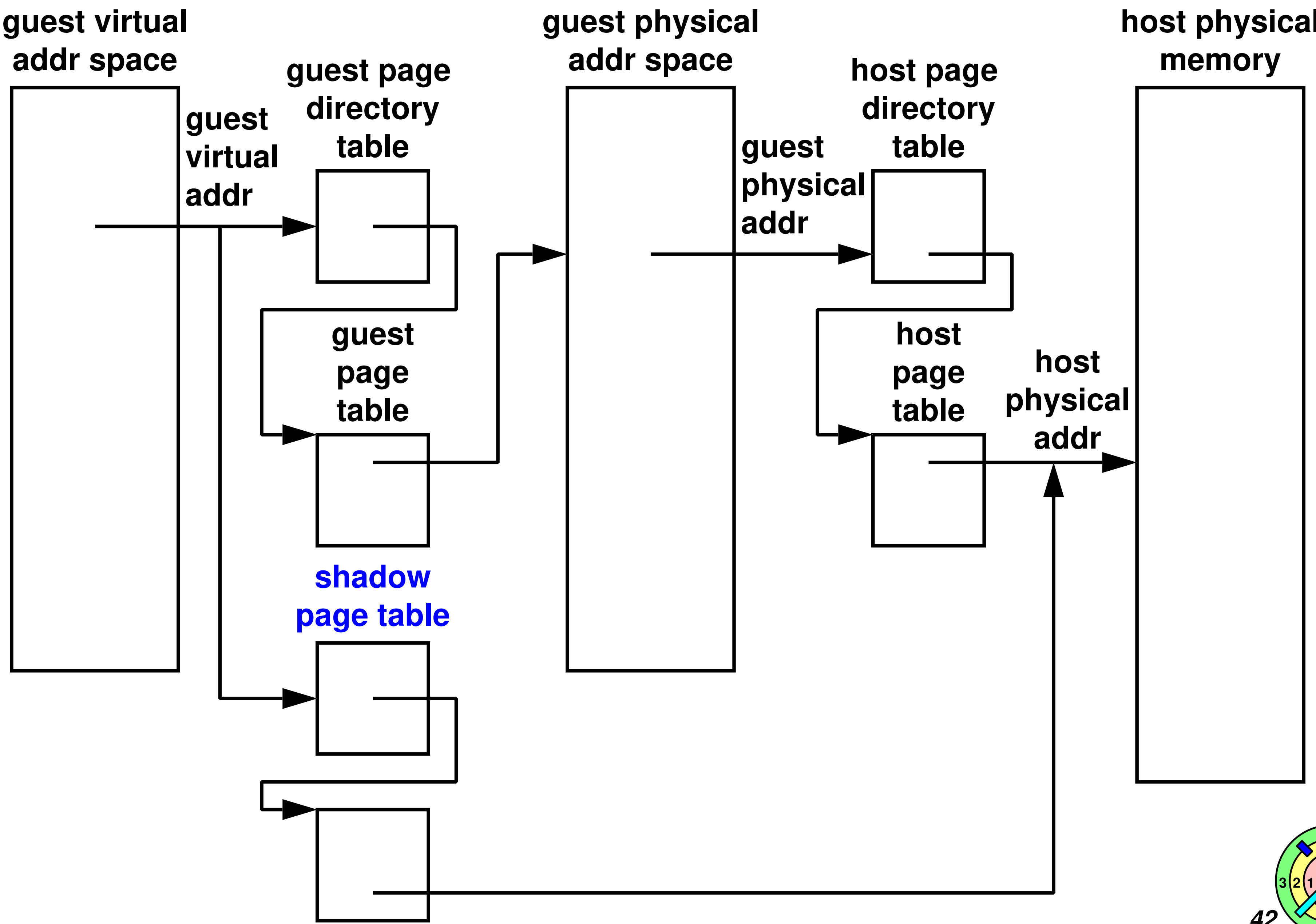
- Two approaches to letting application have control over memory:
  - *pinned pages:* application can *pin* virtual memory pages to physical page frames, preventing these pages from being evicted (unless absolutely necessary)
  - *user-level pagers:* application can specify a user-level page handler for a memory segment
    - on a page fault (or protection violation), kernel trap handler is invoked
    - instead of handling the fault, the kernel passes control to user-space handler to decide how to manage the trap, e.g.:
      - where to fetch the missing page
      - what action to take if the application was a sandbox
      - which page to replace
    - user-level page handler must itself be stored in pinned memory

Extra Slides

Virtual Machines & Virtual Memory



Shadow Page Table



Recoverable Virtual Memory

- Data structures that survive failures
  - want a consistent version of the data structure
  - user marks region of code as needing to be atomic
    - begin transaction, end transaction
  - if crash, restore state before or after transaction

Recoverable Virtual Memory

- On begin transaction:
  - snapshot data structure to disk
  - change page table permission to read-only
- On page fault:
  - mark page as modified by transaction
  - change page table permission to read-write
- On end transaction:
  - log changed pages to disk
  - commit transaction when all mods are on disk
- Recovery
  - read last snapshot + logged changes, if committed