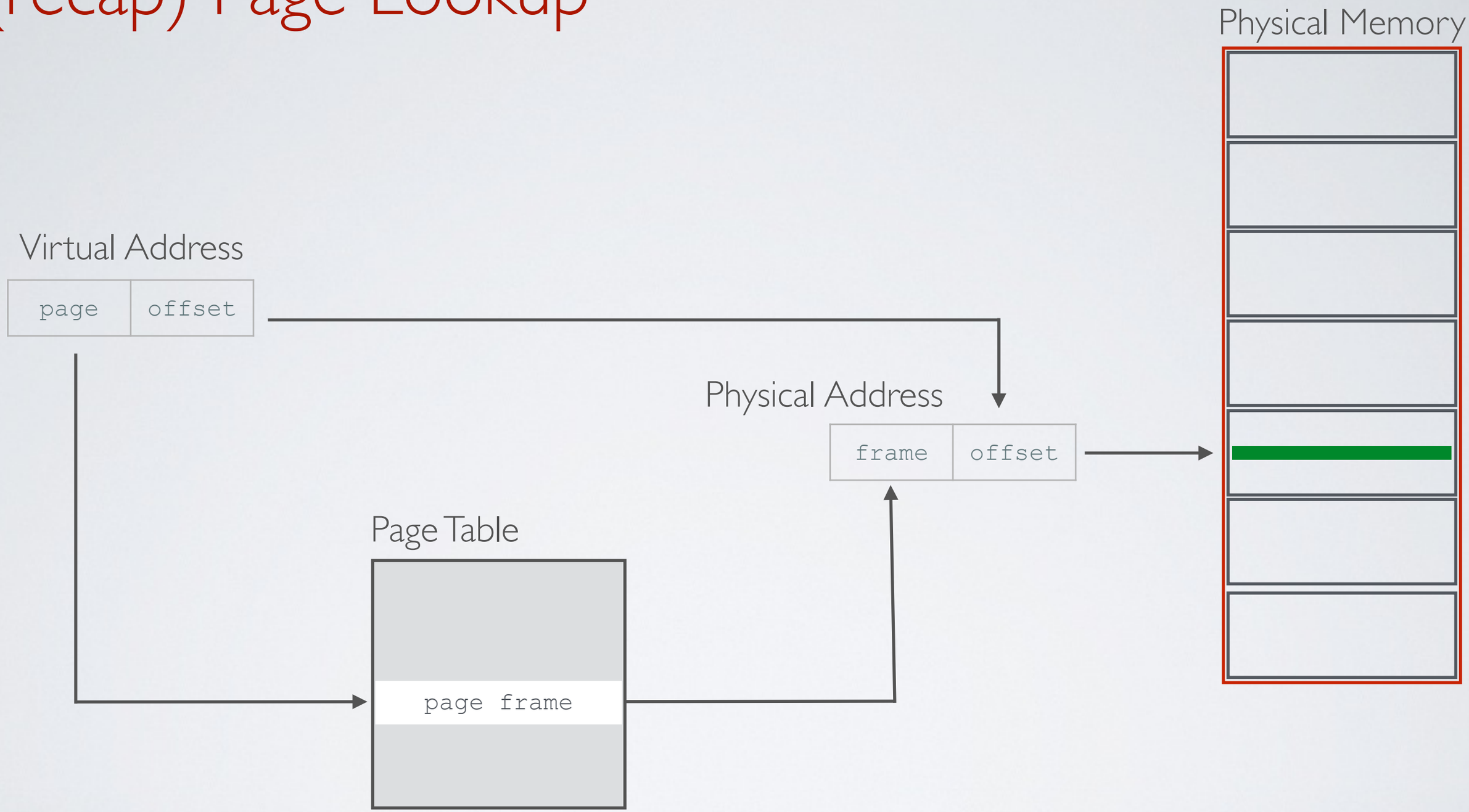
Paging in details

(recap) Page Lookup

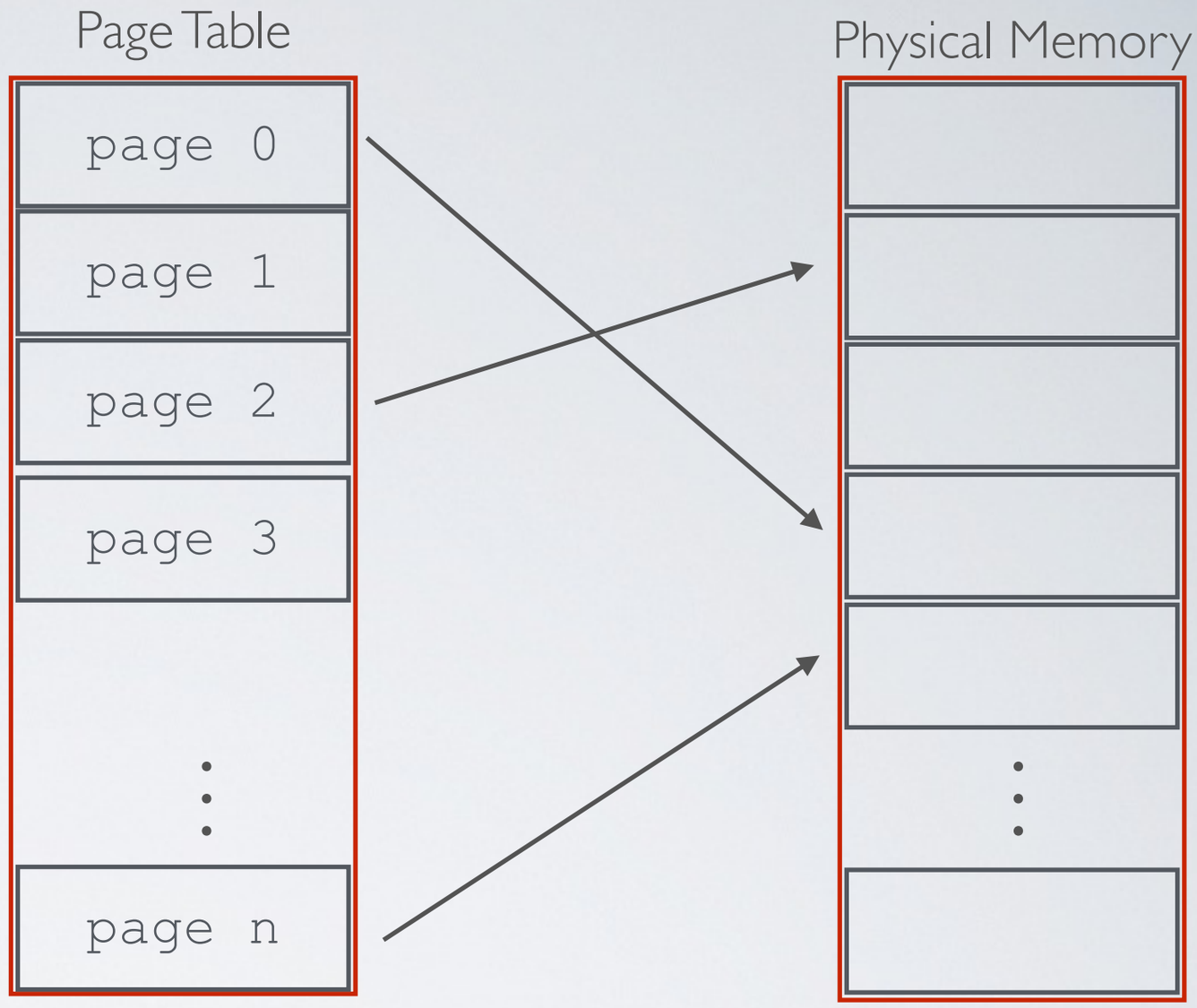


Improving paging

- Smaller page tables
- Faster address translation
- Larger virtual than physical memory (swapping)
- Advanced Functionality

Smaller page tables

The problem



Each process has a page table defining its address space

- Considering 32-bit address space with 4K
  the size of the pages table is $2^{32} / 2^{12} \times 4\ \mathrm{B} = 4\mathrm{MB}$ / process
  this is a big overhead!

Solution

- **Problem**: each process has a page table that maps all pages in its address space

✓ **Solution**: we only need to map the portion of the address space actually being used

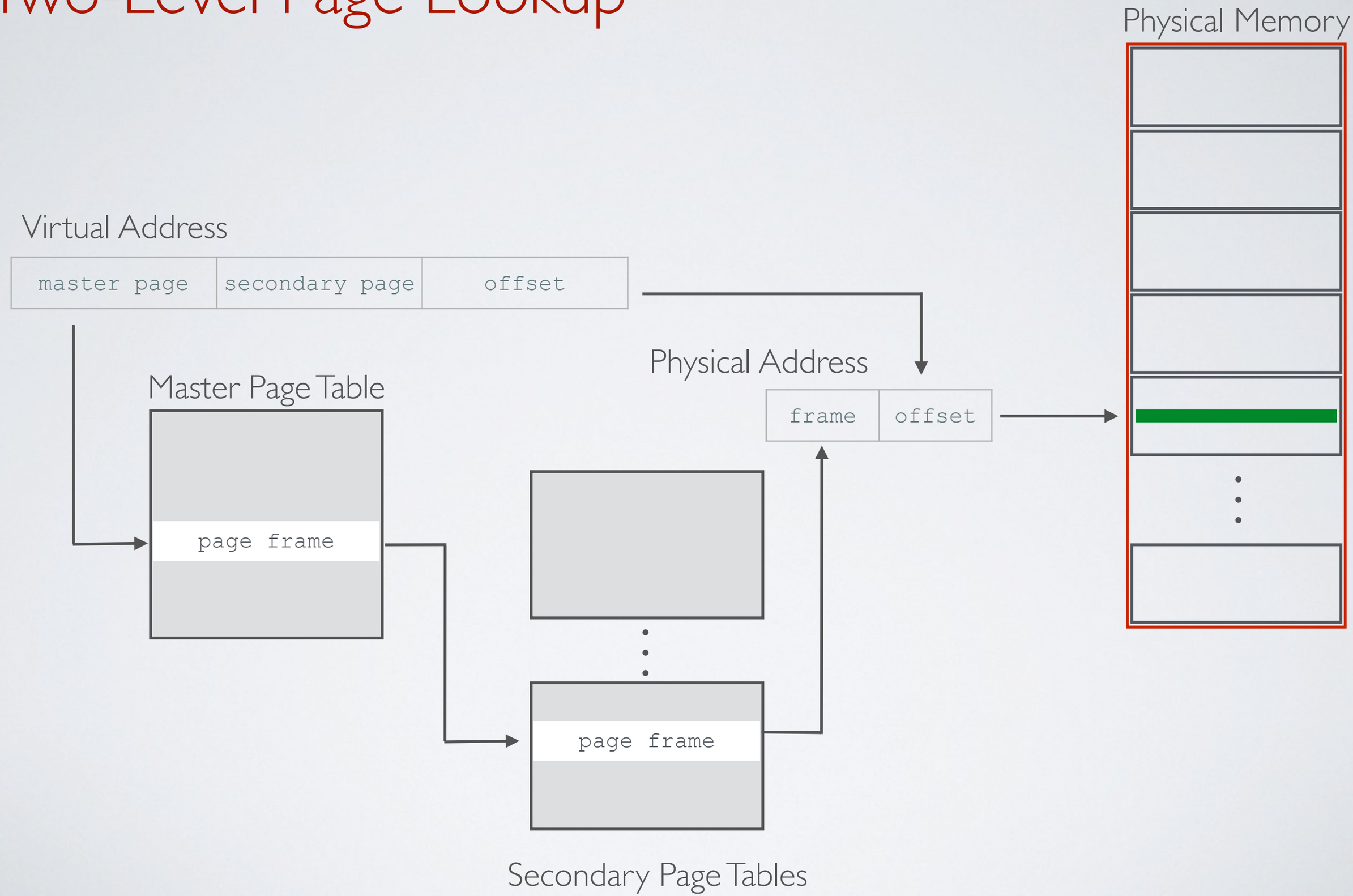
➡ Use another level of indirection : **two-level page tables**

Two-Level Page Tables

Virtual addresses have three parts

- **a master page number** i.e the index in the master page table that maps to a secondary page table
- **a secondary page number** i.e the index in the secondary page table that maps to the physical memory
- **an offset** that indicates where in physical page address is located

Two-Level Page Lookup



32 bits address space, 4K pages, 4 bytes/PTE

- How many bits in offset? 4K
  so the virtual address requires requires **12 bits for the offset**

- We want a master page table to fit in one page
  4K/4 bytes = 1K possible entries
  so the virtual address requires **10 bits for the master page index**

- We also want a secondary page table to fit in one page
  so the virtual address requires **10 bits for the secondary page index**

➡ 10 + 10 + 12 = 32 bits address
This is why 4K page size is recommended

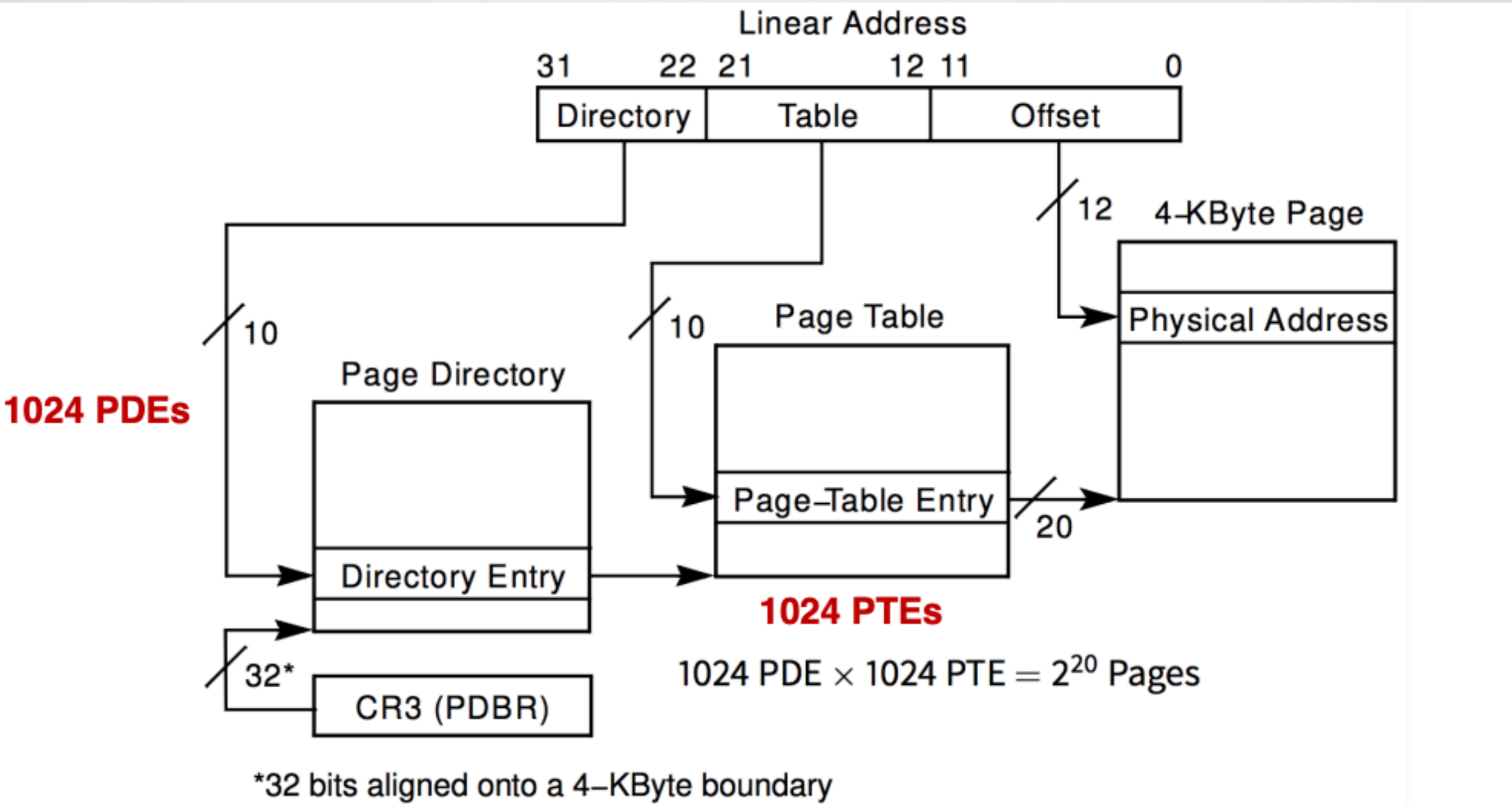
x86 Paging

- Paging enabled by bits in a control register `%cr0`
  (only privileged OS code can manipulate control registers)

- Register `%cr3` points to 4KB page directory
  (for Pintos, see `pagedir_activate()` in `userprog/pagedir.c`)

- Page directory has 1024 PDEs (Page Directory Entries) (see pagination details)
  - Each contains physical address of a page table
  - Each page table has 1024 PTEs (page table entries) and covers 4 MB of virtual memory
  - Each contains physical address of virtual 4K page

x86 Page Translation



Faster Address Translation

Efficient Translations

- **Problem** : expensive memory access
  - One-page table : one table lookup + one fetch
  - Two-page table (32 bits) : 2 table lookups + one fetch
  - 4-page table (64 bits) : 4 table lookup + one fetch

✓ **Solution** : **Translation Lookaside Buffer (TLB)**
cache translations in hardware to reduce lookup cost

Translation Lookaside Buffers (TLBs)

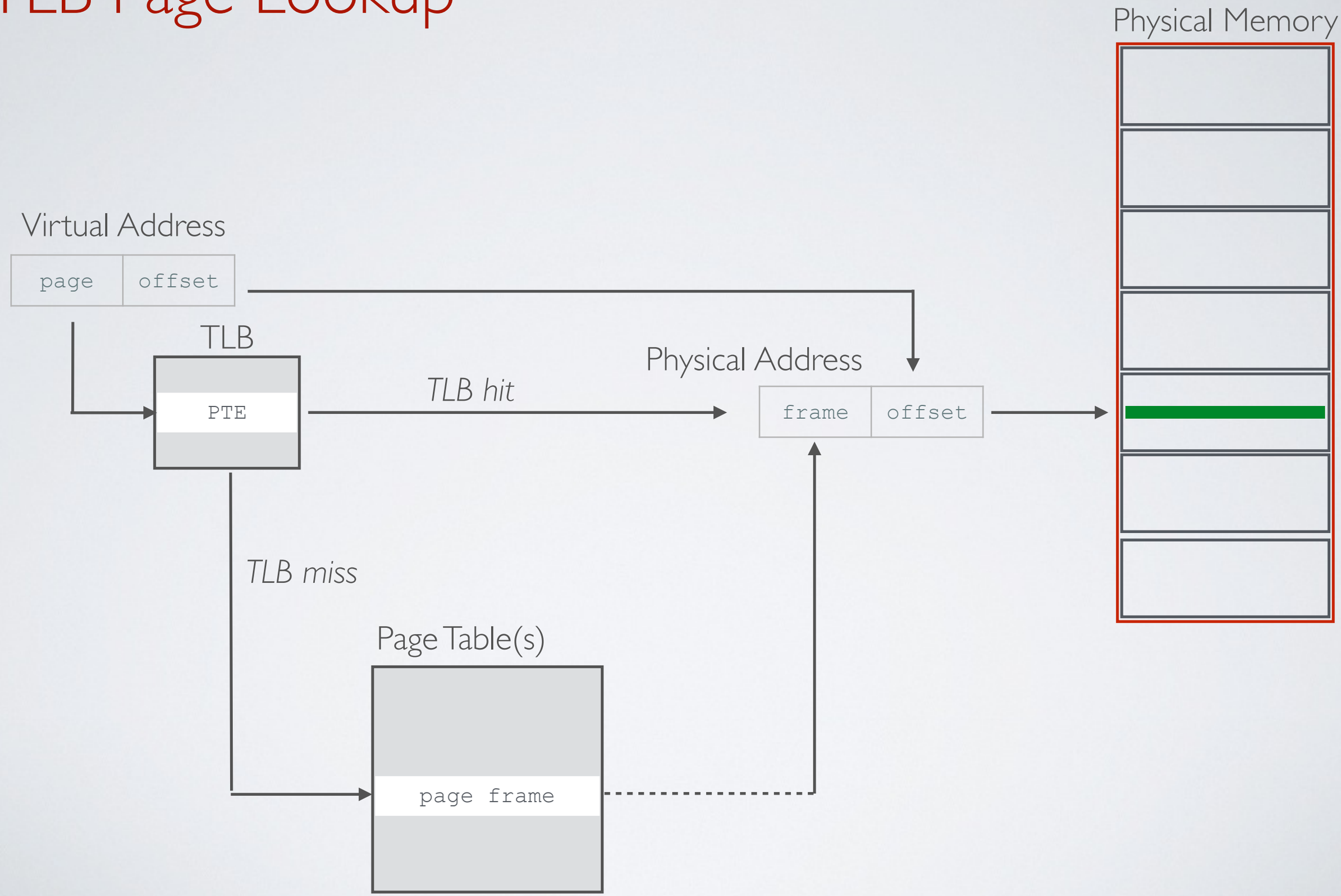
Translation Lookaside Buffers
special hardware to translate virtual page #s into PTEs (not physical addrs) in a single machine cycle

- Typically 4-way to fully associative cache (all entries looked up in parallel)

- Cache 32-128 PTE values (128-512K memory)

➡ TLBs exploit locality : processes only use a handful of pages at a time
TLB hit rate is a very important for performances
(>99% of translations)

TLB Page Lookup



Page lookup

Process is executing on the CPU, and it issues a read to an address
The read goes to the TLB in the MMU

1. TLB does a lookup using the page number of the address
2. Common case is that the page number matches, returning a page table entry (PTE) for the mapping for this address
3. TLB validates that the PTE protection allows reads (in this example)
4. PTE specifies which physical frame holds the page
5. MMU combines the physical frame and offset into a physical address
6. MMU then reads from that physical address, returns value to CPU

➡ This is all done by the hardware

TLB misses

1. TLB does not have a PTE mapping this virtual address

2. PTE in TLB, but memory access violates PTE protection bits

Swapping

Paged Virtual Memory

➡ The OS can use disk to simulate larger virtual than physical memory
the pages can be moved between memory and disk (a.k.a paging in/out)

Paging process over time

- Initially, pages are allocated from memory
- When memory fills up, allocating a page requires some other page to be evicted
- Evicted pages go to disk, more precisely to the swap file/backing store
- Done by the OS, and transparent to the application

Extreme design : **demand paging** paging in a page from disk into memory only if an attempt is made to access it (the main memory becomes a cache for disk)

Page Faults

Read/write/execute protection bits : operation not permitted on page

➡ The TLB traps to the OS and the OS usually will send fault back up to process, or might be playing games e.g., copy on write, mapped files (coming later in this lecture)

Invalid bits : 2 possible reasons

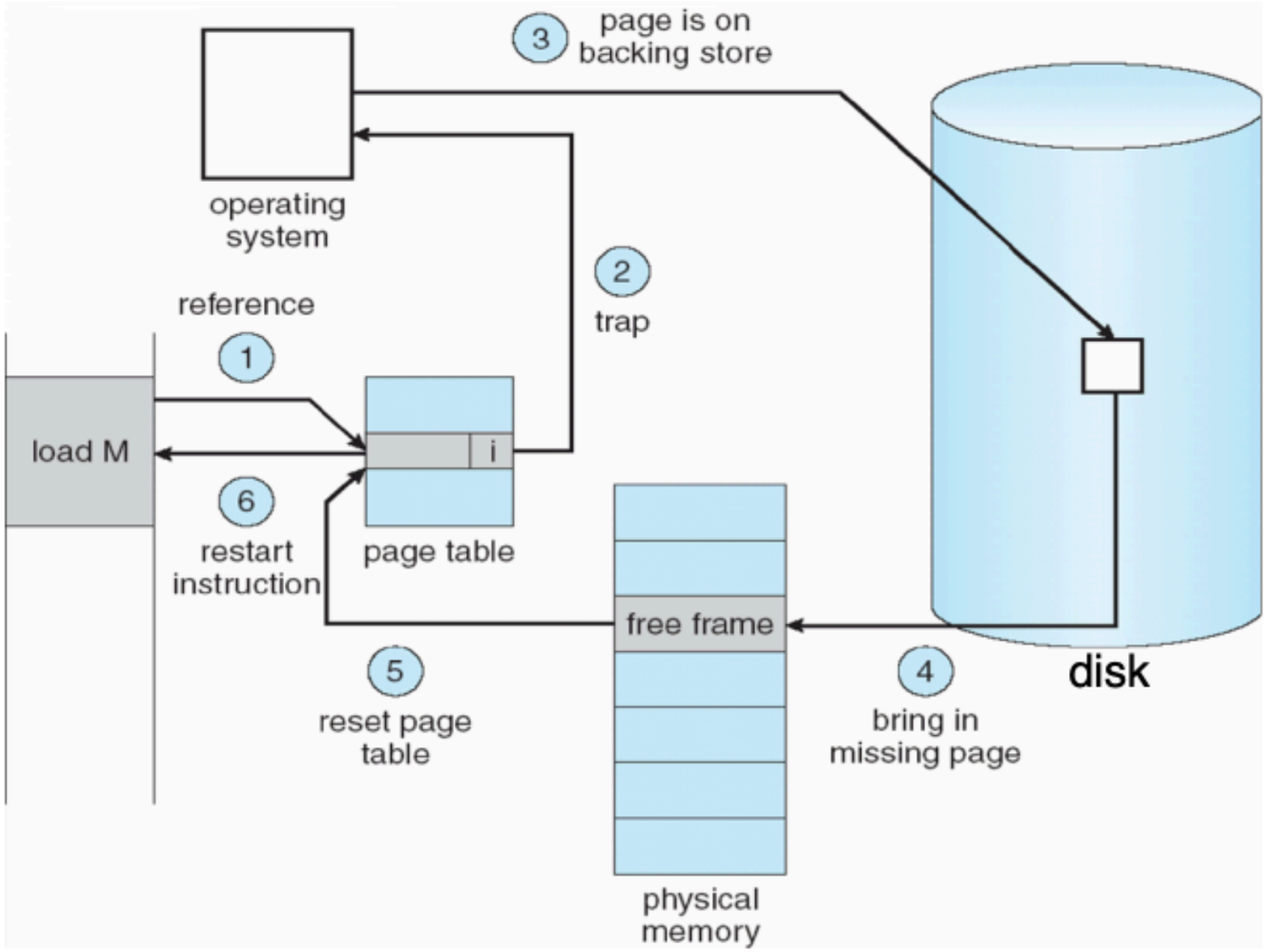
1. Virtual page not allocated

➡ The TLB traps to the OS and the OS sends fault to process (e.g., segmentation fault)TLB traps to the OS (software takes over)

2. Virtual page not allocated in the address space but swapped on disk

➡ The TLB traps to the OS and the OS sends allocates frame, reads from disk, maps PTE to physical frame

Page Faults

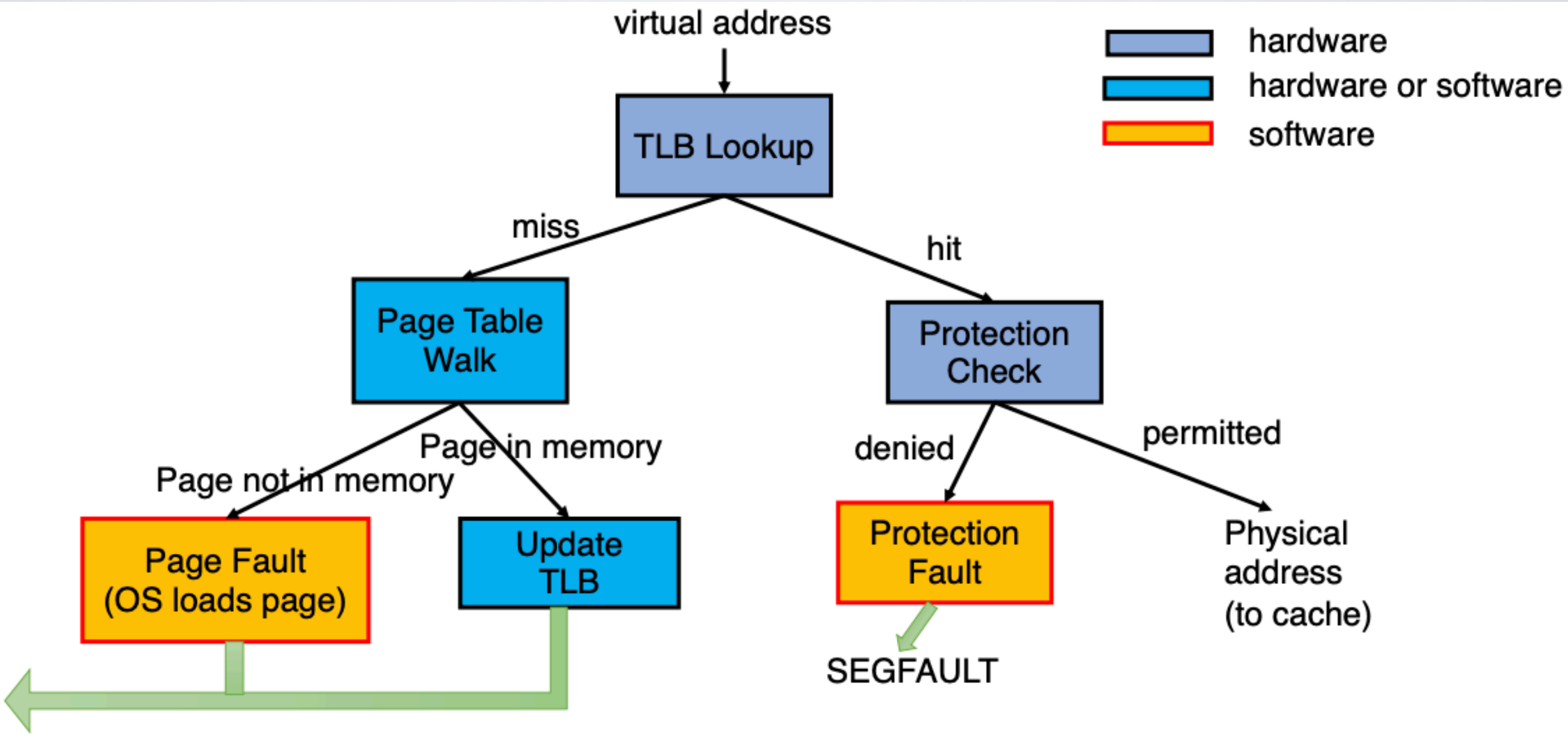


1. When the OS evicts a page, it sets the PTE as invalid and stores the location of the page in the swap file in the PTE
2. When a process accesses the page, the invalid PTE causes a trap (page fault)
3. The trap will run the OS page fault handler
4. Handler uses the invalid PTE to locate page in swap file
5. Reads page into a physical frame, updates PTE to point to it
6. Restarts process

There is more to the topic of swapping

➡ **More on swapping** in the next lecture
Mechanisms and policy to evict page from memory

Address Translation : Putting It All Together



Advanced Functionality

- Shared memory
- Copy on Write
- Mapped files

Sharing

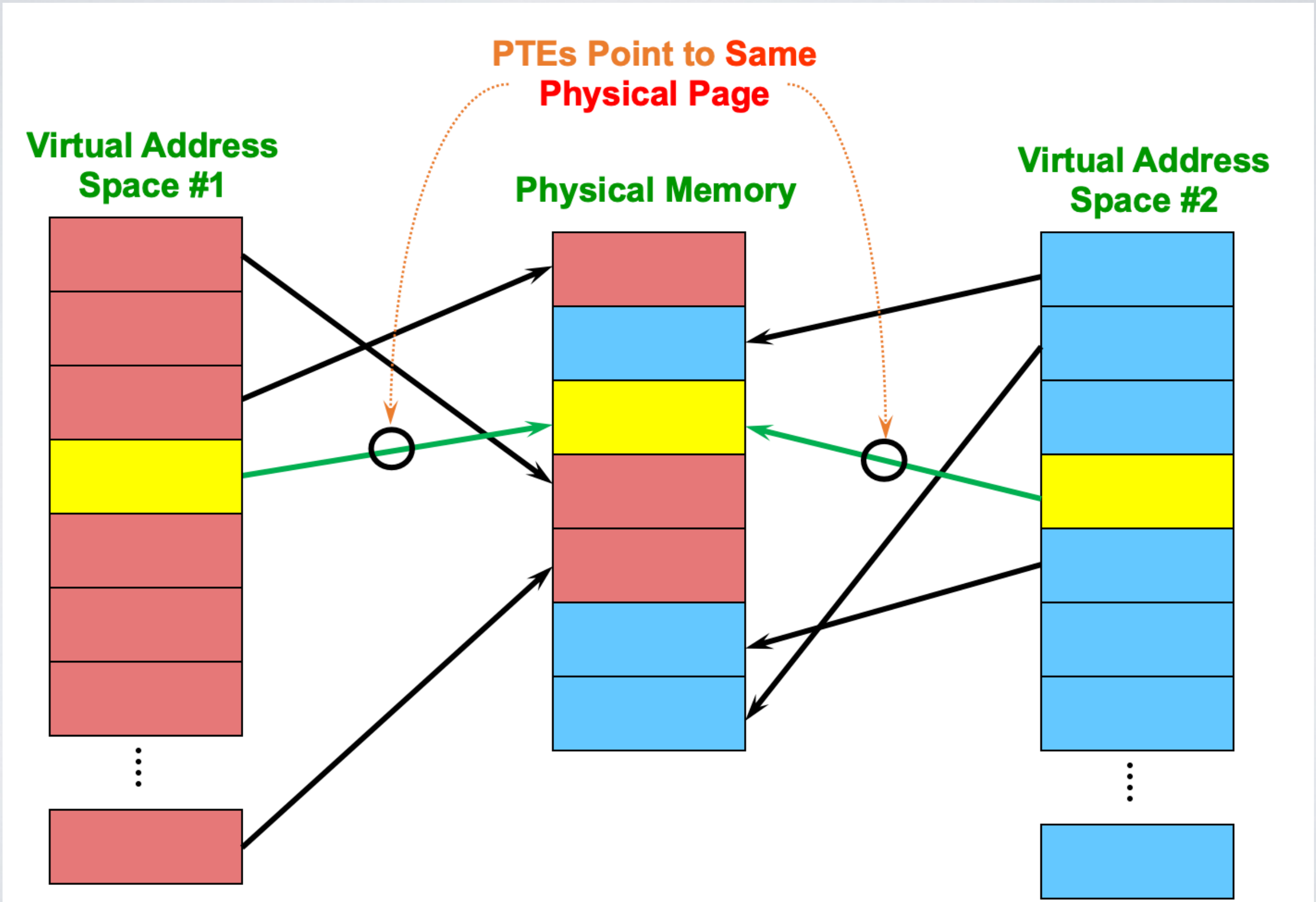
Private VM spaces protect applications from each other

- ◉ But this makes it difficult to share data between processes

- ✓ Have shared memory to allow processes to share data using direct memory references (synchronization required)

- ➡ See Unix System V Shared Memory Segment (`shmget`)

Sharing Pages



Shared memory address mapping

Can map shared memory at same or different virtual addresses in each process' address space ?

- **Different Mapping**
  Flexible but pointers inside shared memory are invalid

- **Same Mapping**
  Less flexible but shared pointer are valid

Copy on Write

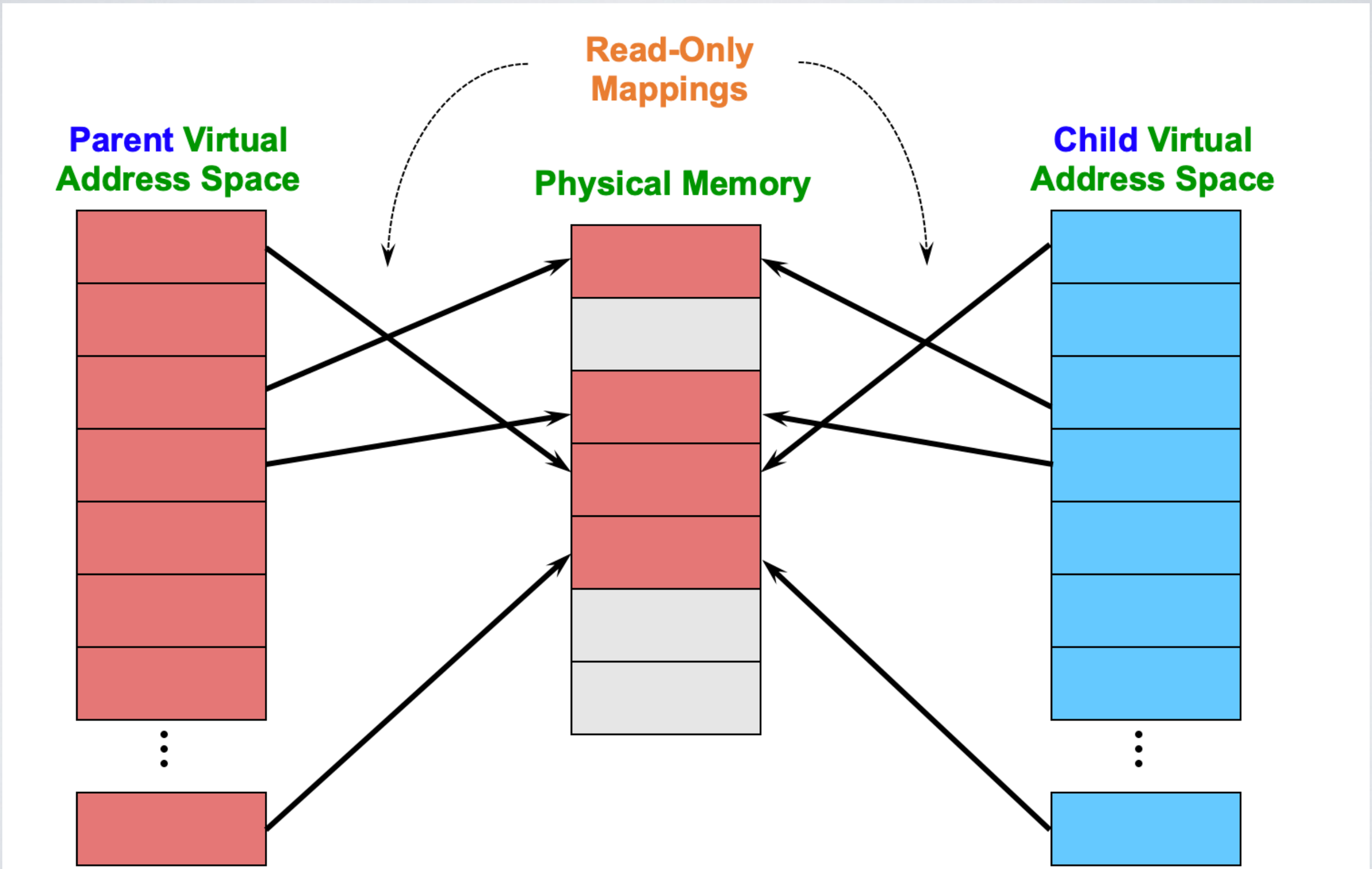
- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()

➡ Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether

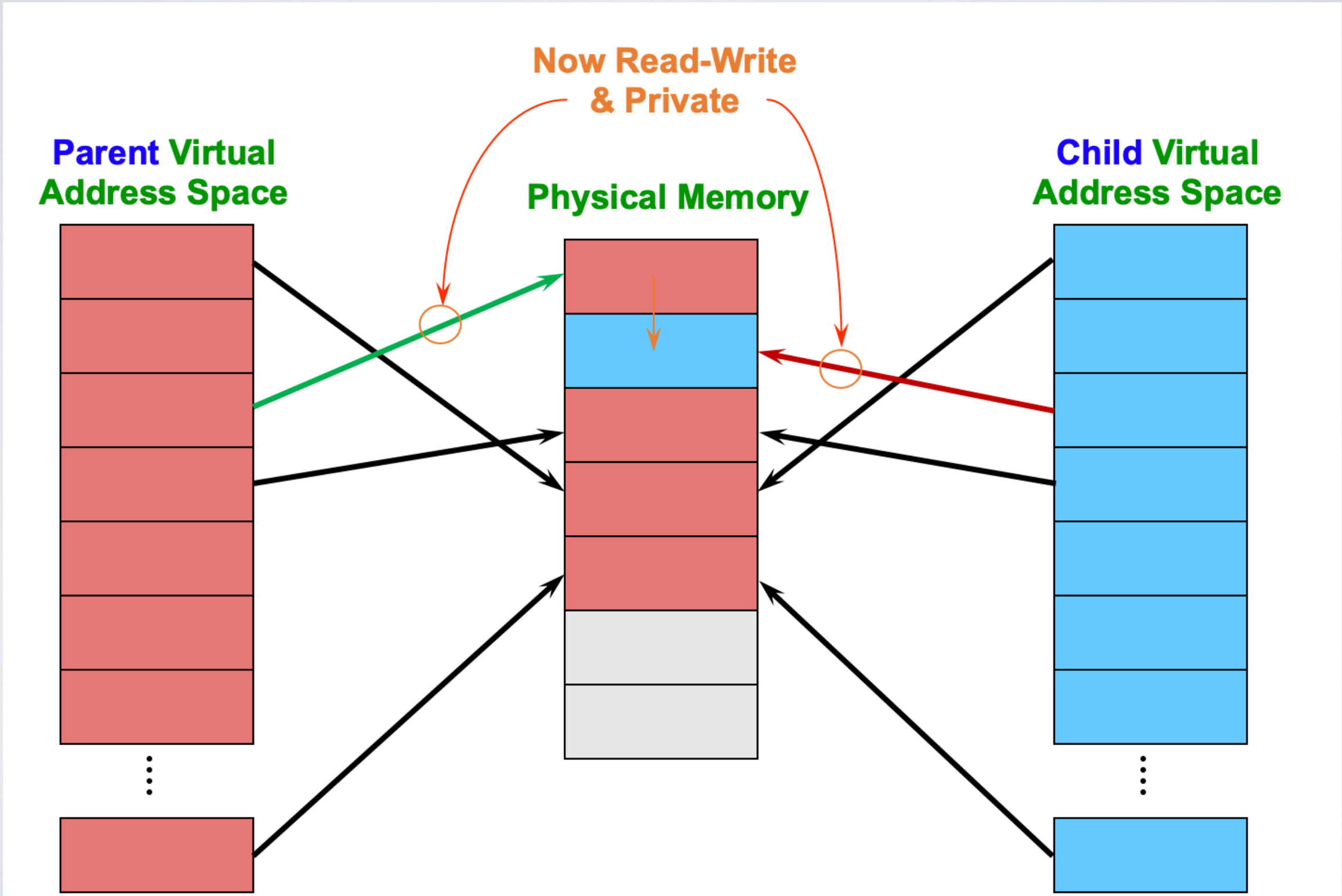
- Create shared mappings of parent pages in child virtual address space (instead of copying pages)

- Shared pages are protected as read-only in parent and child
  Any write operation generates a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction

Example - Fork step



Example - Write step



Mapped Files

Mapped files enable processes to do file I/O using loads and stores
Instead of "open, read into buffer, operate on buffer, …"

➡ Bind a file to a virtual memory region (see Unix `mmap`)

- PTEs map virtual addresses to physical frames holding file data
- Virtual address base + N refers to offset N in file

Initially, all pages mapped to file are invalid (similar to a swapped page)

- OS reads a page from file when invalid page is accessed
- OS writes a page to file when evicted, or region unmapped
- If page is not dirty (has not been written to), no write needed (another use of the dirty bit in PTE)

Advantages and drawbacks of mapped files

➡ File is essentially backing store for that region of the virtual address space (instead of using the swap file)

✓ Uniform access for files and memory (just use pointers)

◉ Process has less control over data movement
OS handles faults transparently

◉ Does not generalize to streamed I/O (pipes, sockets, etc.)

Next time

- **More on swapping**
  Mechanisms and policy to evict page from memory

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*