

# CE 440 Introduction to Operating System

## Lecture 11: Virtual Memory II Fall 2025

**Prof. Yigong Hu**



Slides courtesy of Manuel Egele, Ryan Huang and Baris Kasikci

# Administrivia

## **Lab 2 is out**

- Due Friday 11/07 11:59 pm
- Start early!

## **Homework 3 is released**

- Related to midterm

# Lecture Overview

**Today we'll cover more paging mechanisms:**

**Two-level page table**

**Address translation**

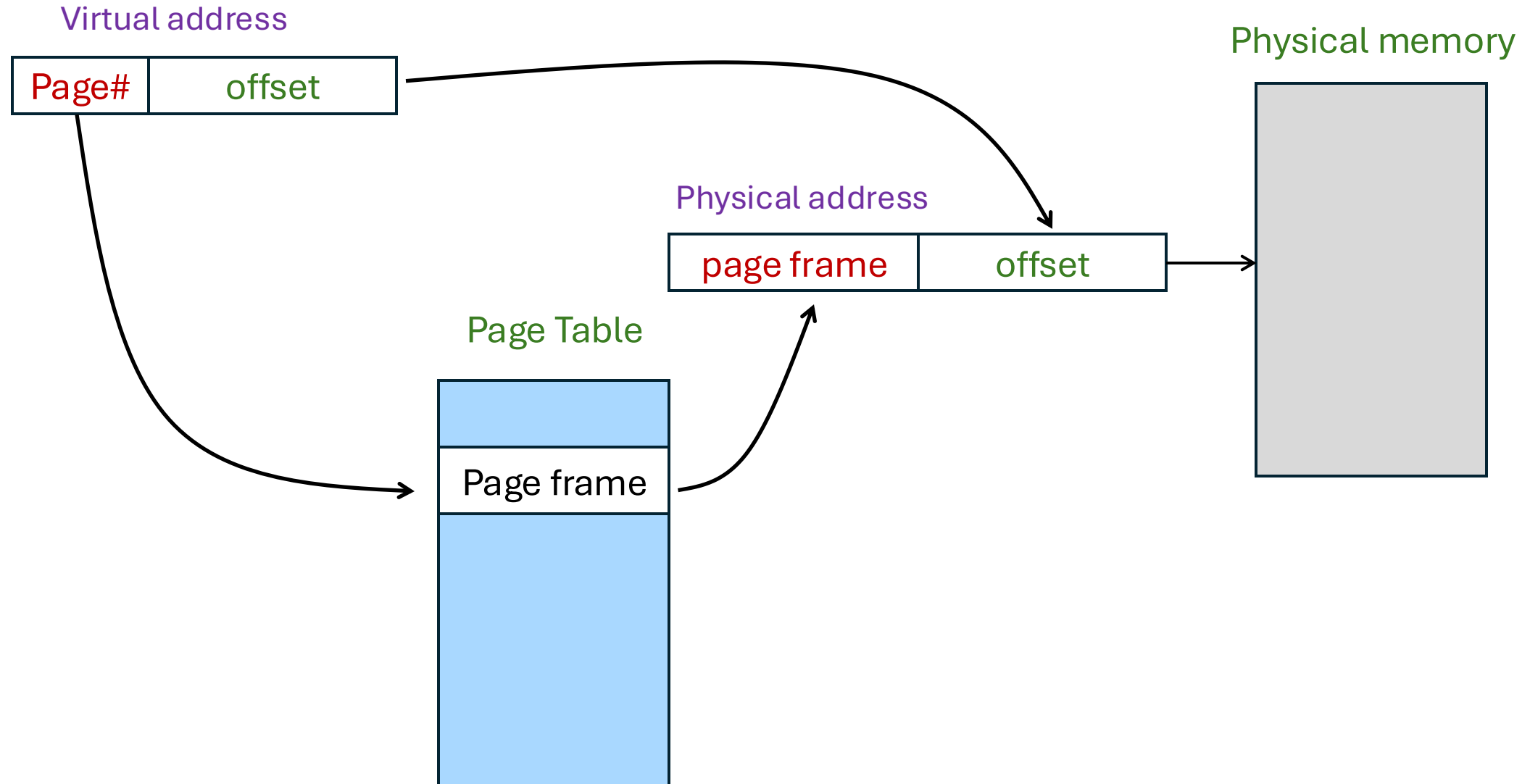
**Optimizations**

- Managing page tables (space)
- Efficient translations (TLBs) (time)
- Demand paged virtual memory (space)

**Advanced functionality**

- Sharing memory
- Copy on Write
- Mapped files

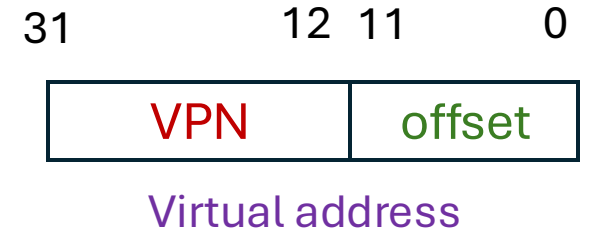
# Recap: Page Lookups



# Recap: Paging Example

## Pages are 4K

- VPN is 20 bits ( $2^{20}$  VPNs), offset is 12 bits



## Virtual address is 0x7468

- Virtual page is 0x7, offset is 0x468

## Page table entry 0x7 contains 0x2

- Physical page number is 0x2
- Seventh virtual page is at address 0x2000 (2nd physical page)

**Physical address = 0x2000 + 0x468 = 0x2468**

# Problem of Page Tables

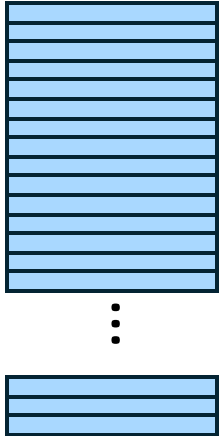
**Size of the page table for a 32-bit address space w/ 4K pages**

- $2^{32} / 2^{12} \times 4 \text{ B} = 4\text{MB}$
- This is far far too much overhead for each process

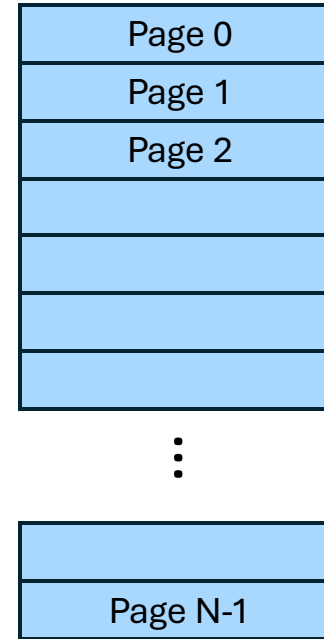
**How can we reduce this overhead?**

# Page Table Evolution

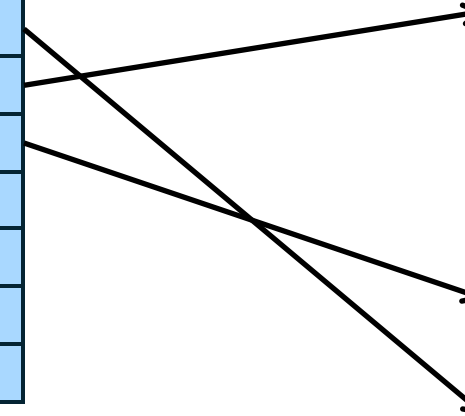
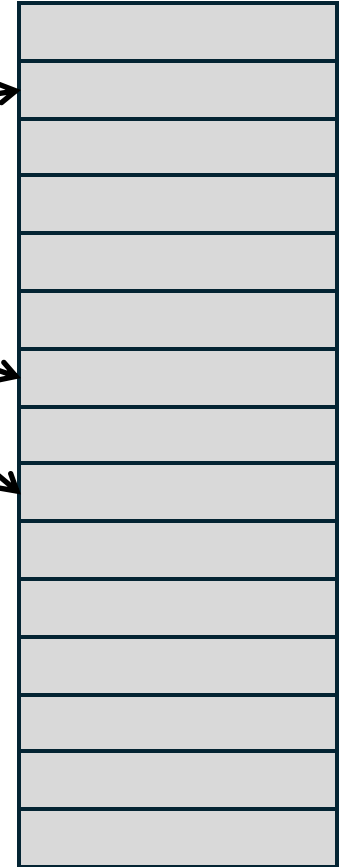
Linear (Flat)  
Page Table



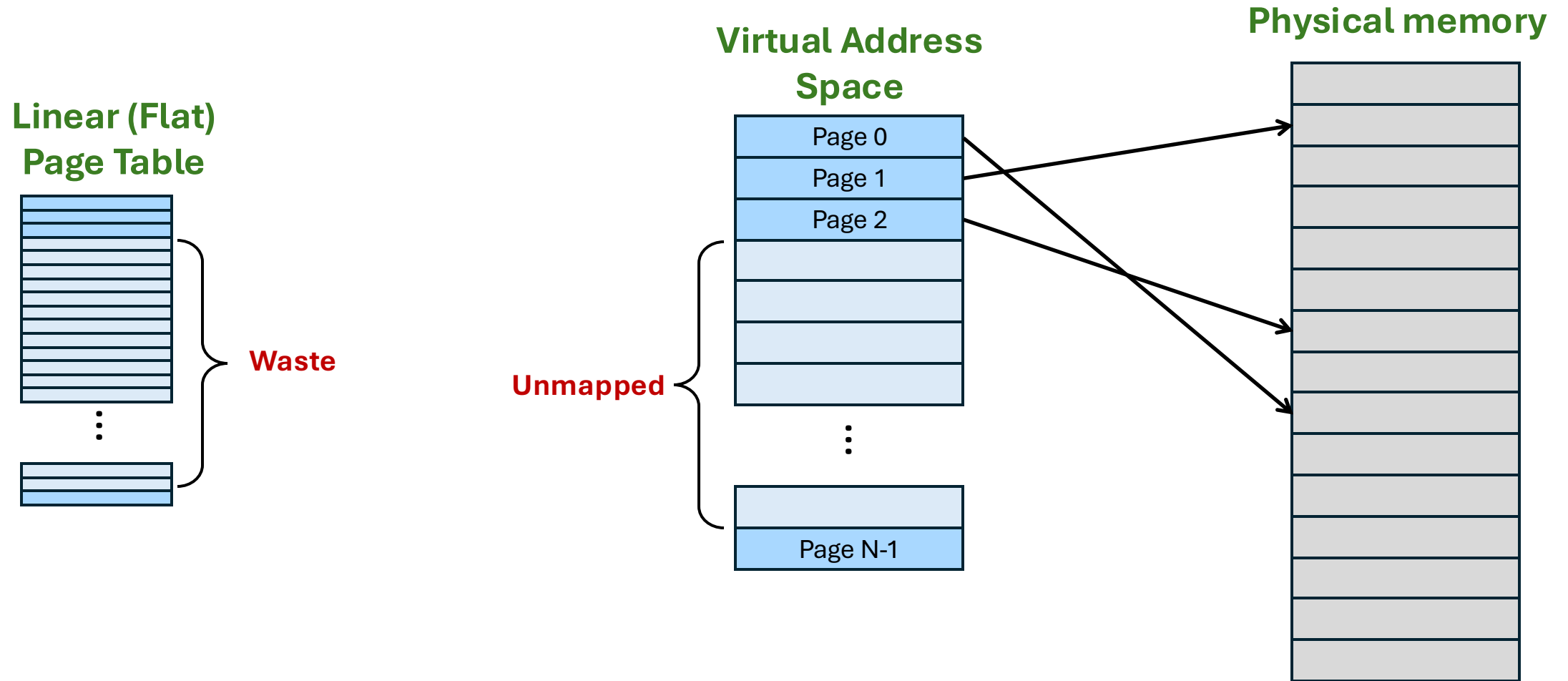
Virtual Address  
Space



Physical memory



# Page Table Evolution





# Solution

## Size of the page table for a 32-bit address space w/ 4K pages

- $2^{32} / 2^{12} \times 4 \text{ B} = 4 \text{ MB}$
- This is far far too much overhead for each process

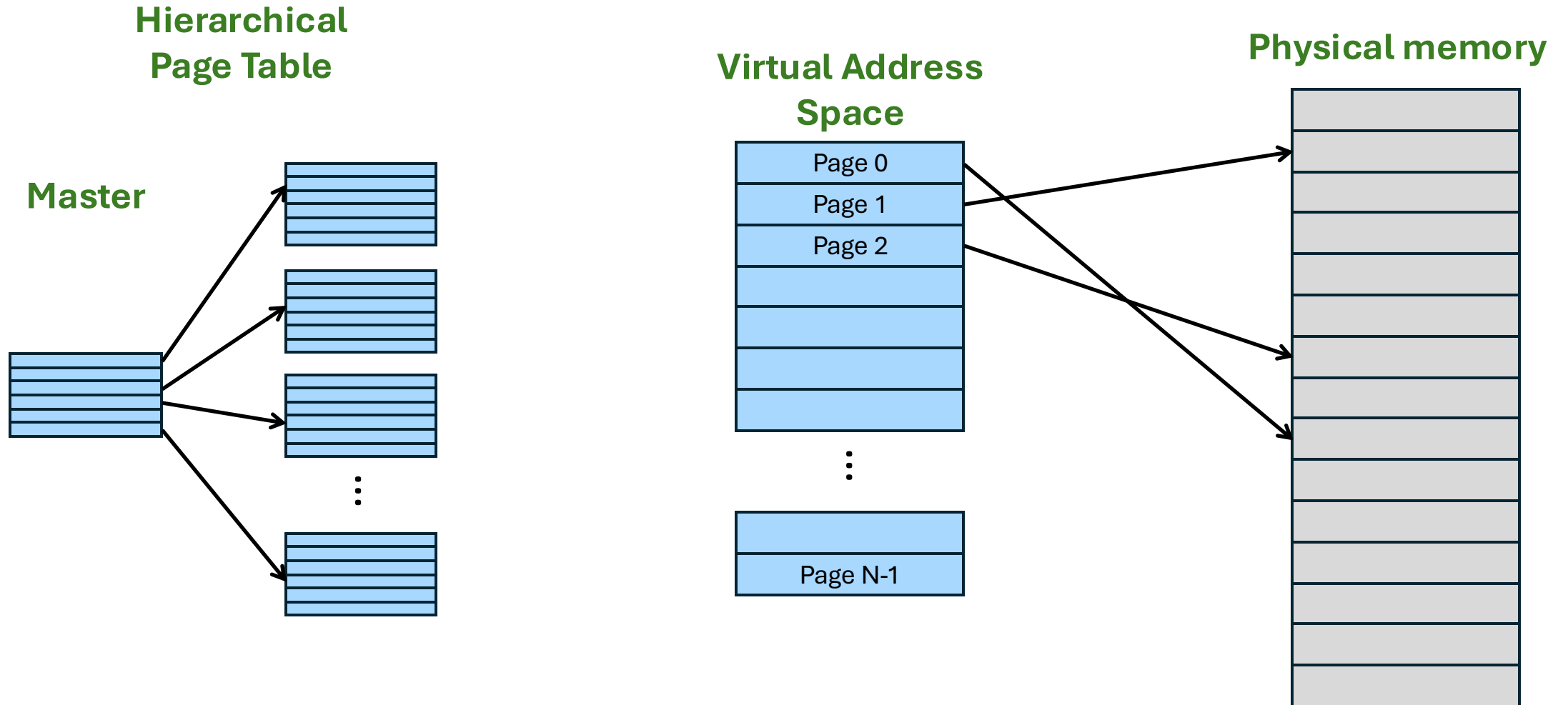
## How can we reduce this overhead?

- **Observation:** only need to map the portion of the address space actually being used (tiny fraction of entire addr space)

## How do we only map what is being used?

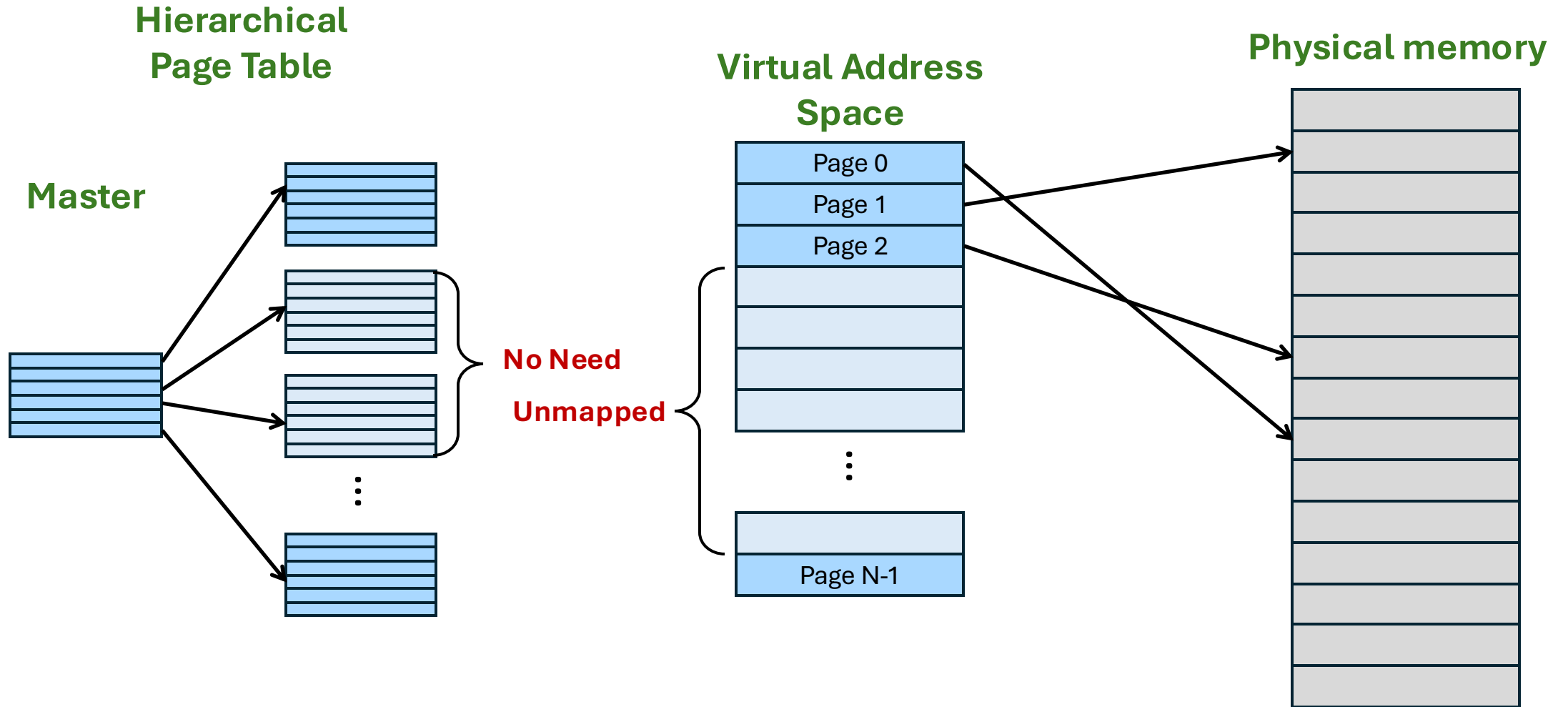
- Can dynamically extend page table...
- Does not work if addr space is sparse (internal fragmentation)

# Page Table Evolution



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

# Page Table Evolution



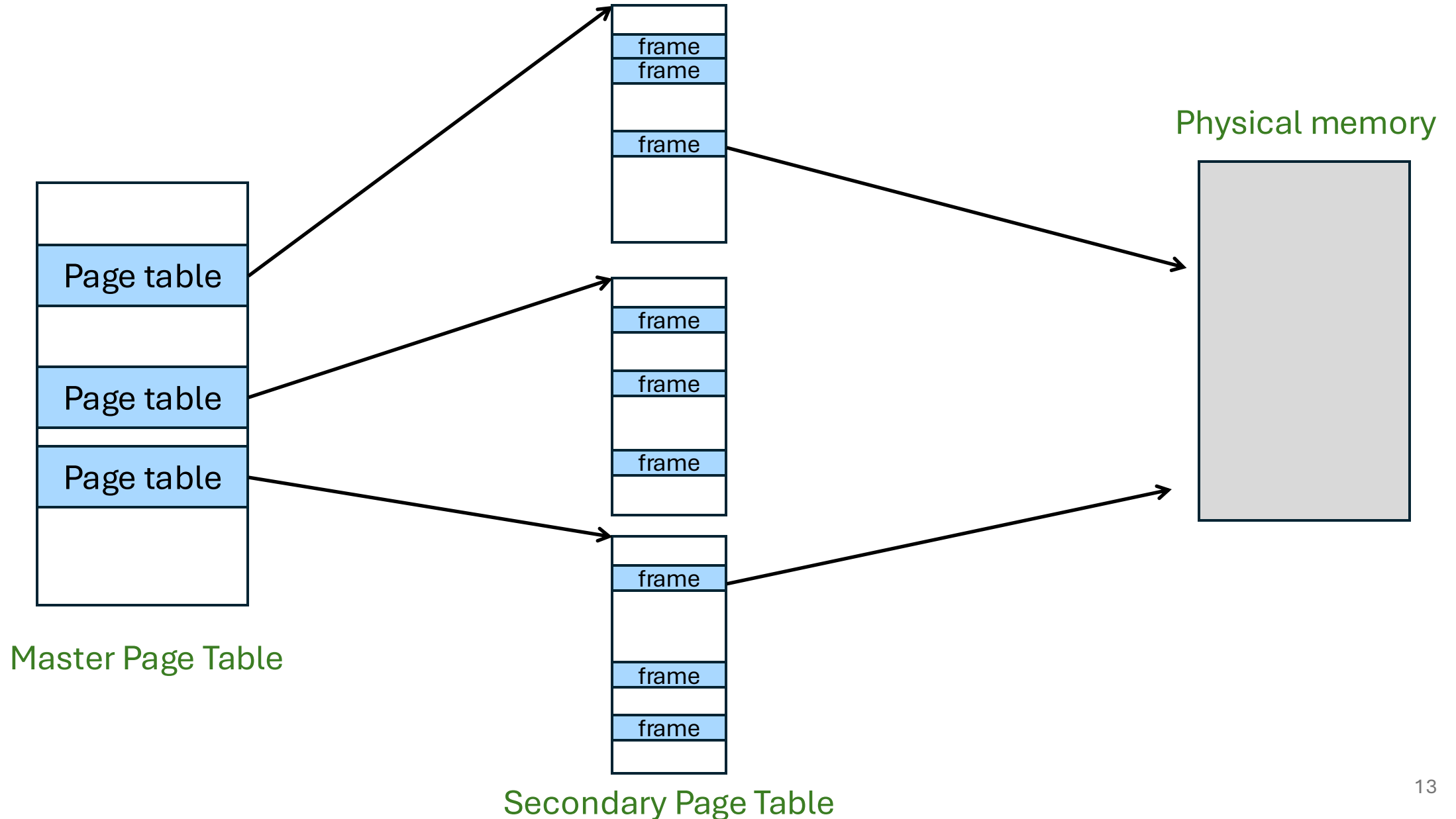
Use another level of indirection: **two-level page tables**

# Two-Level Page Tables

## Two-level page tables

- One master page table that maps VAs to some secondary page table
- A secondary page table maps page number to some physical page
- Offset indicates where in physical page the address is located

# Two-Level Page Table Translation

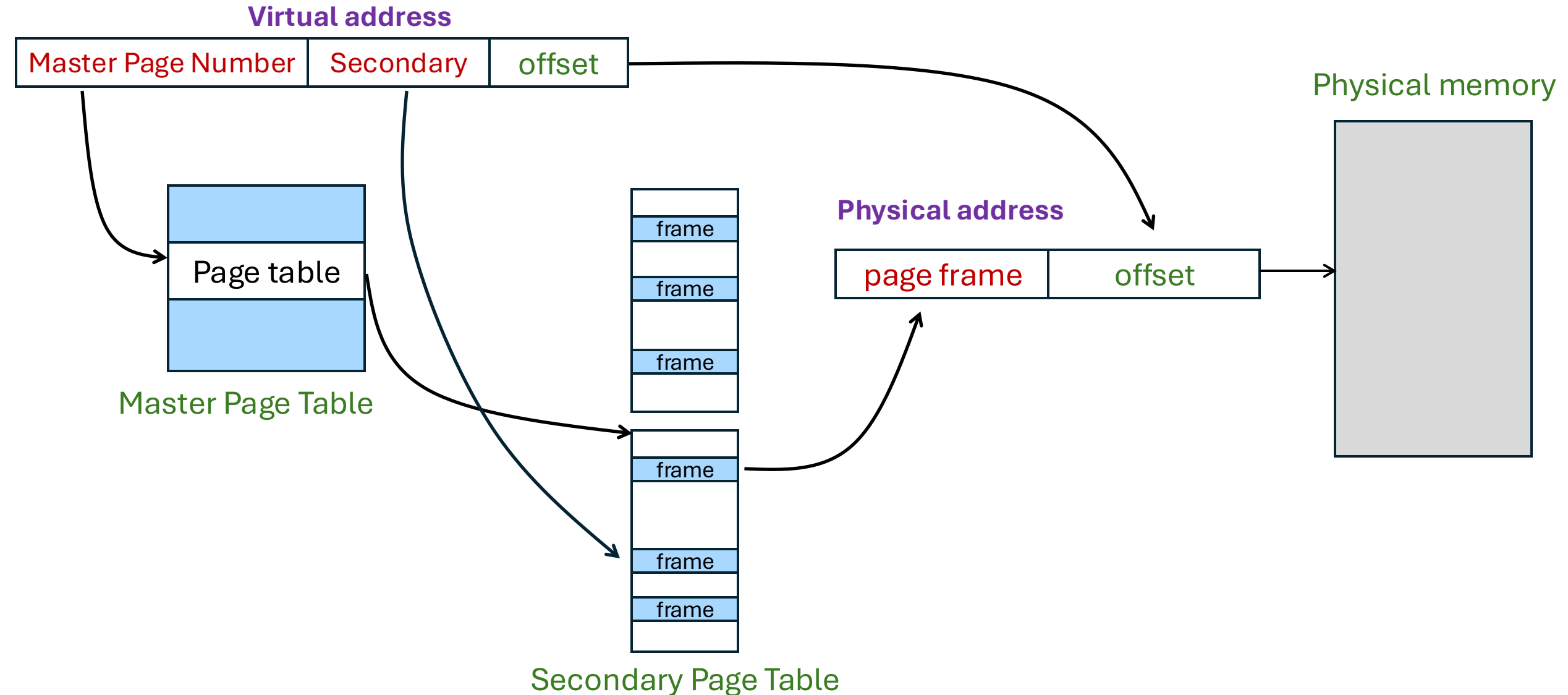


# Two-Level Page Tables

## Two-level page tables

- One master page table that maps VAs to some secondary page table
- A secondary page table maps page number to some physical page
- Offset indicates where in physical page the address is located
- Virtual addresses (VAs) have **three** parts:
  - Master page number, secondary page number, and offset

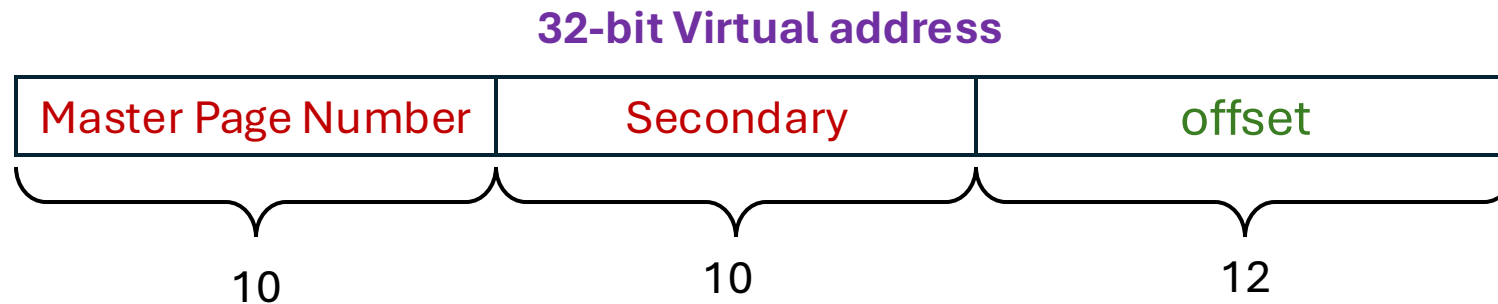
# Two-Level Page Tables Look Up



# Two-Level Page Table

## Example

- 4KB-sized pages, 4 bytes/PTE
- How many bits in offset?  $\log_2(4K) = 12$  bits
- We want the master page table in one page:  $4K/4$  bytes = 1K entries
- Hence, 1024 secondary page tables. How many bits?
- Master  $\log_2(1K) = 10$ , offset = 12, inner =  $32 - 10 - 12 = 10$  bits





# Wait a Second...

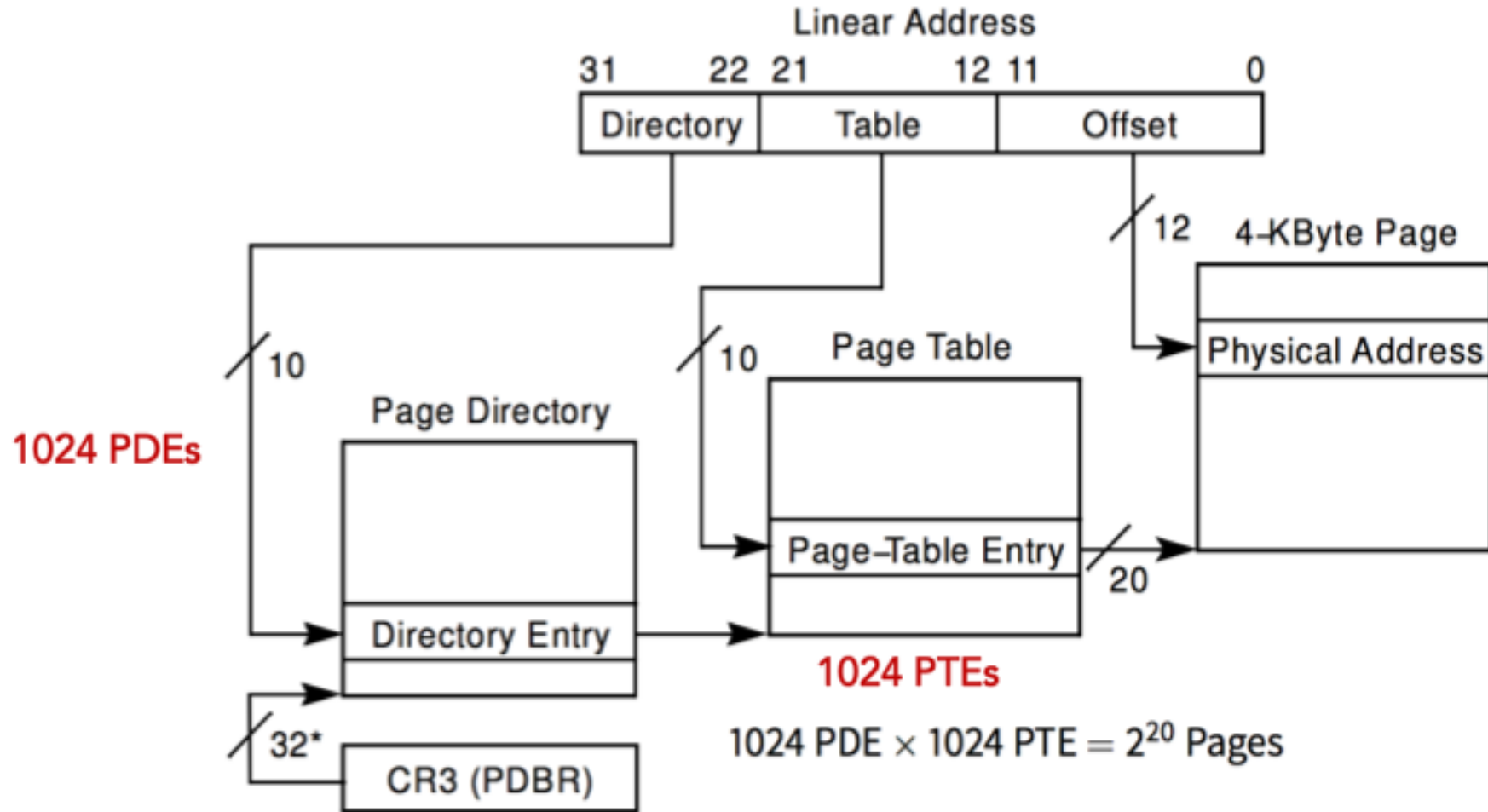
**We introduced two-level page tables to reduce the overhead of storing page tables**

- Each page table costs  $2^{32} / 2^{12} \times 4 \text{ B} = 4\text{MB}$  to store

**But even if we add another level, isn't the overhead the same?**

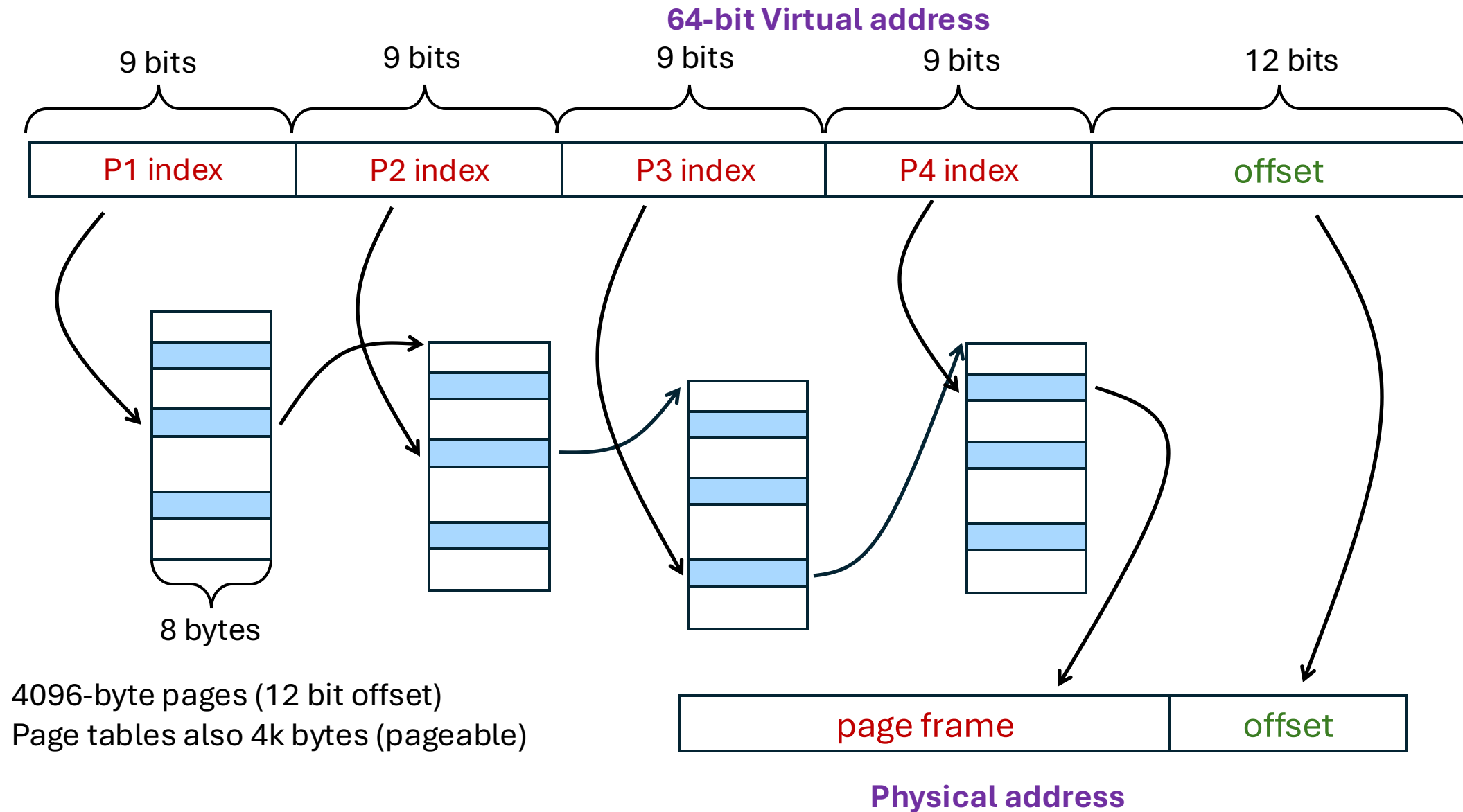
- 1024 secondary page tables
- Each secondary page table has  $2^{10}$  PTEs, thus has a size of 4KB
- Total size of these page tables is  $1024 \times 4\text{KB} = 4\text{MB}$ ...
- In fact, we also have one master page table, which has a size of 4KB...

# X86 Page Translation



\*32 bits aligned onto a 4-KByte boundary

# X86\_64: Four Level Page Table



# Addressing Page Tables

**Where do we store page tables (which address space)?**

## **Physical memory**

- Easy to address, no translation required
- But, allocated page tables consume memory for lifetime of VAS

## **Virtual memory (OS virtual address space)**

- Cold (unused) page table pages can be paged out to disk
- But, addressing page tables requires translation
- How do we stop recursion?
- Do not page the outer page table (called **wiring**)

**If we're going to page the page tables, might as well page the entire OS address space, too**

- Need to wire special code and data (fault, interrupt handlers)

# Lecture Overview

Today we'll cover more paging mechanisms:

Two-level page table

**Address translation**

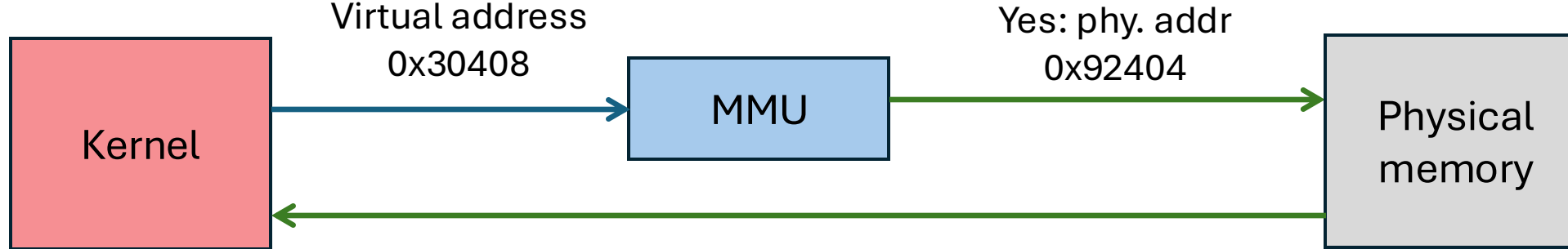
**Optimizations**

- Managing page tables (space)
- Efficient translations (TLBs) (time)
- Demand paged virtual memory (space)

**Advanced functionality**

- Sharing memory
- Copy on Write
- Mapped files

# Recap: Memory Translation



**The MMU must translate virtual address to physical address on:**

- Every instruction fetch, load, store

**What does the MMU need to do to translate an address?**

- Page Table
  - Read PTE from memory, check valid, merge address
  - Set “accessed” bit in PTE, Set “dirty bit” on write
- 2-level Page Table
  - Read and check first level
  - Read, check, and update PTE
- N-level Page Table ...

# Efficient Translations

**Our original page table already doubled the cost of memory access**

- One lookup into the page table, another to fetch the data

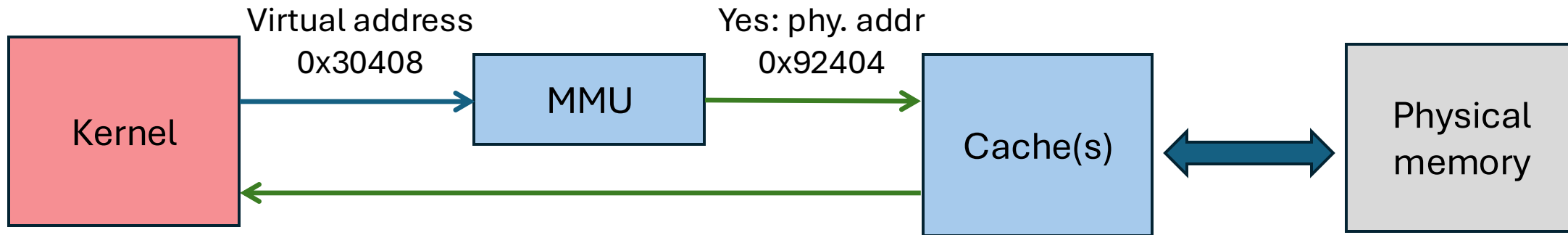
**Now two-level page tables triple the cost!**

- Two lookups into the page tables, a **third** to fetch the data
- Worse, 64-bit architectures support 4-level page tables
- And this assumes the page table is in memory

**How can we use paging but also reduce lookup cost?**

- Cache translations in hardware
- Translation Lookaside Buffer (TLB)
- TLB managed by Memory Management Unit (MMU)

# Caching



**Cache : a memory region that can be accessed more quickly than the main memory**

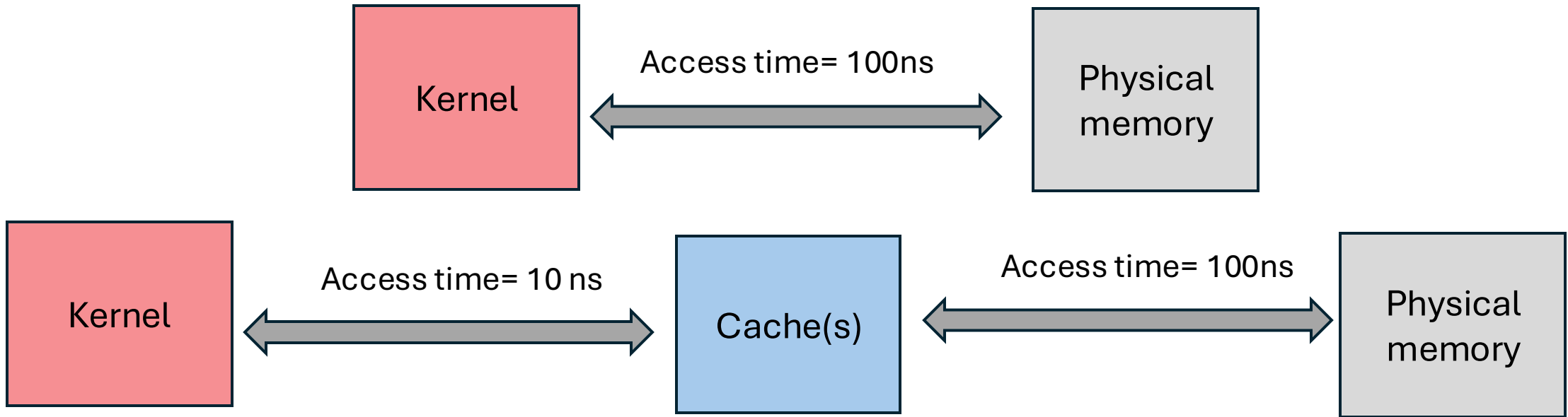
- Caches usually index data using the physical address returned by the MMU
- Store recently used data, instructions, or even address translations (TLB)
- **Cache hit**: data returned immediately.
- **Cache miss**: data fetched from main memory and inserted into cache.

**Improve performance because of program locality:**

- **Temporal Locality** (Locality in Time):
  - Keep recently accessed data items closer to processor
- **Spatial Locality** (Locality in Space):
  - Move contiguous blocks to the upper levels



# Caching Example



**Average Memory Access Time = (Hit Rate x HitTime) + (Miss Rate x MissTime)**

**If HitRate = 90%, what is AMAT?**

- $(0.9 \times 10) + (0.1 \times 110) = 20 \text{ ns}$

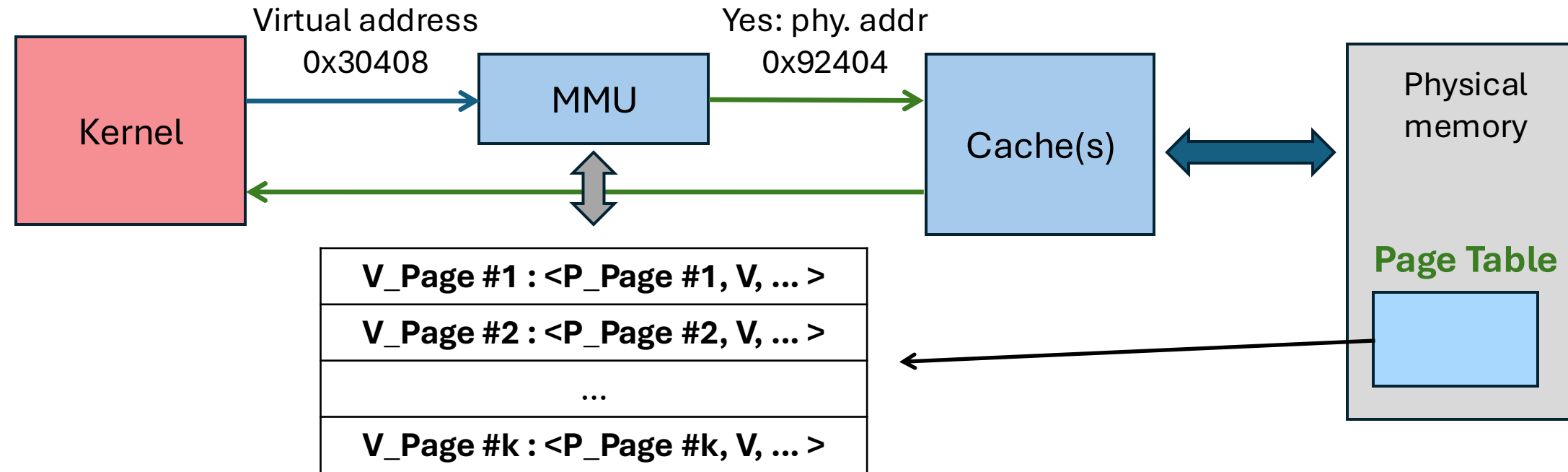
**If HitRate = 99%, what is AMAT?**

- $(0.99 \times 10) + (0.01 \times 110) = 11 \text{ ns}$

**If HitRate = 10%, what is AMAT**

- $(0.1 \times 10) + (0.9 \times 110) = 100 \text{ ns}$

# Caching Address Translation



## Cache results of recent translations !

- Different from a traditional cache
- Cache Page Table Entries using Virtual Page # as the key

# TLBs

## Translation Lookaside Buffers

- Translate **virtual page #s into PTEs** (not physical address)
- Can be done in a single machine cycle

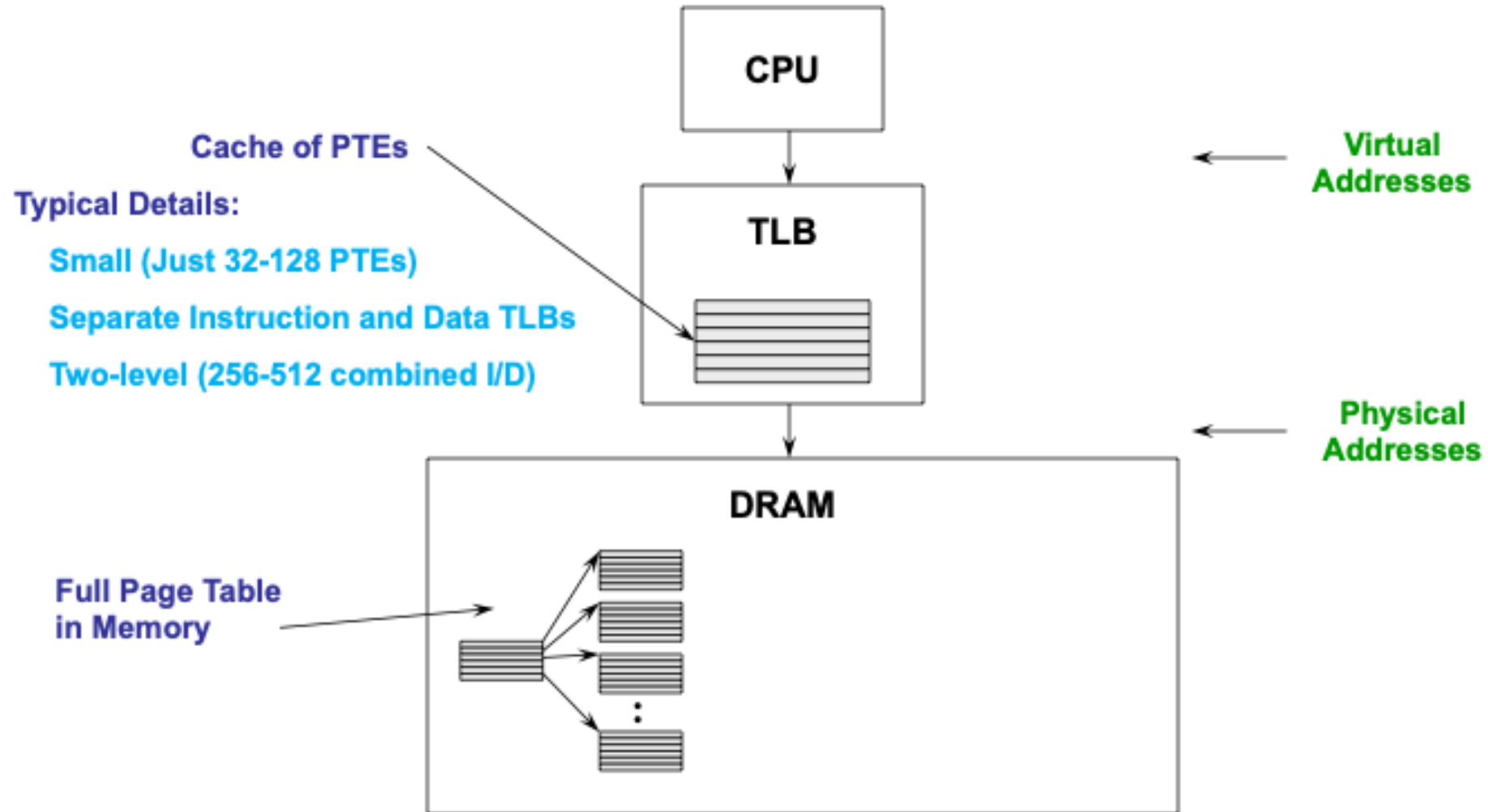
## TLBs implemented in hardware

- Typically 4-way to fully associative cache (all entries looked up in parallel)
- Cache tags are virtual page numbers
- Cache values are PTEs (entries from page tables)
- With PTE + offset, can directly calculate physical address

## TLBs exploit locality

- Processes only use a handful of pages at a time
  - 32-128 entries/pages (128-512K)
  - Only need those pages to be “mapped”
- Hit rates are therefore very important

# TLBs



# Managing TLBs

## Address translations for most instructions are handled using the TLB

- >99% of translations, but there are misses (TLB miss)...

## Who places translations into the TLB (loads the TLB)?

- **Hardware-managed TLB** (Memory Management Unit) [x86]
  - Knows where page tables are in main memory
  - OS maintains tables, HW accesses them directly
  - Tables have to be in HW-defined format (inflexible)
- **Software-managed TLB** (OS) [MIPS, Alpha, Sparc, PowerPC]
  - TLB faults to the OS, OS finds appropriate PTE, loads it in TLB
  - Must be fast (but still 20-200 cycles)
  - CPU ISA has instructions for manipulating TLB
  - Tables can be in any format convenient for OS (flexible)

# Managing TLBs (2)

## **OS ensures that TLB and page tables are consistent**

- When it changes the protection bits of a PTE, it needs to invalidate the PTE if it is in the TLB

## **Reload TLB on a process context switch**

- Invalidate all entries
- Why? What is one way to fix it?

## **When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted**

- Choosing PTE to evict is called the **TLB replacement policy**
- Implemented in hardware, often simple (e.g., Last-Not-Used)

# The Common Workflow

**Situation: Process is executing on the CPU, and it issues a read to an address**

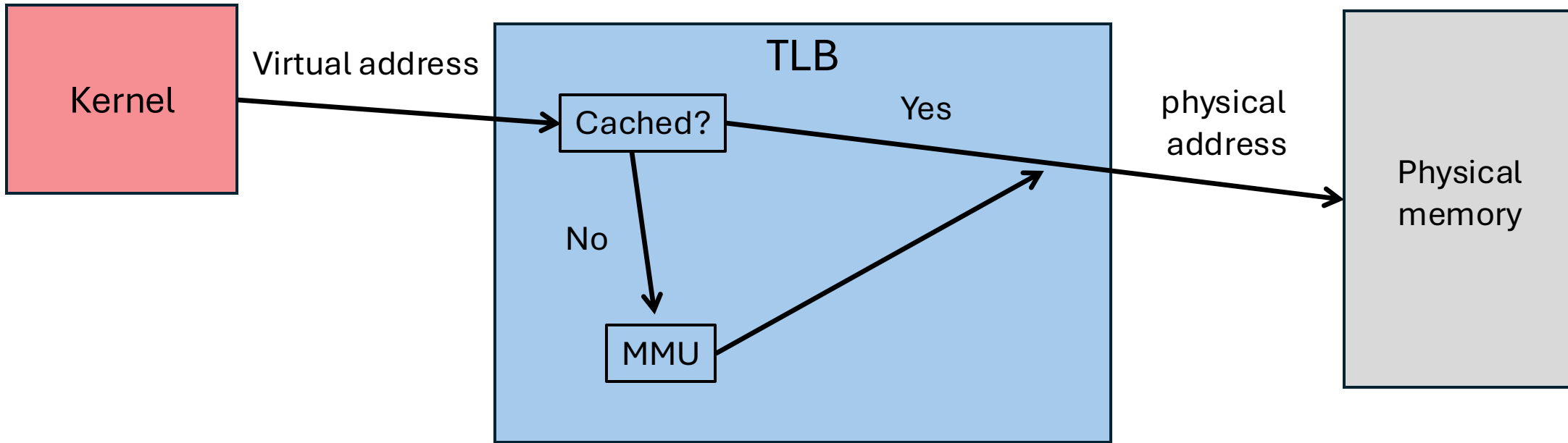
- What kind of address is it? Virtual or physical?

**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

**Note: This is all done by the hardware**

# TLB Misses



**At this point, one other things can happen**

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



# Reloading the TLB

**If the TLB does not have mapping, two possibilities:**

1. MMU loads PTE from page table in memory
  - Hardware managed TLB, OS not involved in this step
  - OS has already set up the page tables so that the hardware can access it directly
2. Trap to the OS
  - Software managed TLB, OS intervenes at this point
  - OS does lookup in page table, loads PTE into TLB
  - OS returns from exception, TLB continues

**A machine will only support one method or the other**

**At this point, there is a PTE for the address in the TLB**

# Lecture Overview

Today we'll cover more paging mechanisms:

Two-level page table

Address translation

Optimizations

- Managing page tables (space)
- Efficient translations (TLBs) (time)
- Demand paged virtual memory (space)

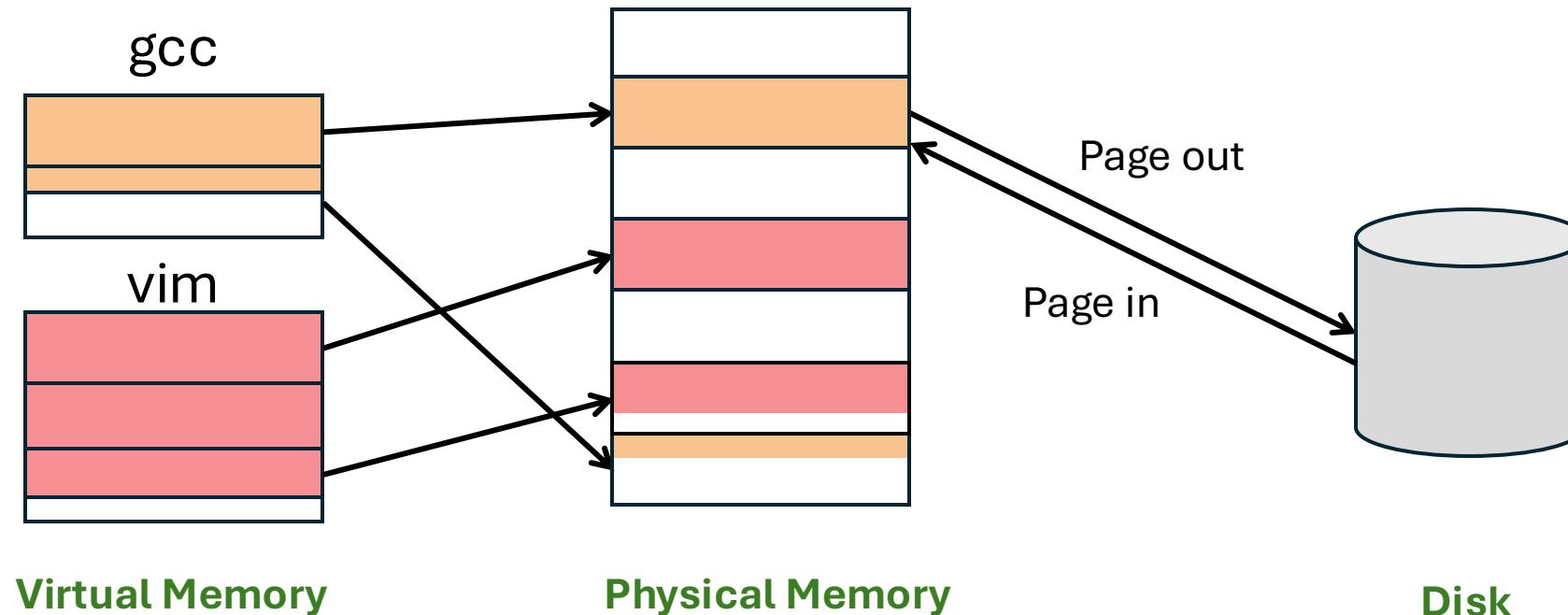
**Advanced functionality**

- Sharing memory
- Copy on Write
- Mapped files

# Demanded Paging

## Pages can be moved between memory and disk

- Use disk to simulate larger virtual than physical memory
- This process is called paging in/out



# **Demanded Paging**

**Pages can be moved between memory and disk**

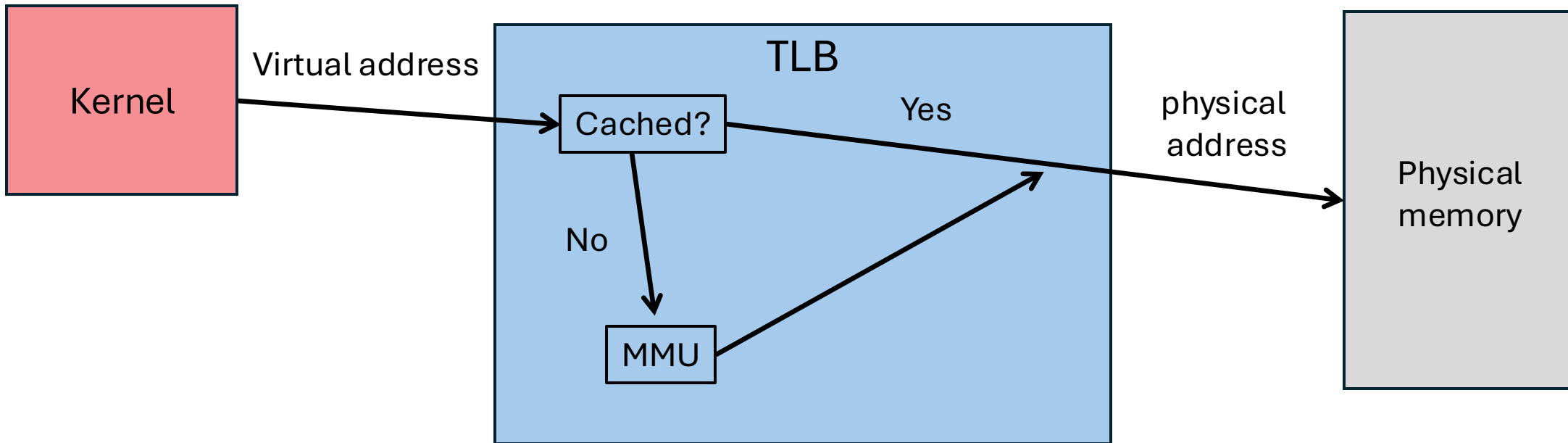
## **Paging process over time**

- 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 (**where? 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
- Main memory becomes a cache for disk

# Recap: TLB Misses



**At this point, one other things can happen**

1. TLB does not have a PTE mapping this virtual address
2. PTE in TLB, but memory access violates PTE protection bits

# TLB Misses (2)

**Page table lookup (by HW or OS) can cause a recursive fault if page table is paged out**

- Assuming page tables are in OS virtual address space
- Not a problem if tables are in physical memory
- Yes, this is a complicated situation

**When TLB has PTE, it restarts translation**

- Common case is that the PTE refers to a valid page in memory
  - These faults are handled quickly, just read PTE from the page table in memory and load into TLB
- Uncommon case is that TLB faults again on PTE because of PTE protection bits (e.g., page is invalid)
  - Becomes a page fault...

# Page Faults

## **PTE can indicate a protection fault**

- Read/write/execute – operation not permitted on page
- Invalid – virtual page not allocated, or page not in physical memory

## **TLB traps to the OS (software takes over)**

- R/W/E – OS usually will send fault back up to process, or might be playing games (e.g., copy on write, mapped files)
- Invalid
  - Virtual page not allocated in address space
    - OS sends fault to process (e.g., segmentation fault)
  - Page not in physical memory
    - OS allocates frame, reads from disk, maps PTE to physical frame

# Page Faults (2)

## What happens when a process accesses a page is evicted?

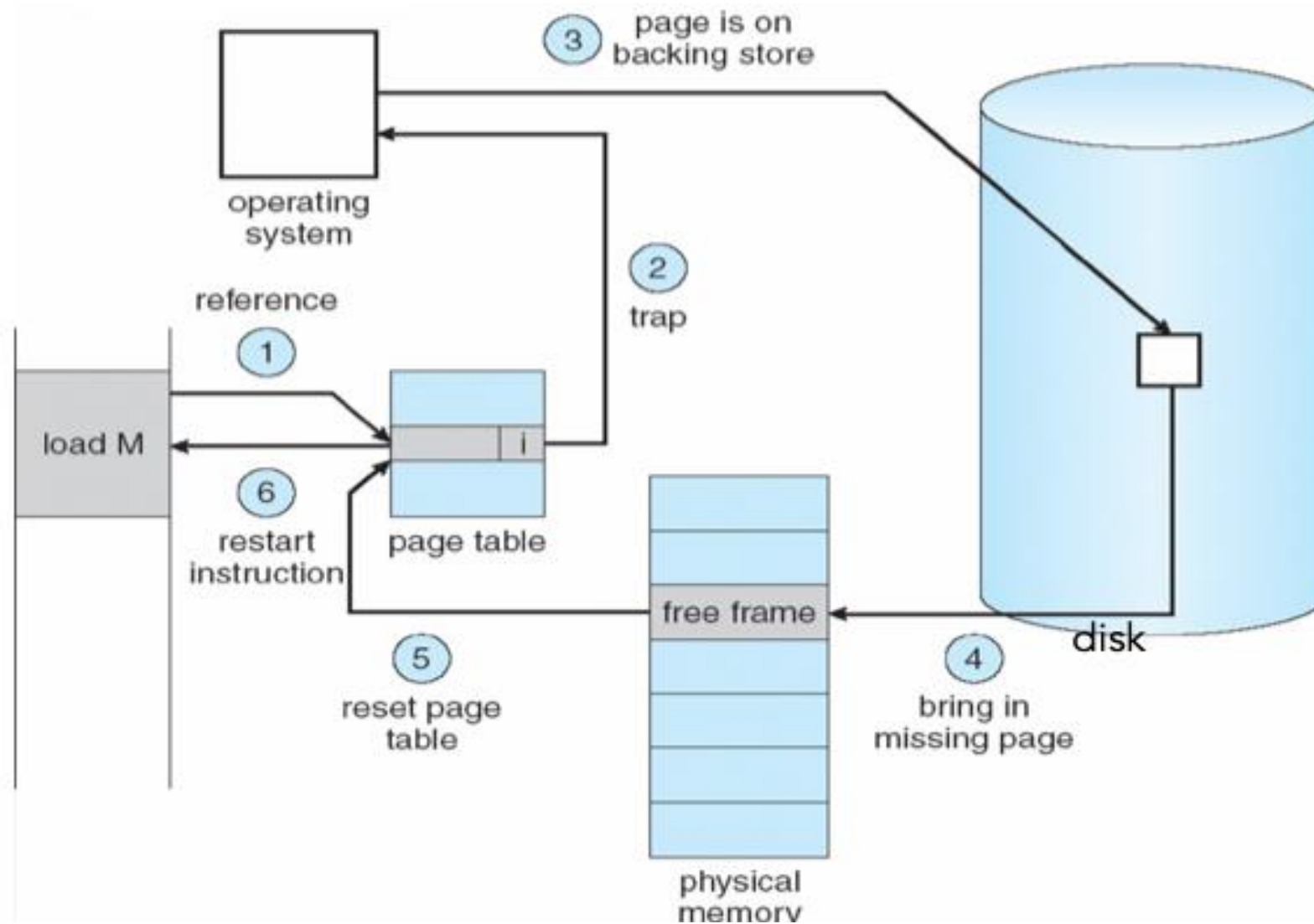
1. When the OS evicts a page, it sets the PTE as invalid and stores the location of the page in the swap file in the PTE
2. When a process accesses the page, the invalid PTE causes a trap (page fault)
3. The trap will run the OS page fault handler
4. Handler uses the invalid PTE to locate page in swap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

## But where does it put it? Have to evict something else

- OS usually keeps a pool of free pages around so that allocations do not always cause evictions



# Page Faults & Paging



# Address Translation Summary

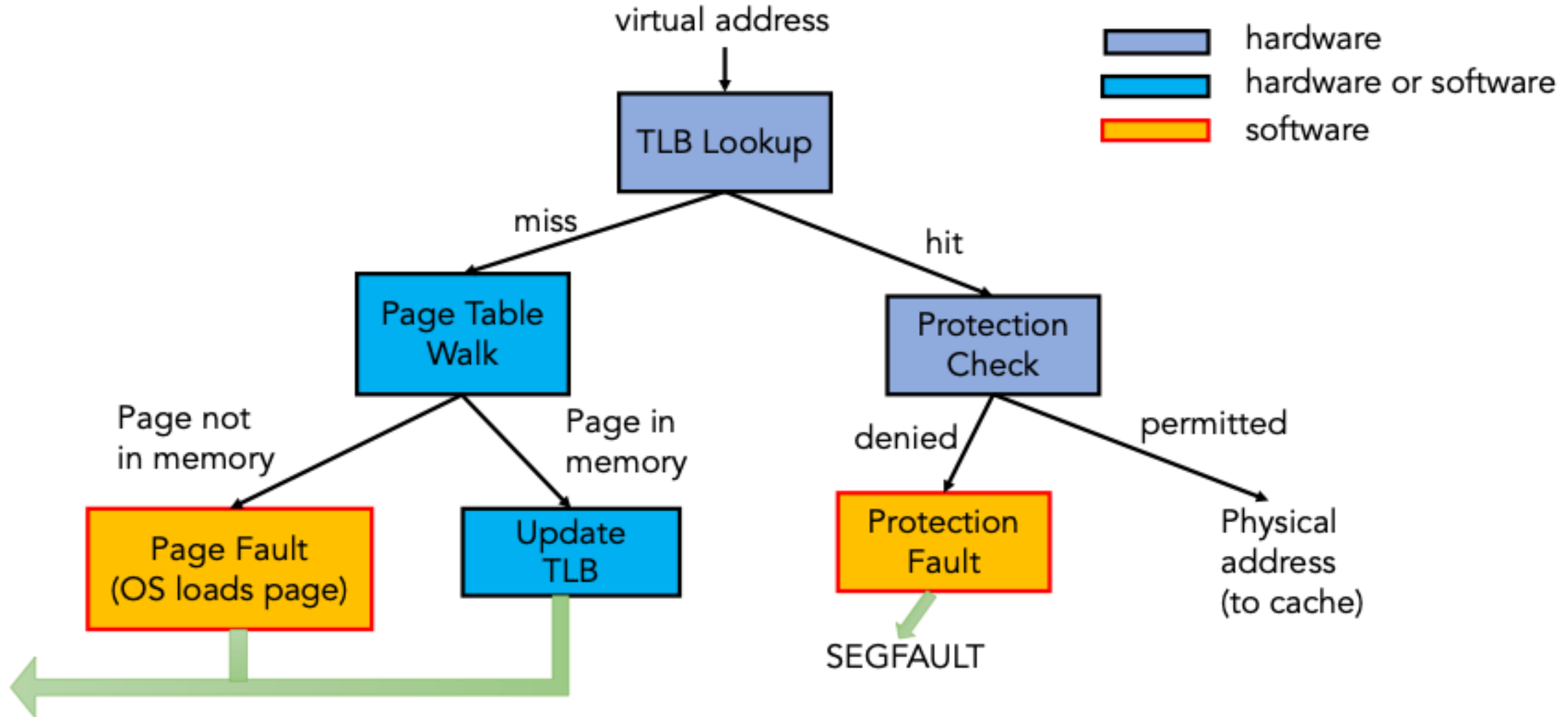
**We started this topic with the high-level problem of translating virtual addresses into physical addresses**

**We've covered all of the pieces**

- Virtual and physical addresses
- Virtual pages and physical page frames
- Page tables and page table entries (PTEs), protection
- TLBs
- Demand paging

**Now let's put it together, bottom to top**

# Address Translation: Putting It All Together



# Lecture Overview

Today we'll cover more paging mechanisms:

Two-level page table

Address translation

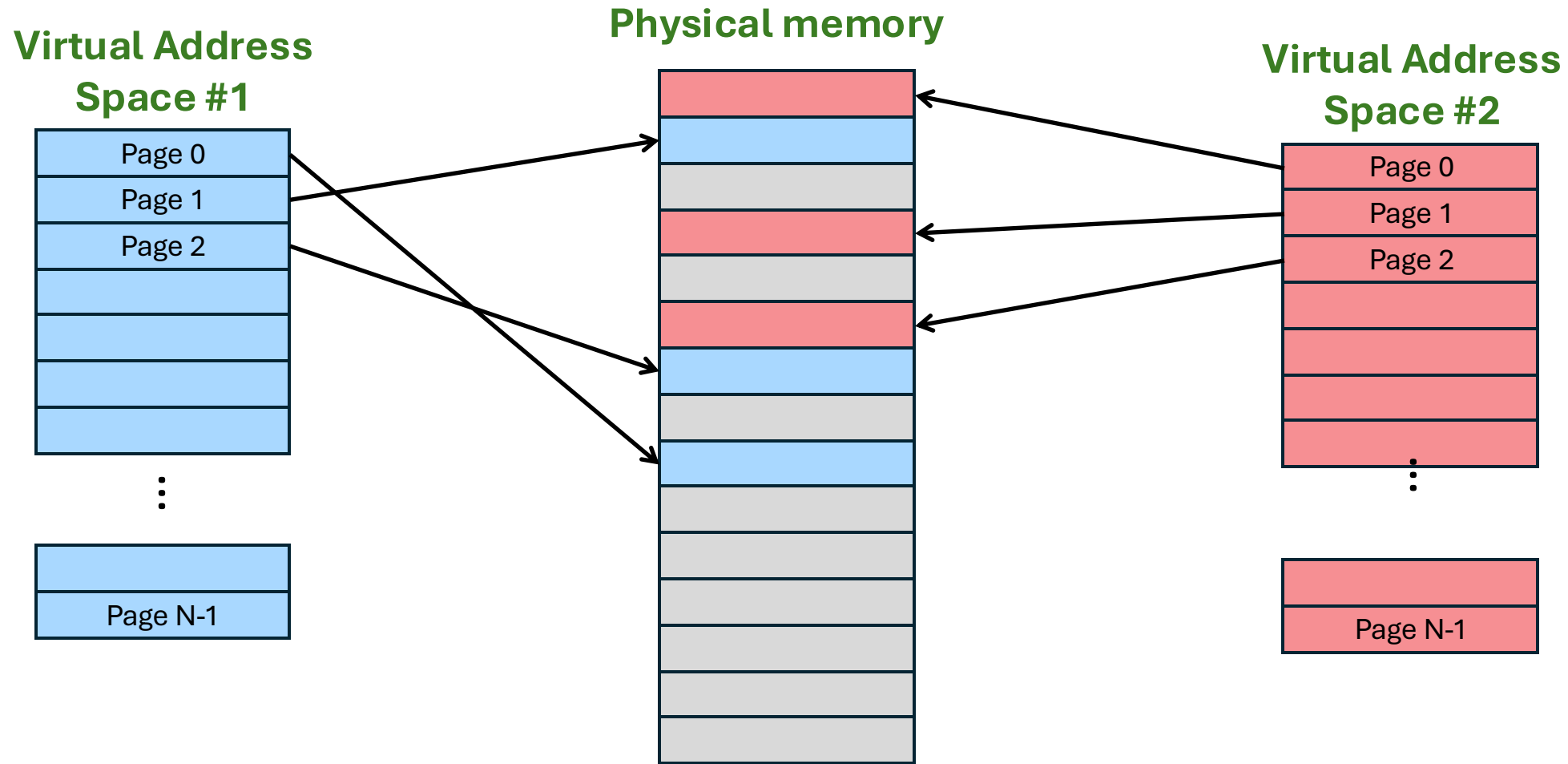
Optimizations

- Managing page tables (space)
- Efficient translations (TLBs) (time)
- Demand paged virtual memory (space)

**Advanced functionality**

- Sharing memory
- Copy on Write
- Mapped files

# Isolation: No Sharing



# Sharing

**Private virtual address spaces protect applications from each other**

- Usually exactly what we want

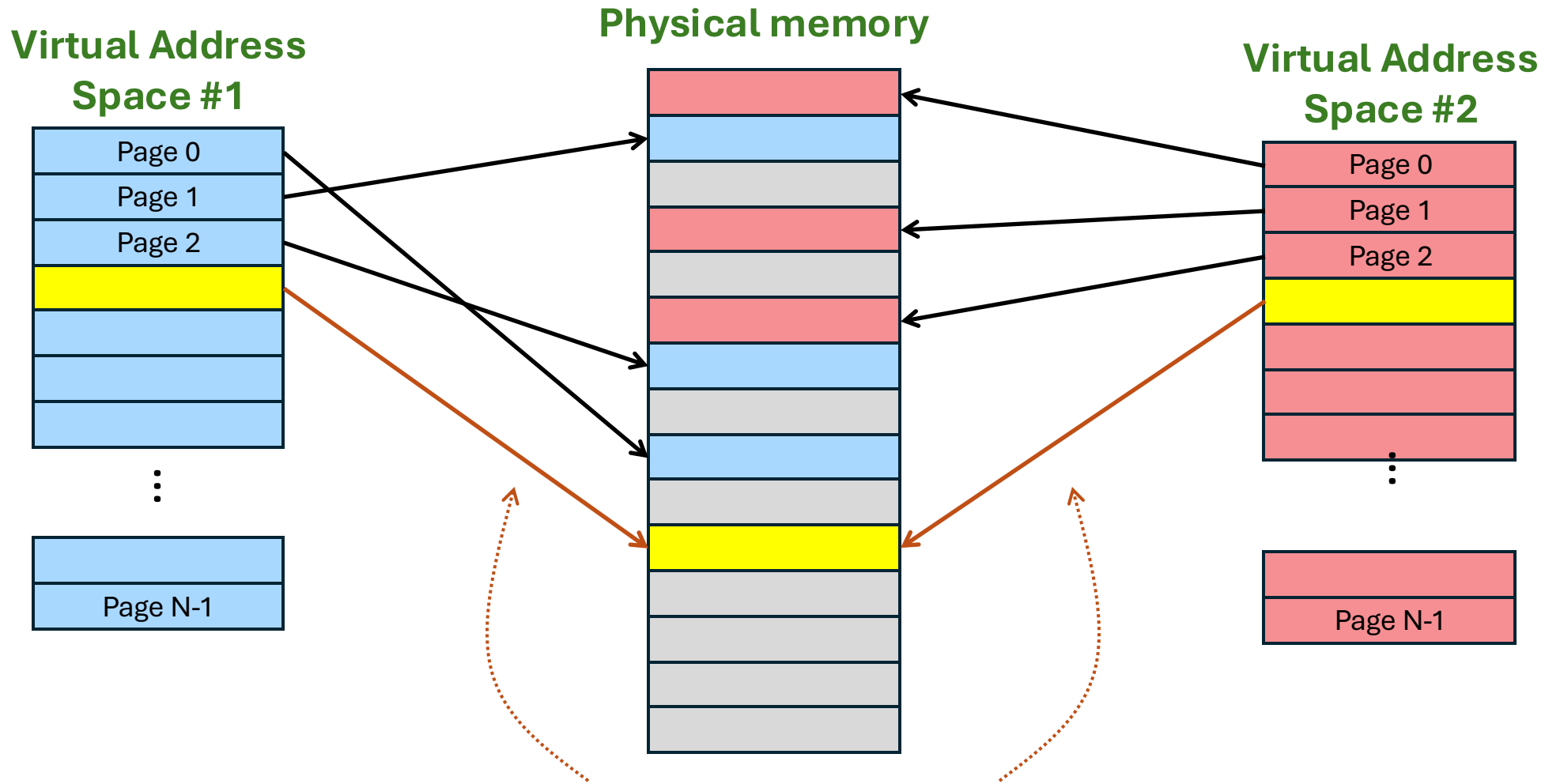
**But this makes it difficult to share data (have to copy)**

- Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying

**We can use **shared memory** to allow processes to share data using direct memory references**

- Both processes see updates to the shared memory segment
  - Process B can immediately read an update by process A
- How are we going to coordinate access to shared data?

# Sharing Pages



**PTEs Point to Same Physical Page**

# Sharing(2)

## How can we implement sharing using page tables?

- Have PTEs in both tables map to the same physical frame
- Each PTE can have different protection values
- Must update both PTEs when page becomes invalid

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

- Different: Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid (Why?)
- Same: Less flexible, but shared pointers are valid (Why?)

## What happens if a pointer inside the shared segment references an address outside the segment?



# Copy on Write

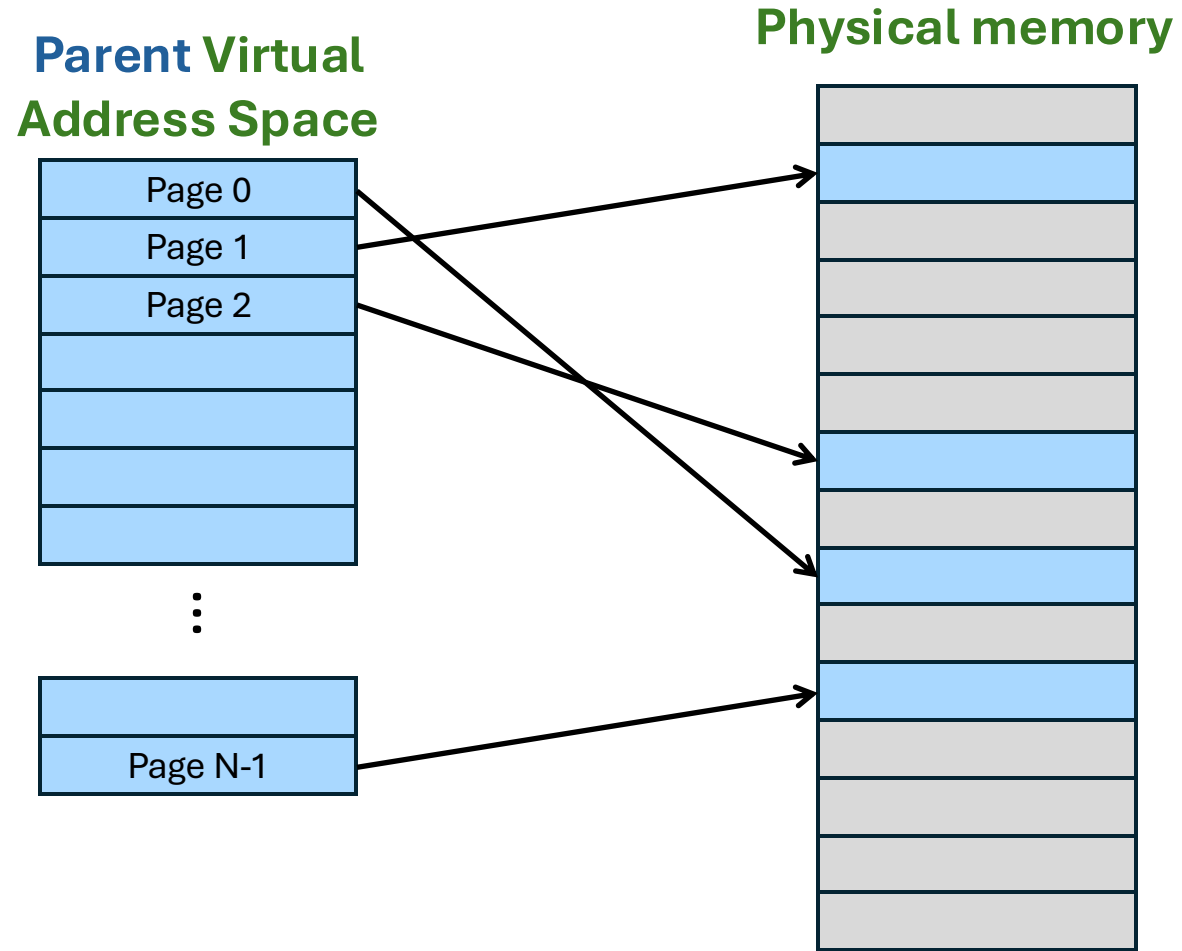
## OSes spend a lot of time copying data

- System call arguments between user/kernel space
- Entire address spaces to implement fork()

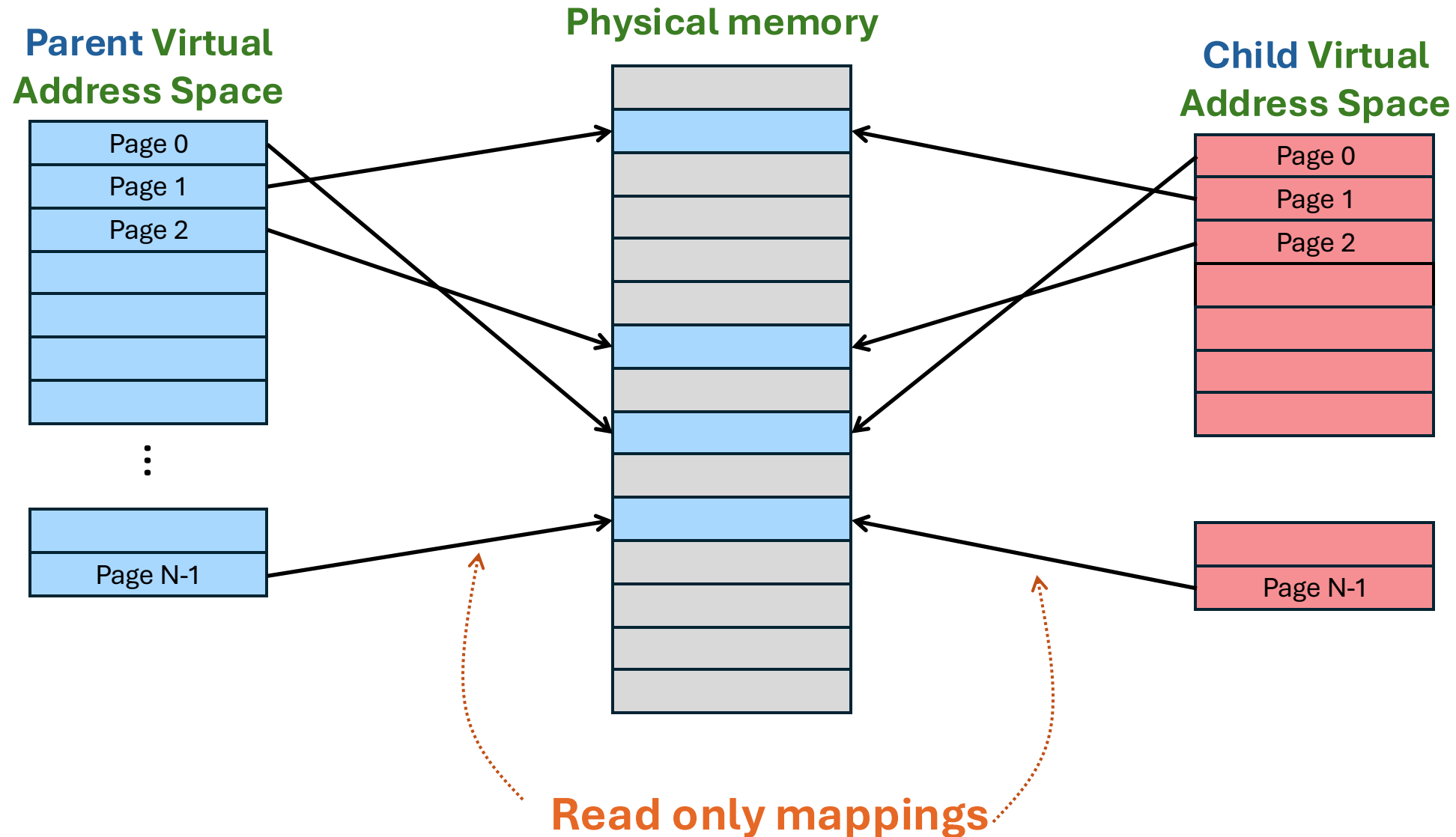
## Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether

- Instead of copying pages, create **shared mappings** of parent pages in child virtual address space
- Shared pages are protected as read-only in parent and child
  - Reads happen as usual
  - Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
- How does this help fork()?

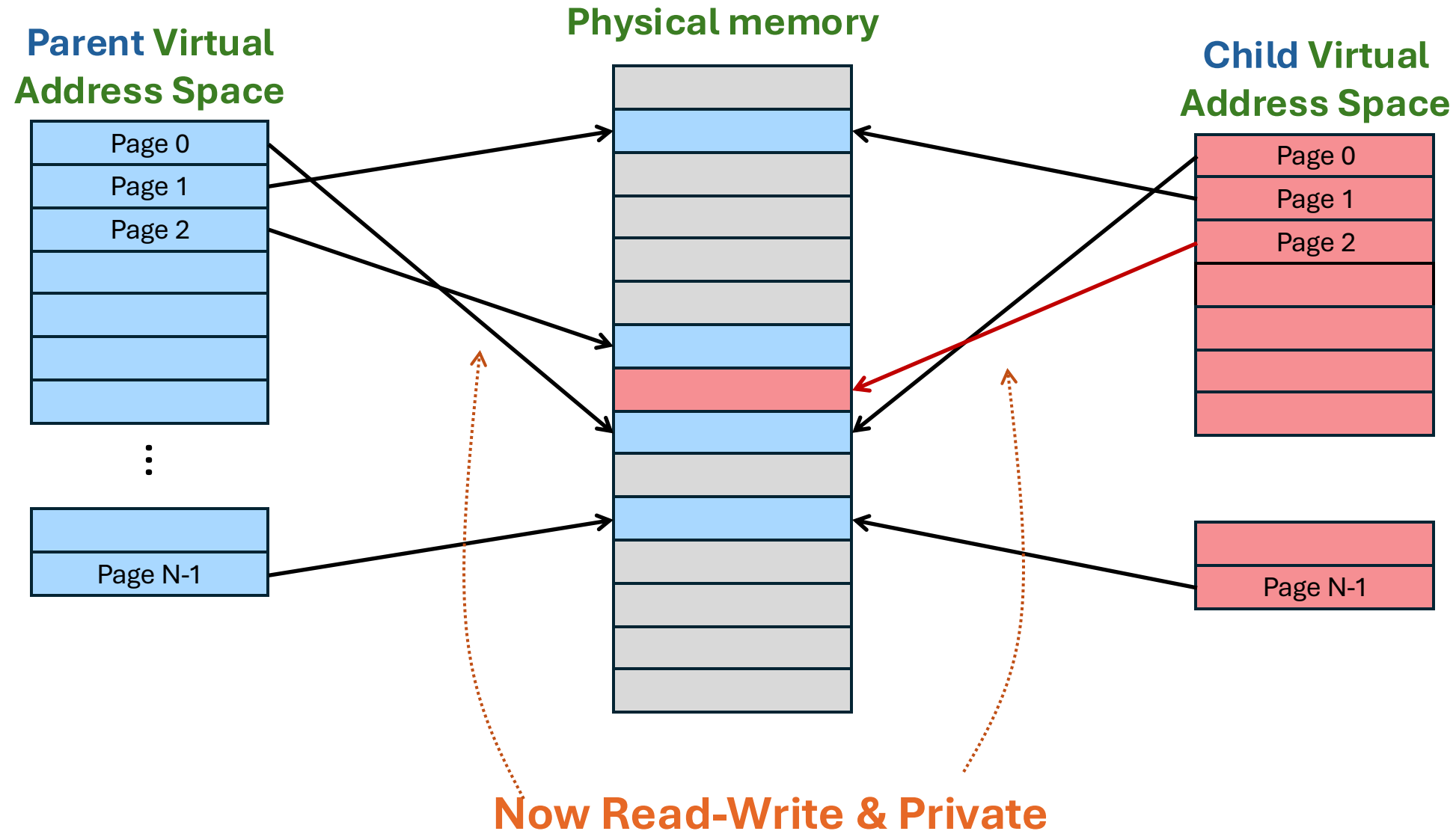
# Copy on Write: Before Fork



# Copy on Write: After Fork



# Copy on Write: On A Write



# 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 (mmap() in Unix)**

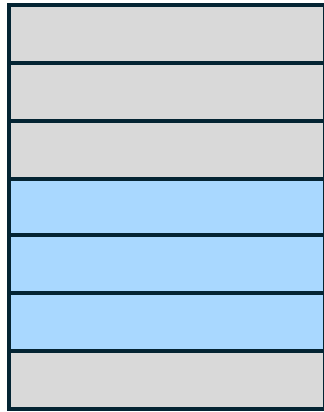
- PTEs map virtual addresses to physical frames holding file data
- Virtual address  $\text{base} + N$  refers to offset  $N$  in file

**Initially, all pages mapped to file are invalid**

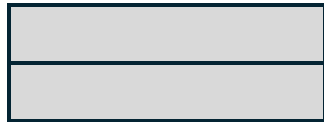
- OS reads a page from file when invalid page is accessed
- OS writes a page to file when evicted, or region unmapped
- If page is not dirty (has not been written to), no write needed
  - Another use of the dirty bit in PTE

# Mapped Files

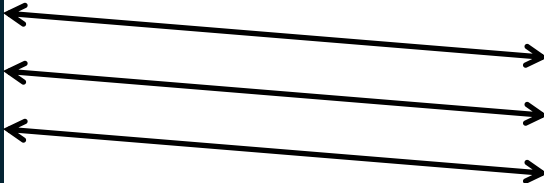
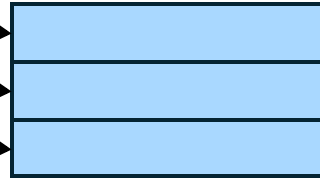
Virtual Address  
Space



⋮



Mapped File



# Mapped Files (2)

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

- Virtual address space not backed by “real” files also called [Anonymous VM](#)

## Advantages

- Uniform access for files and memory (just use pointers)
- Less copying (why?)

## Drawbacks

- Process has less control over data movement
  - OS handles faults transparently
- Does not generalize to streamed I/O (pipes, sockets, etc.)

# Summary

## Paging mechanisms:

## Optimizations

- Managing page tables (space)
- Efficient translations (TLBs) (time)
- Demand paged virtual memory (space)

## Recap address translation

## Advanced Functionality

- Sharing memory
- Copy on Write
- Mapped files

## Next time: Paging policies



# Next time...

**Chapters 21-23**