Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 13: Segmentation & Paging

Prof. Ryan Huang

Administration

Midterm this Wednesday

In-person, closed-books, on-paper

- Cannot use notes, slides, books
- No electronic devices
- Use no. 2 pencils, black or blue pens in your exam

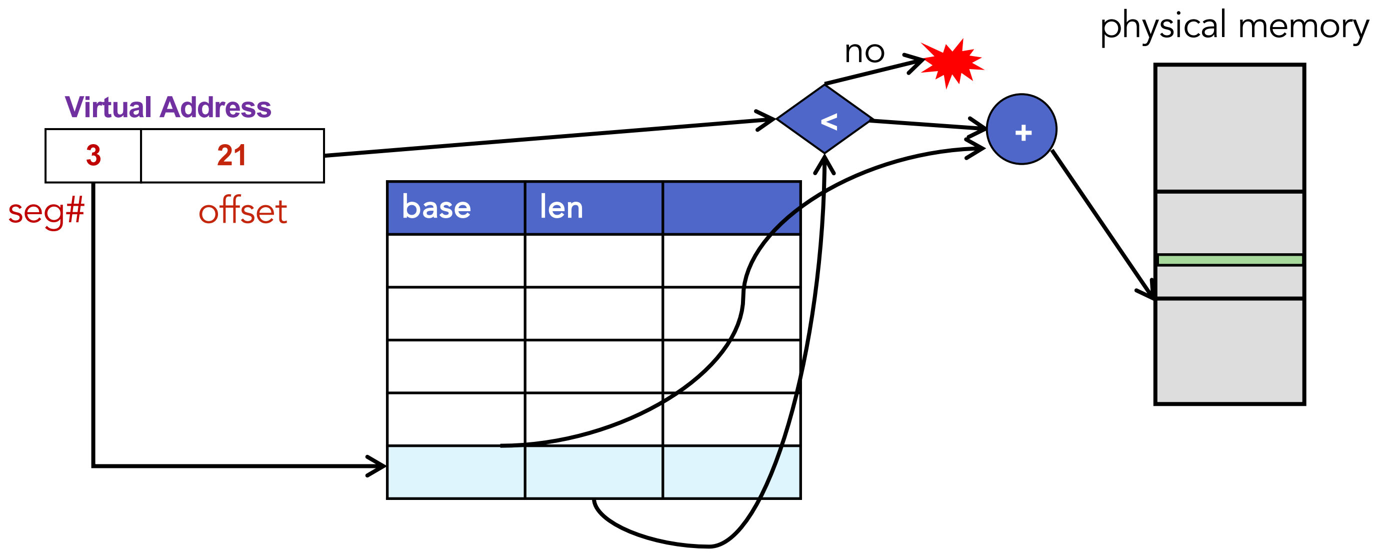
Room assignment released on Piazza, make sure you go to the right room!

- The exams will be pre-placed, so take the right seat!

See pinned Piazza post on how to prepare for the exam

Segmentation

Segment #	Base	Bound	Protection	
0	0x4000	0x700	r/o	code segment
1	0	0x500	r/w	data segment
2	n/a	0		unused
3	0x2000	0x1000	r/w	stack segment



Segmentation: translation algorithm

```
(base, bound, protection) = segment_info[segment]

if (segment is invalid or protected, or offset >= bound) {

      trap to kernel

} else {

      physical address = offset + base

}
```

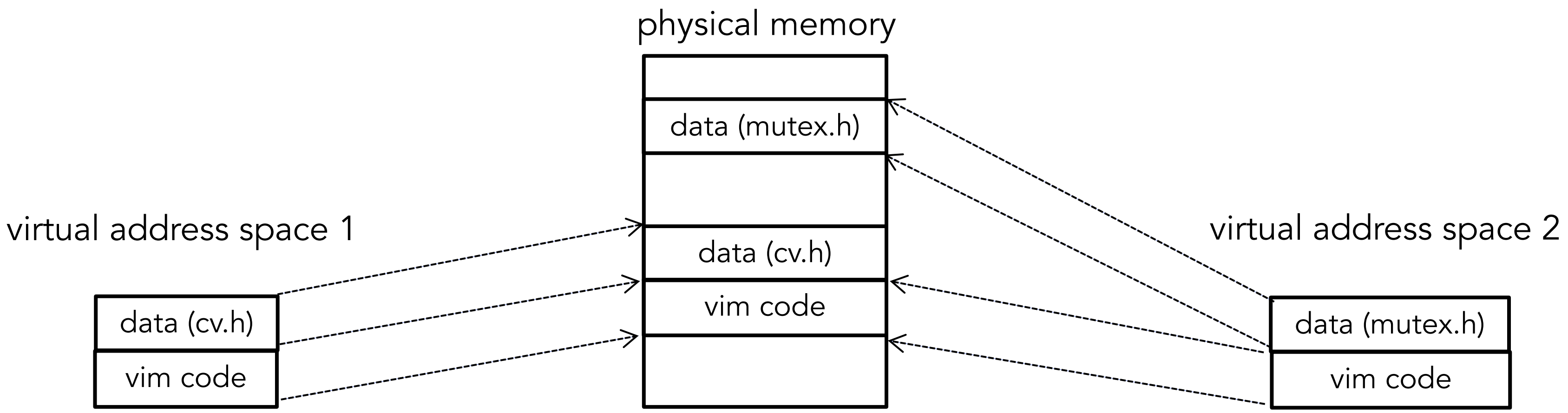
What must be changed on a context switch?

Segmentation Trade-offs

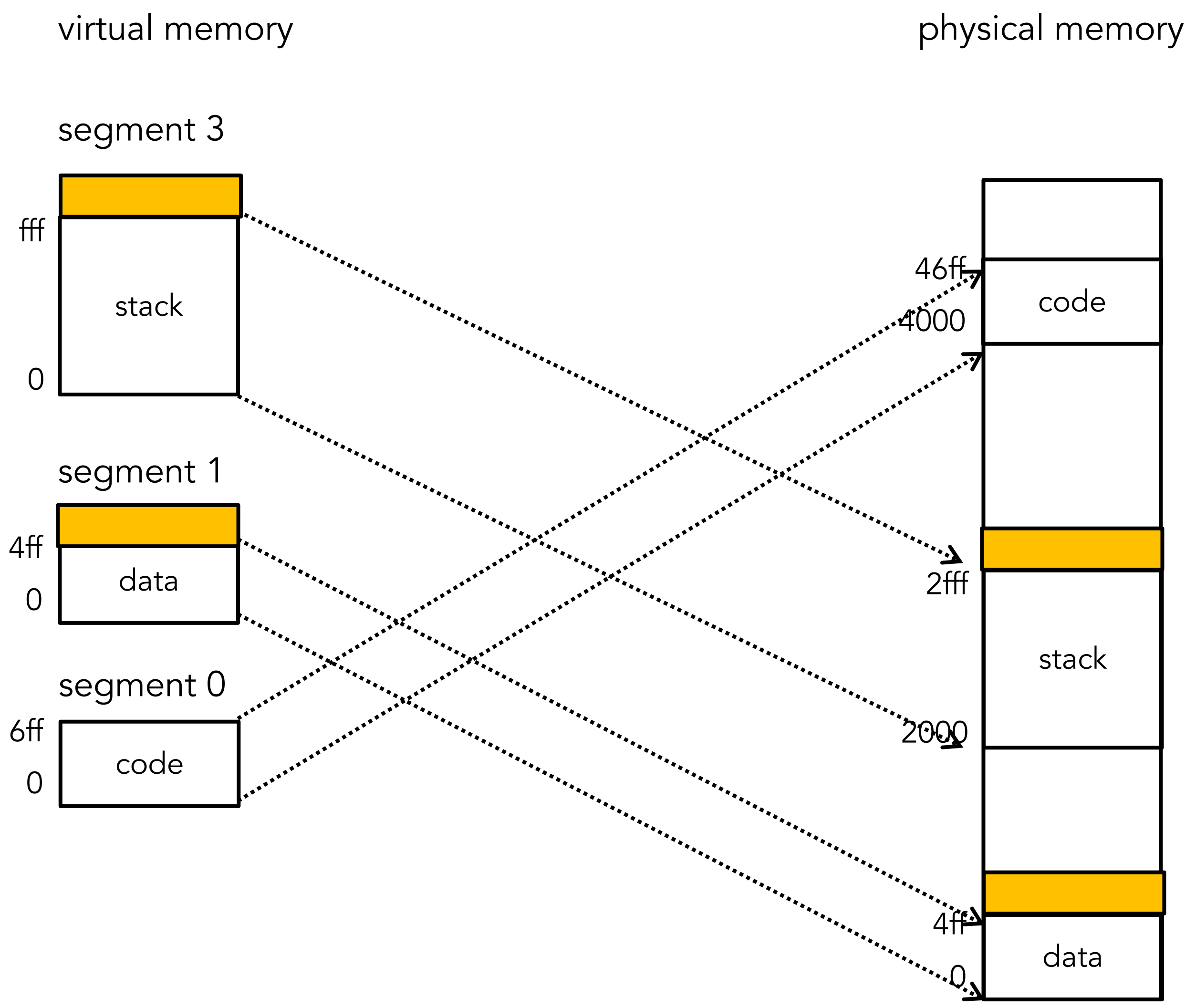
Advantages

- Don’t need entire process in memory
  - Needed segments in memory, others swapped to disk
- Can easily share memory among processes (how?)
- Allows multiple growing regions

Segmentation supports partial sharing



Segmentation allows multiple growing regions



Segmentation Trade-offs

Advantages

- Don't need entire process in memory
- Can easily share memory among processes
- Allows multiple growing regions

Disadvantages

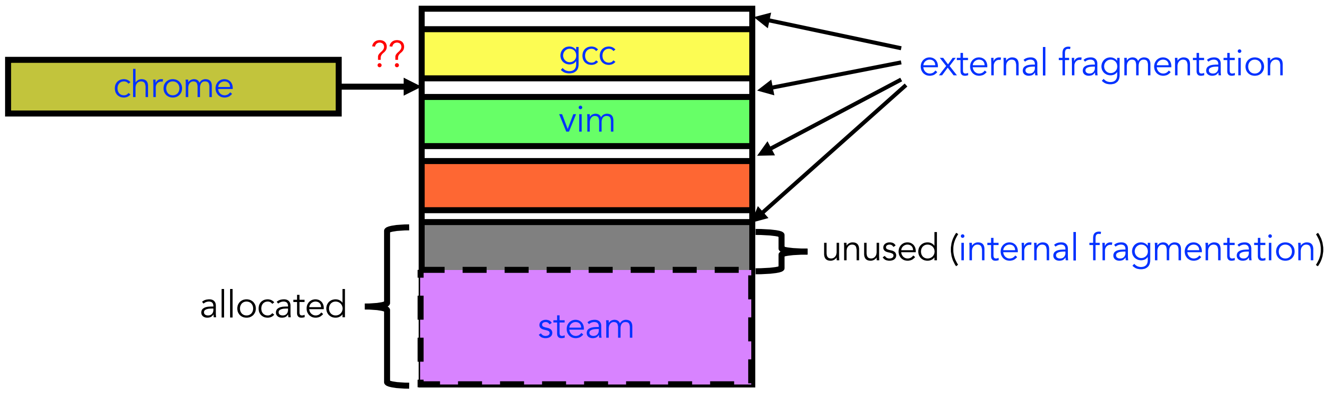
- Translation is more costly (than base+bound)
- Does not support large address space?
  - Each segment still requires *N* contiguous bytes of physical mem
    - but possible for sum of all segments > physical mem size
- Makes *fragmentation* a real problem.

Fragmentation

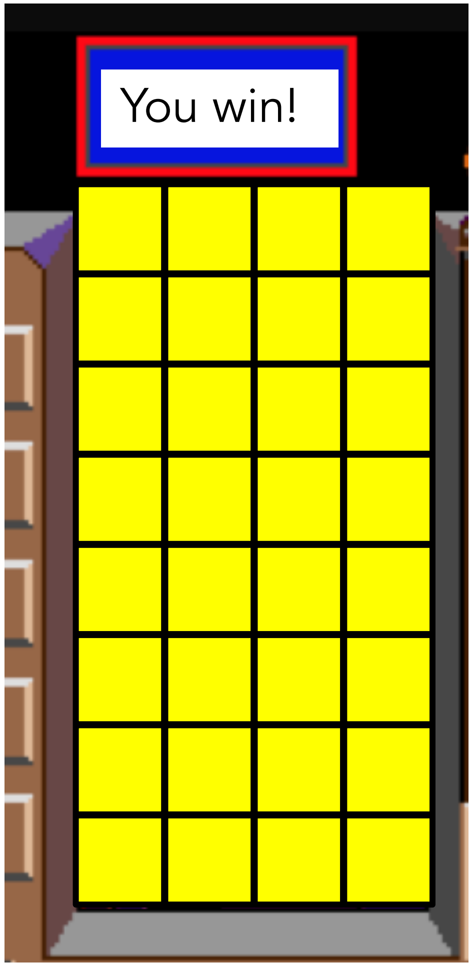
Fragmentation ⇒ Inability to use free memory

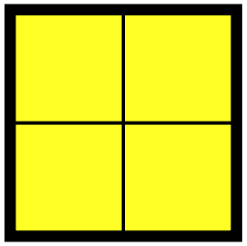
Over time:

- many small holes (external fragmentation)
- no external holes, but force internal waste (internal fragmentation)



Tetris analogy





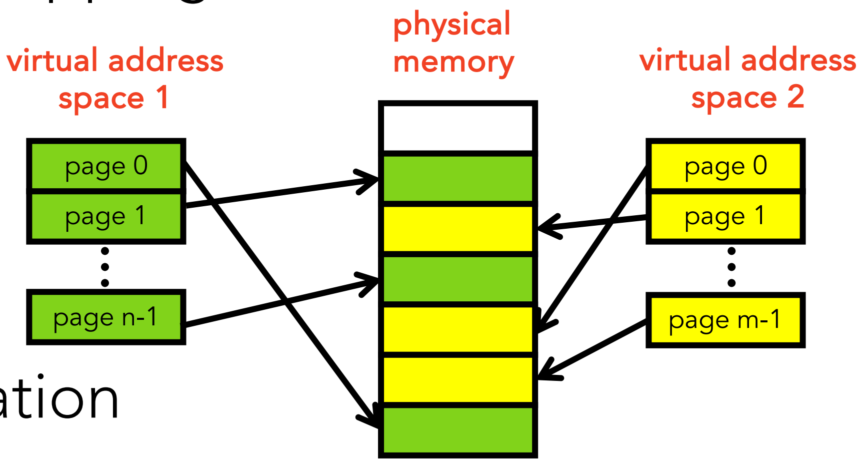
Idea 3: Paging

Divide memory up into fixed-size units (*pages*)

- For both address space and physical memory

Map virtual pages to physical pages

- Each process has separate mappings



Advantages?

- Eliminate external fragmentation
- Simplify allocation
  - any physical page can store any virtual page

Paging data structure

Virtual page #

Physical page #

~~Seg~~ #	Base / page size	~~Bound~~	Prot.
0	0x4~~000~~	~~0x700~~	r/o
1	0	~~0x500~~	r/w
3	n/a	~~0~~	
4	0x2~~000~~	~~0x1000~~	r/w

~~Segment~~ table

Page

Paging data structure

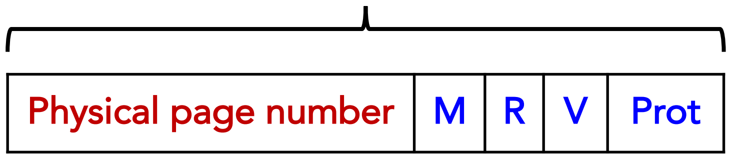
VPN	PPN	Protc.	Valid
0	4	r/o	1
1	0	r/w	1
2	n/a		0
3	2	r/w	1
4	1	r/w	1
…			

Page table

Page tables

- Virtual page number (VPN) → physical page number (PPN)
  - PPN also called page frame number
- One page table entry (PTE) per page in address space

Page Table Entries (PTEs)



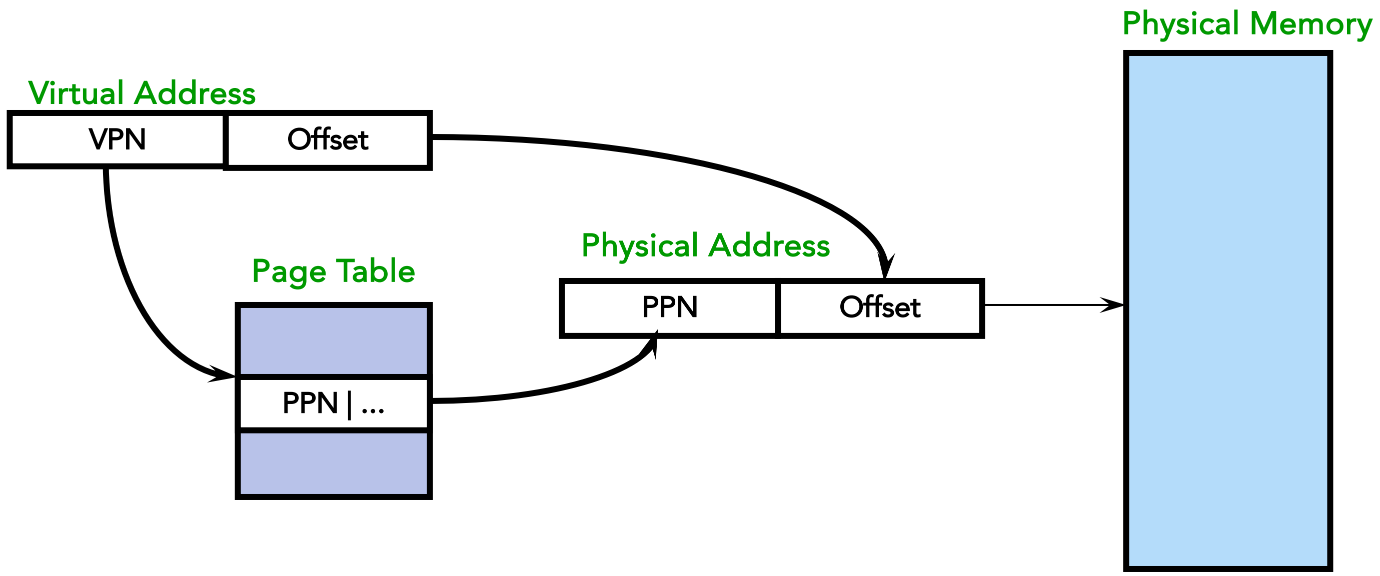
Page table entries control mapping

- Physical page number (PPN) determines physical page
- Various other bits
  - Modify: whether or not the page has been written
  - Reference: whether the page has been accessed
  - Valid: whether or not the PTE can be used
  - Protection: what operations are allowed on page
- These bits are set by the CPU or the OS

Why is Virtual Page Number not in the PTE?

Paging: translation

Virtual address: (virtual page number, offset)

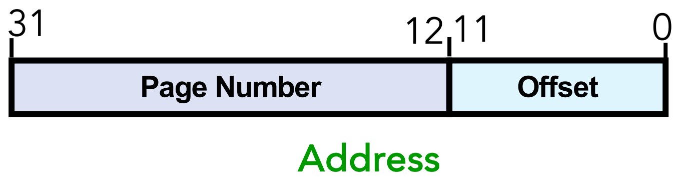


```
if (virtual page is invalid) {
        trap to OS fault handler
} else {
        physical page # = pageTable[virtual page #].physPageNum
        physical address = {physical page #}{offset}
}
```

Paging example

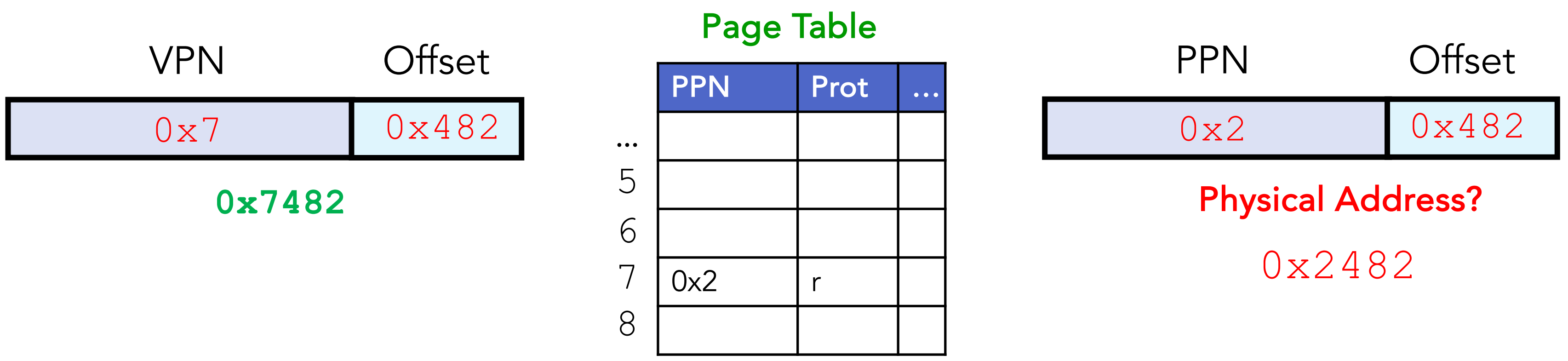
32-bit machines, pages are 4KB-sized

- Least significant 12 ($log_2 4k$) bits of address are page offset
- Most significant 20 (32 – 12) bits are page number



What is the maximum number of pages?

Virtual address is `0x7482`



	PPN	Prot	...
...			
5			
6			
7	0x2	r	
8			

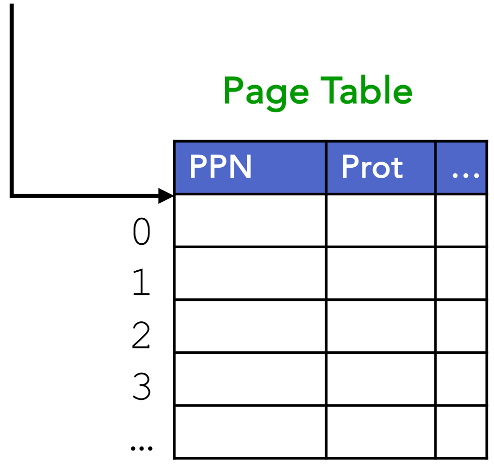
Page table switching

Page tables are stored in memory

- Different processes have different page tables

When context switching to another process, we must switch the page tables

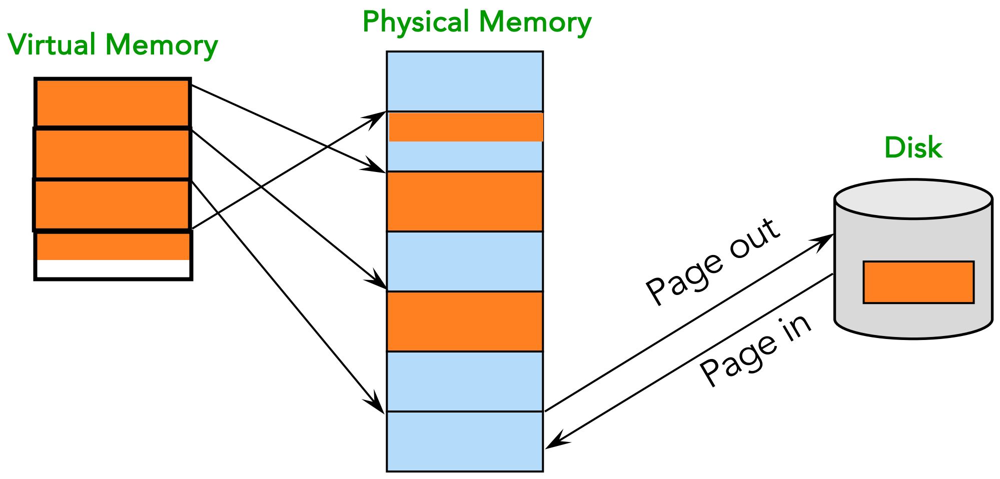
- How?
- Change the page table base register
  - On x86, register **`%cr3`**



Page swapping

Pages can be moved between memory and disk

- Each page can be resident (in memory) or non-resident (on disk)
- This process is called swapping or paging in/out
- Enables address spaces larger than physical memory



What happens if a process accesses a non-resident page?

Page faults

When a process accesses a non-resident page, it causes a trap (page fault)

- The CPU invokes the OS page fault handler
- Is the access an error?
  - No!
  - The fault occurs because OS swapped the page to disk previously → OS needs to fix this problem now
- Page fault handler needs to
  1. Read the page from disk
  2. Find a physical page (frame) to store the page content
  3. Update the Page Table Entry (PTE) to point to the physical page
- CPU retries the access

Valid vs. resident page

Valid → virtual page is legal for process to access

Access to invalid page is an error

Resident → virtual page is in physical memory

Access to non-resident page is not an error

- It occurs because OS previously swapped the page to disk
  → OS needs to fix this problem now

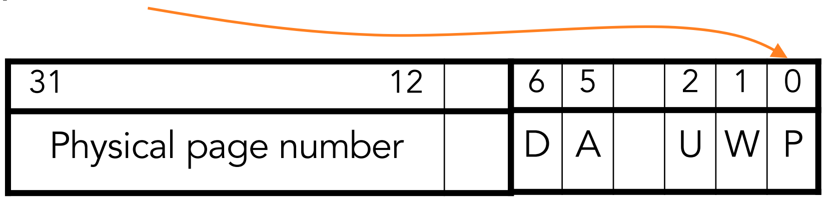
Page faults can occur due to

1. invalid accesses → terminate process
2. page not in physical memory → fix and retry access

More details on page faults

PTE contains bits to indicate the status of a page

- e.g., Present bit on x86 PTE



When OS swaps a page to disk

- It must set the non-resident bit (e.g., P=0) for that page
- It also should record the page location on the disk
- Where to record that location?
  - In PTE
  - May not be possible due to format imposed by CPU, e.g., x86
    - OS can define its own data structures, e.g., *supplemental page table*
      - use this table in the page fault handler

Page sizes

Commonly 4 KB, an empirical choice

- Can choose larger sizes, e.g., 8 KB, 1 MB, 4 MB
- Typically not smaller

Cons of using smaller page size

- More PTE entries $\rightarrow$ more memory to store page tables
- Likely more page faults

Cons of using larger page size

- Internal fragmentation

Paging trade-offs

Pros

- Easy to allocate memory
- Easy to swap out chunks of a process
- Eliminates external fragmentation
- Flexible sharing among processes
    - How?
        - Have PTEs in both tables map to the same physical frame
        - Each PTE can have different protection values

Paging trade-offs

Pros

Cons

- Can still have *internal fragmentation*
- Memory required to hold page table is significant
  - 32 bit address space w/ 4KB pages = 2^{20} PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!

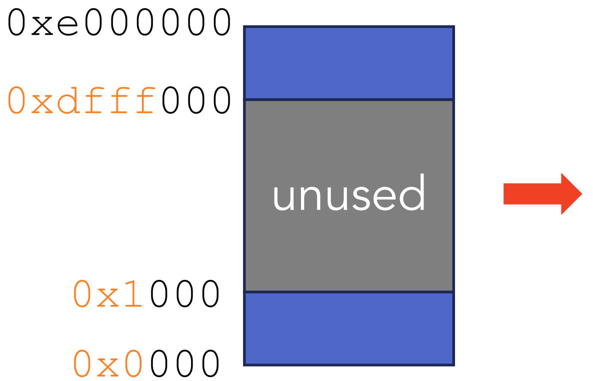
How to reduce page table memory overhead?

Multi-level page table

Observation: only need to map the portion of the address space actually being used

How do we only map what is being used?

- Dynamically extend page table?
- Does not work if address space is sparse



Need to allocate a page table with size of at least 57,344 PTEs

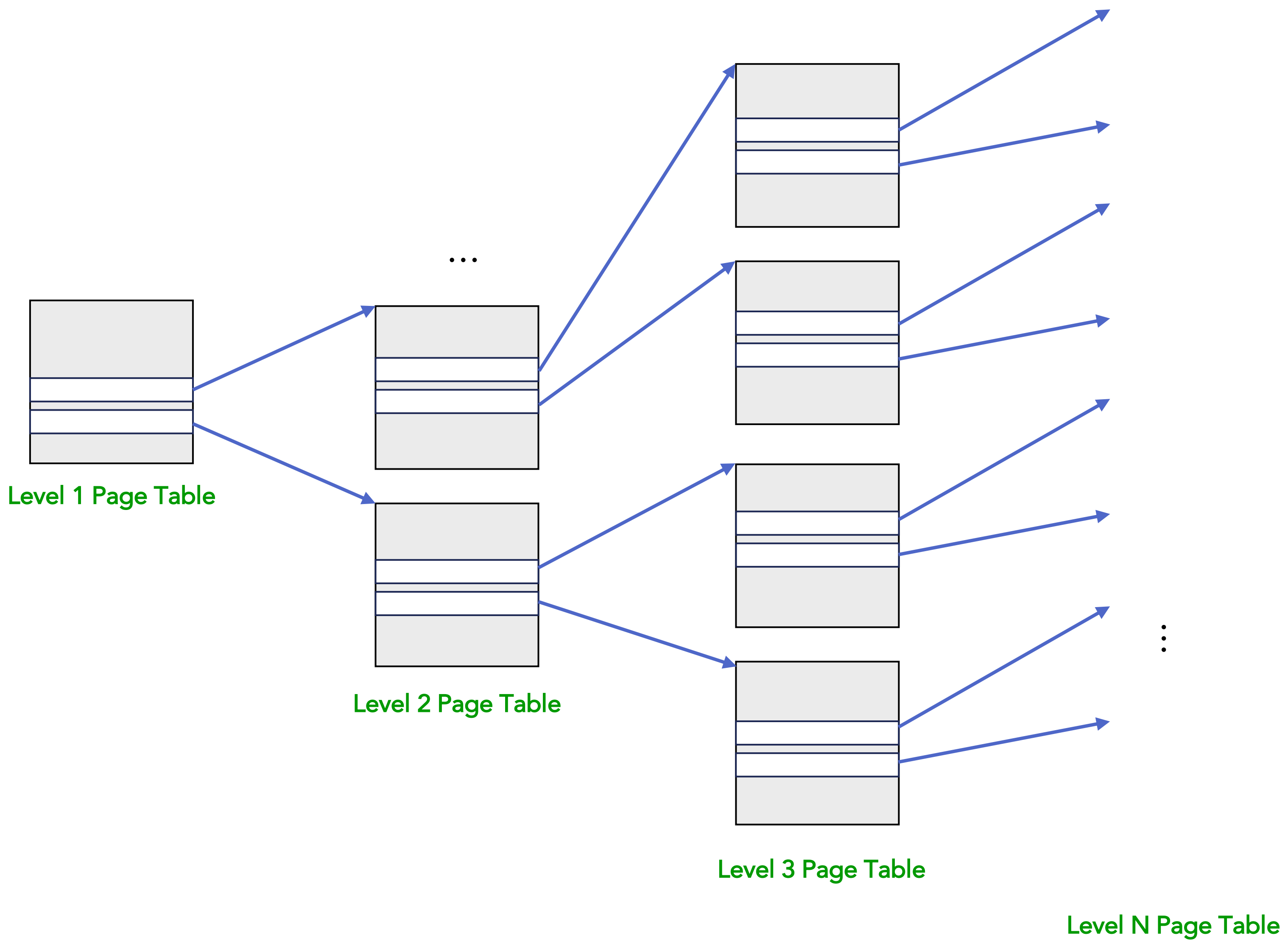
- VPN ∈ [0, 0xdfff]

Only 2 PTEs are used

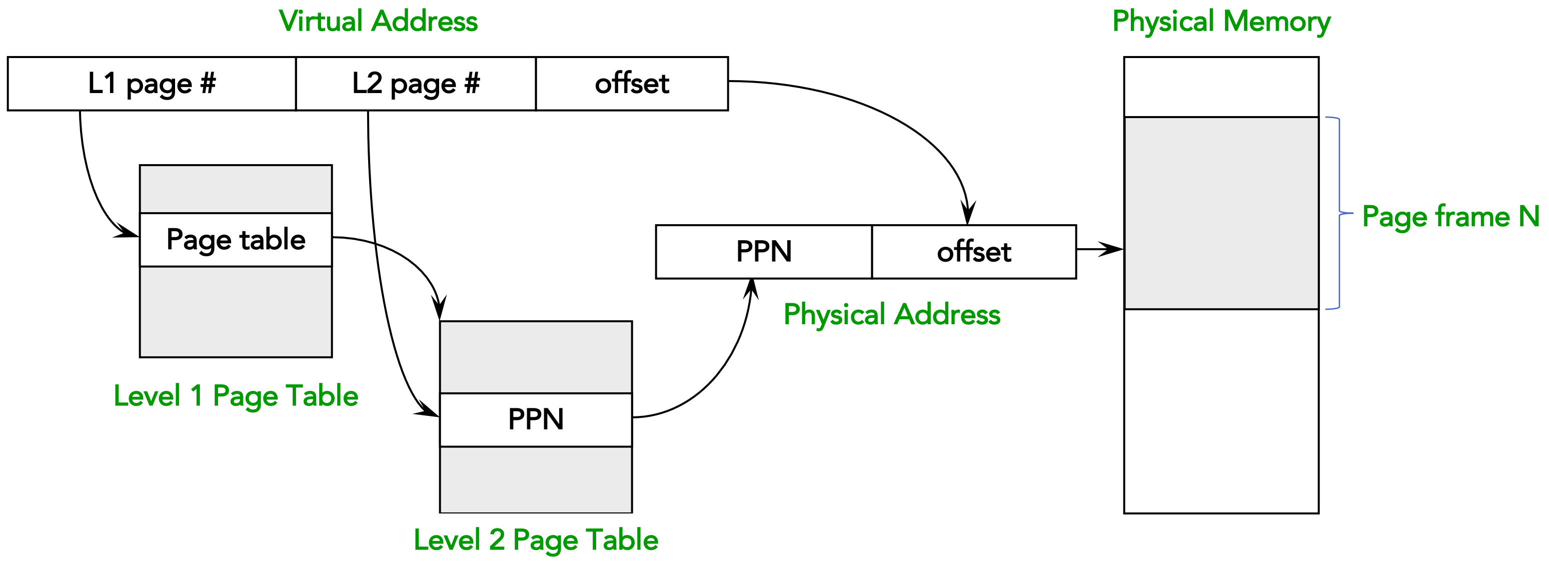
Use a level of indirection: multi-level page tables

- Turn page table from a simple array to a "tree"

Multi-level page table



Two-level page tables



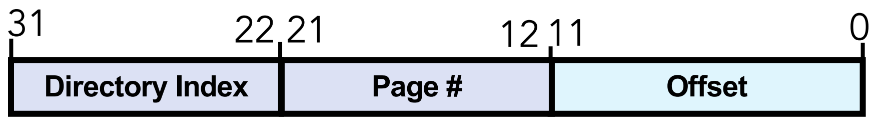
Virtual addresses has three components:

1. Master page number
   - Indexes into level 1 page table (also called page directory) → start address of the level 2 page table
2. Secondary page number
   - Indexes into level 2 page table → physical page number
3. Offset

Address format with two-level page tables

With 32-bit address space, 4 KB pages

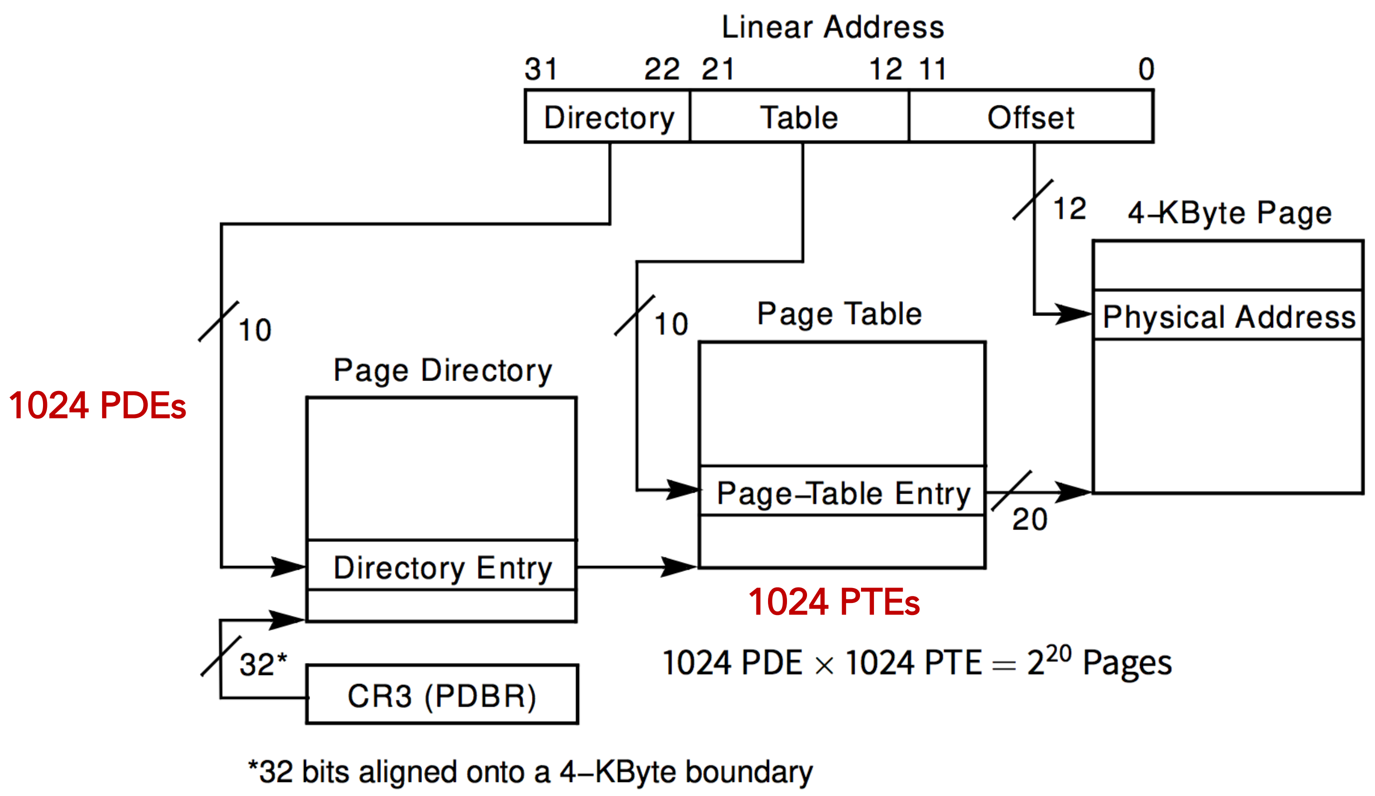
- How many bits in offset? $\log 2(4K)$ = 12 bits
- Assume a single page directory fitting in one page
- Each page directory entry (PDE) is 4 bytes
  - 4 KB / 4 bytes = 1024 PDEs
- Hence, 1024 page tables (one PDE → one page table)
- Directory = 10, offset = 12, inner = 32 – 10 – 12 = 10 bits



Virtual address

- Each page table now fits in one page (i.e., 4 KB) as well

x86 two-level page table



Overhead w/ two-level PT

Overhead of single-level PT

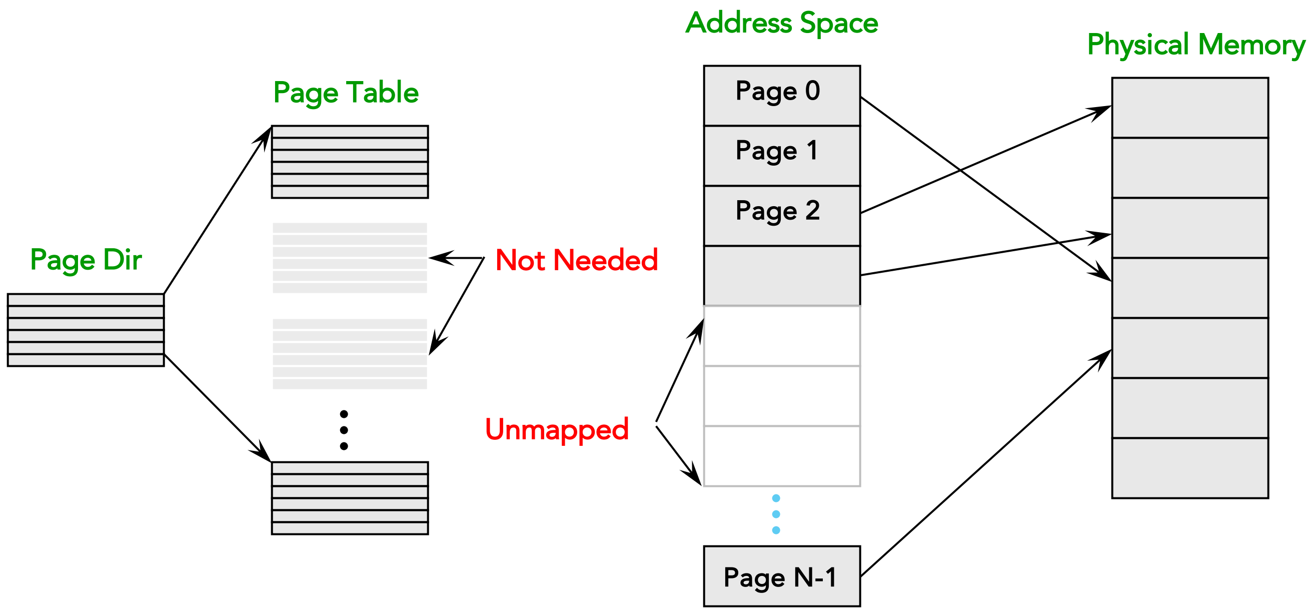
- Each page table costs $2^{32} / 2^{12} \times 4$ B = 4MB

Isn't the overhead the same with two-level PT?!

- 1024 page tables
- Each page table has 2^{10} PTEs, thus has a size of 4KB
- Total size of these page tables is 1024×4KB = 4MB
- We also have one page directory, which is 4KB…

Key: address space should have relatively large unmapped regions

Unmapped region in address space

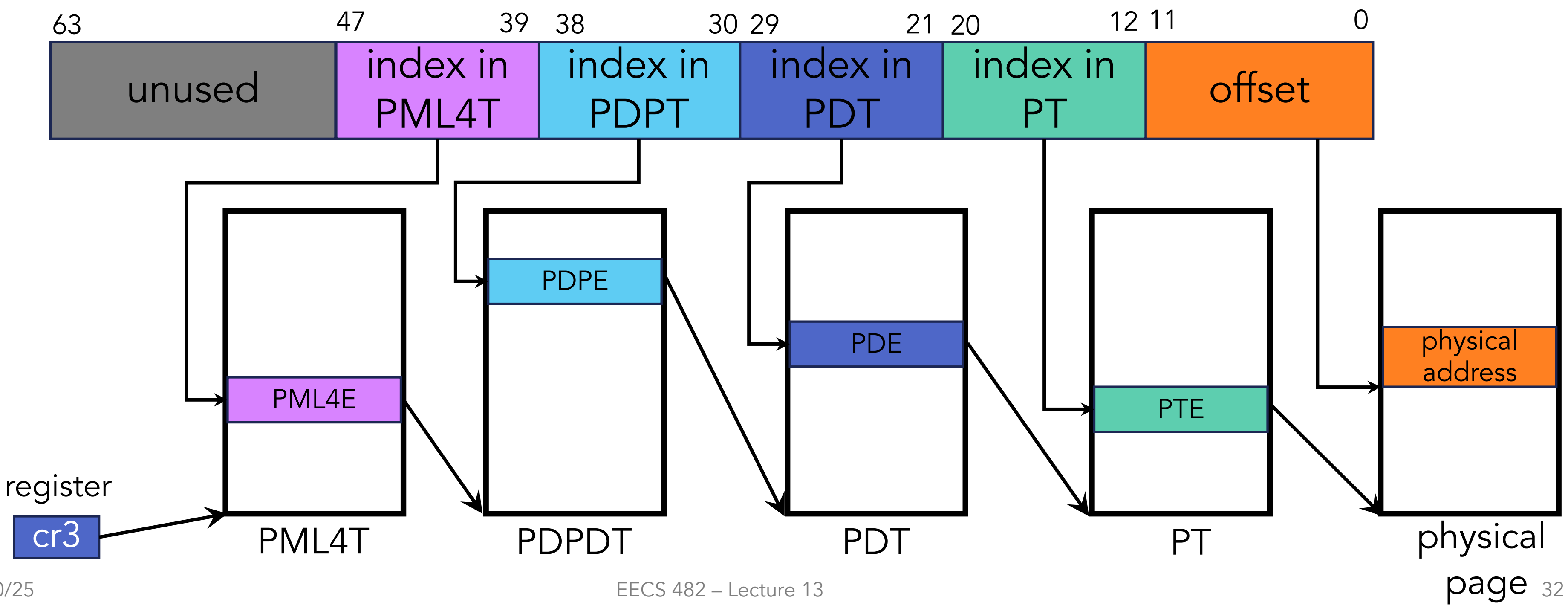


A large unmapped region → no need to allocate page tables entirely, save memory

Multi-level paging

x86_64 uses four-level page table

- Page Map Level 4 Table (PML4T)
- Page Directory Pointer Table (PDPT)
- Page Directory Table (PDT)
- Page Table (PT)



Efficient translations

A limitation of paging is translation overhead

- Doubled the cost of memory access
  - One lookup into the page table, another to fetch the data

Multi-level page table more costly!

- Two-level page table triples the costs
  - Two lookups into the page tables, a third to fetch the data
- 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: a cache for page table entries

- Implemented in hardware, fast
- Cache tags are virtual page numbers
- Cache values are PTEs
- With PTE + offset, can directly calculate physical address

Must invalidate TLB on context switch

- Unless TLB supports encoding address space ID in entries

Managing TLBs

TLBs exploit locality

- High hit rate (handle > 90% translations), but there are misses (TLB miss)

Who handles the TLB miss?

1. Hardware-managed TLB (MMU) [x86]
   - CPU will find the PTE from the page table
     - Called page table walk
   - Fast but inflexible: tables need to in HW-defined format

2. Software-managed TLB (OS) [MIPS, Alpha, Sparc, PowerPC]
   - CPU throws TLB faults to the OS
   - OS finds appropriate PTE, loads it in TLB
   - Slow but flexible: tables can be in any format convenient for OS

Pro tip: indirection + caching

Indirection: solves all problems in CS!
- Except for performance ☹ (or the problem of too many indirections!)

Caching: solves the performance problems caused by indirection
- Pointers added by indirection are small, so are cached easily

Caching and virtual memory

The TLB is a cache for …?

Physical memory is a cache for …?

CPU caches are a cache for …?

How do the concepts from CPU caches (EECS 370) relate to the concepts of virtual memory?

CPU caching and virtual memory

What associativity is used in CPU caches?

- Multiple choices
  - Direct mapped
  - Set associative
  - Fully associative

What associativity is used in VM?

- Fully associative
- Why?

CPU caching and virtual memory

In CPU caches, how do you find data in the set?

- Compare with tags in a cache entry
- Fully associate caches → compare address with every cache block

Will this work for virtual memory?

How would you solve this in EECS 281?