Systems 1 – CS 6013

Computer Architecture and Operating Systems

Lecture 19: VM – Multi-level Paging

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

*(adapted from slides by Ryan Stutsman, Andrea Arpaci-Dusseau, and Sarah Diesburg, and MSD presentations)

Lecture 19 – Topics

- Page Replacement Multi-Level Paging

Miscellaneous

- Questions?

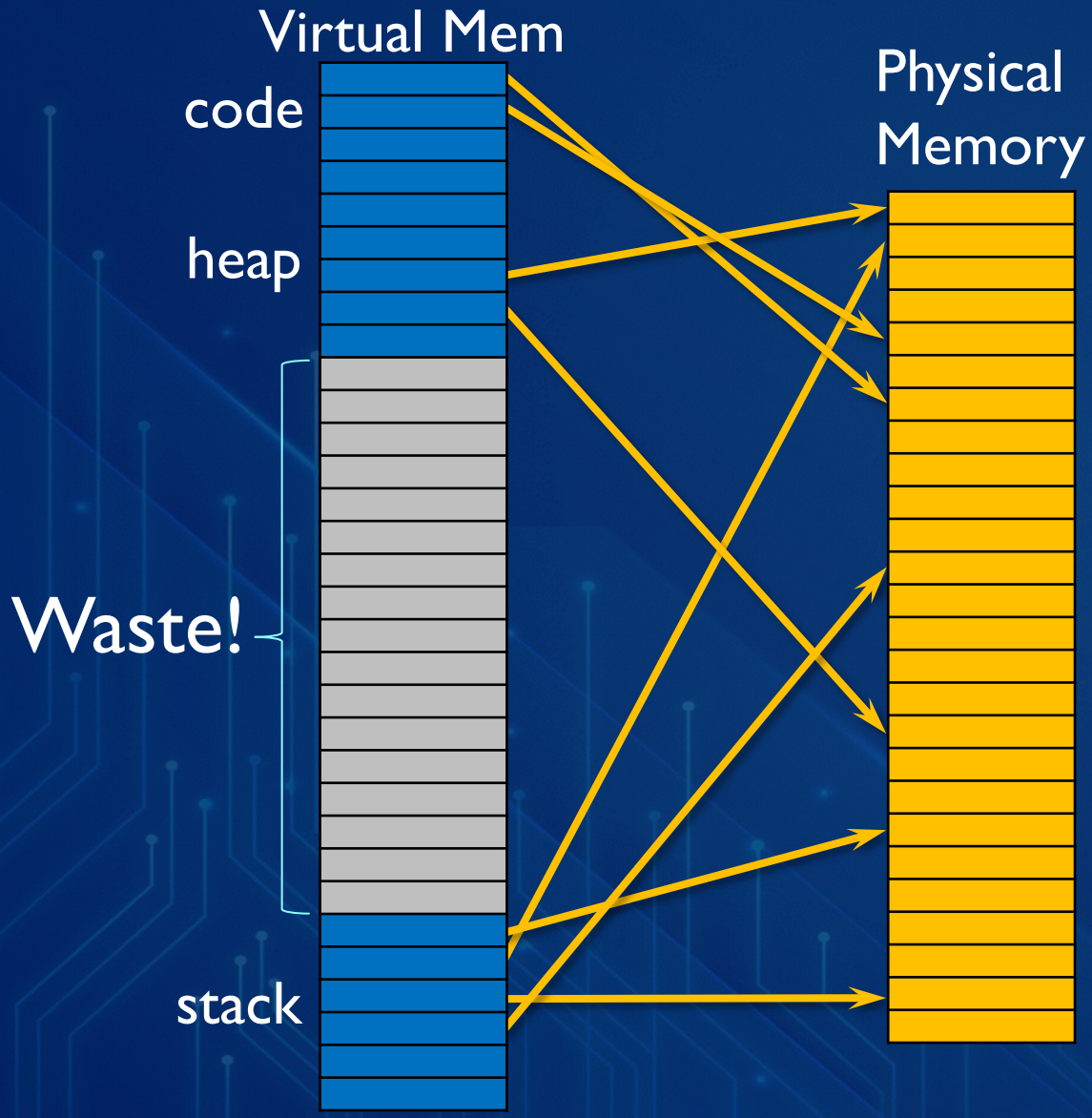
Review

- We time-share the CPU using LDE and scheduling.
- Time sharing memory is very inefficient.
- Instead, OS uses virtual memory concept to share the physical memory among processes.
  - Base and bound: Internal fragmentation.
  - Segmentation: External fragmentation due to variable size segments.
  - Paging: no external fragmentation due to fixed size pages, minimal internal fragmentation, but more complicated memory accesses.

Disadvantages of Paging

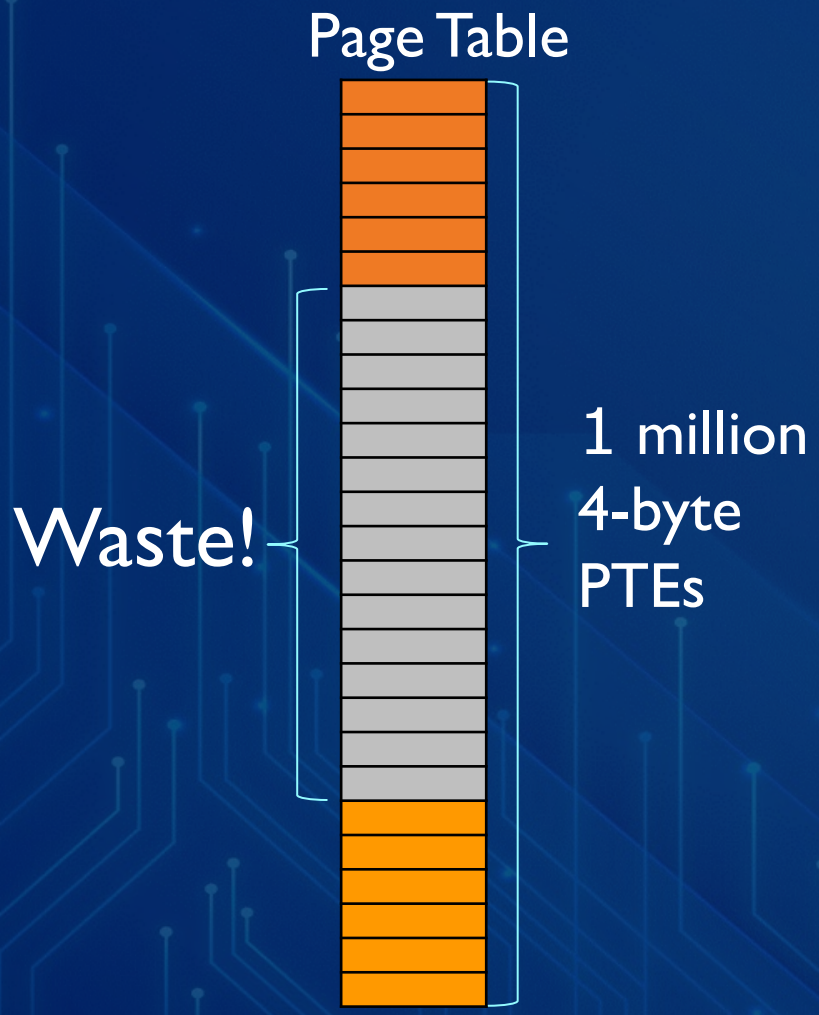
- Extra memory reference must be looked up in page table
  - Very inefficient
  - Page table must be stored in memory
  - MMU stores only base address of page table
  - **Use TLBs** to avoid extra access
- Storage for page tables may be substantial
  - Simple page table requires PTE for all pages in address space
    - Entry needed even if page not allocated
  - Problematic with dynamic stack and heap within address space (today)

Big Tables due to Hole!



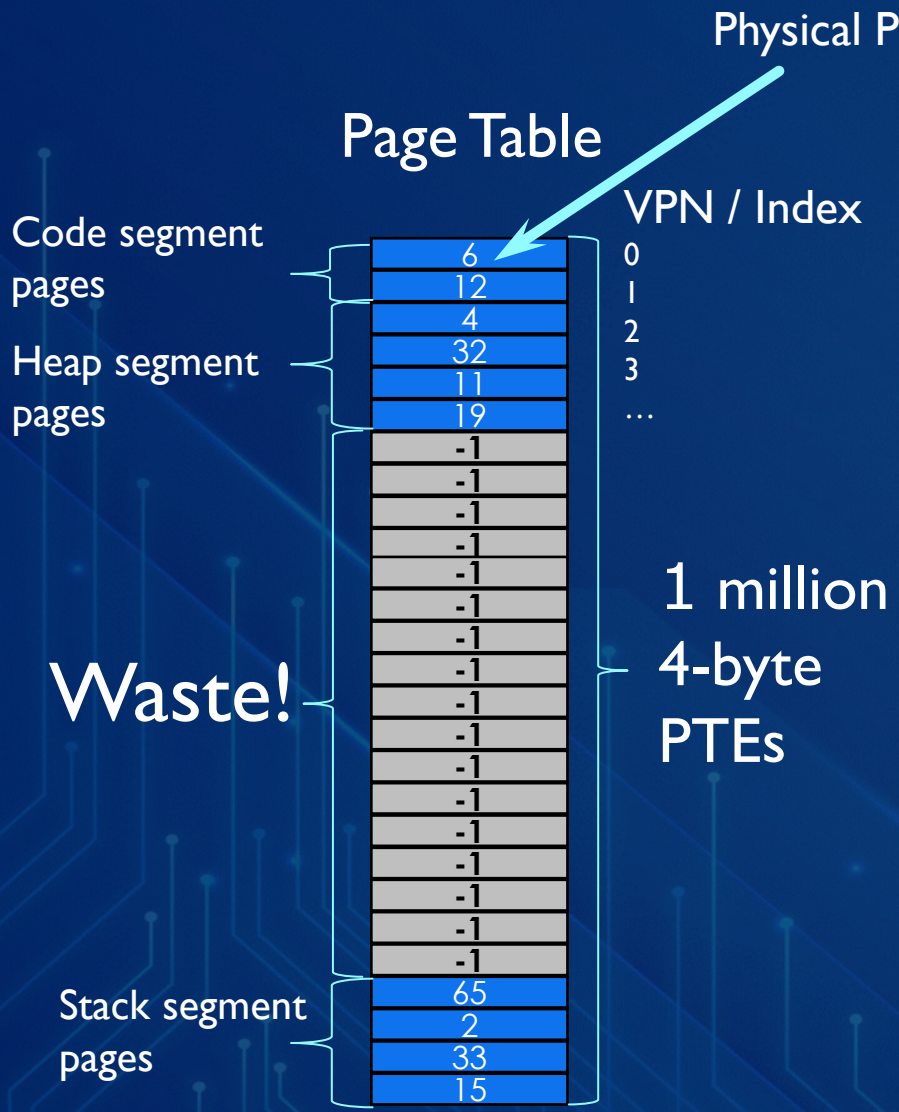
- Dark blue pages are in physical memory… what does that mean?
  - They were accessed recently.
- Purpose of Page Table Entry?
  - Map virtual addresses to the physical address.
- Problem: There must be a Page Table Entry (PTE) for every virtual page…
  - Including all the pages that are not being used.
- This creates "wasted" space in the TLB (Cache)

Most PTEs are Invalid (no PPN)



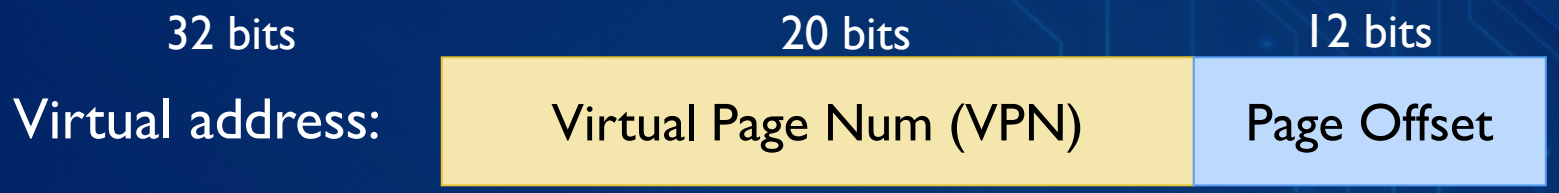
- Page table sits in memory
  - Maps VA to PA
  - [Note: Orange entries represent pages that are currently loaded into physical memory.]
- PT wastes memory on page mappings that are not used.
  - PT for the process is unnecessarily large.
- What is a solution to this?
  - How have we been thinking of (storing) the Page Table?
    - As an Array…
  - So…

Most PTEs are Invalid (no PPN)



- Our current view of a Page Table is just a big array. If we were writing code to do this in C:
- `int * pt = malloc( sizeof( int ) * 1 million)`
  - Use the virtual page number as the index.
- What part of the program is in virtual page 0?
  - Code segment
- Where does virtual page 0 live in physical memory?
  - Index 0 (ie, virtual page 0) gives?
  - pt[0] => 6 // What does 6 mean?
    - Physical page 6 – given to process by the OS.
    - Note PPN 6 is (probably) unrealistic for our user process… why?
      - OS stored in low physical memory
- Note, PPN of -1 means not in physical memory…

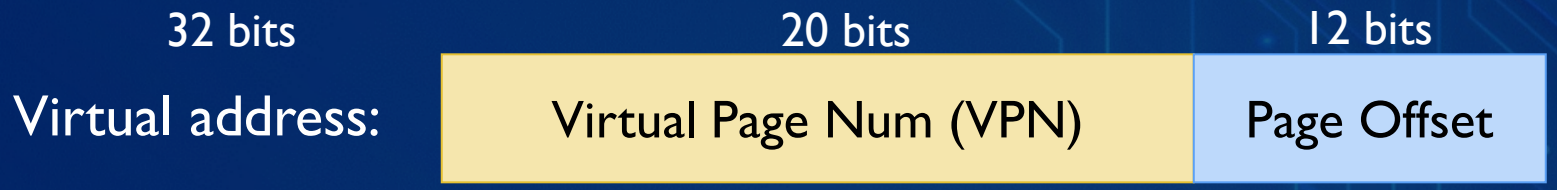
Virtual Address Translation



- Assume each assembly *instruction* is represented by 4 bytes of data…
- How many instructions can your program have in 1 page of memory?
  - 1KB (1024 instructions * 4 bytes each == 4 KB (ie, one page))
  - Which virtual page are these instructions in?
    - Code starts at virtual address 0… So these will be in virtual page 0.
- Consider that each line of code has its own address (everything in memory has its own address) – what do those addresses look like? What are values of the 1st 20 bits of each of those address?
  - All 0! 0000 0000 0000 0000 0000
- What are the offsets of those instructions? How many bits do they use?
  - 12 bits

		Hex Version
• 0th instruction:	0000 0000 0000	– 0x000
• 1th instruction:	0000 0000 0100	– 0x004
• 2nd instruction:	0000 0000 1000	– 0x008
• 3rd instruction:	0000 0000 1100	– 0x00C
• 4th instruction:	0000 0001 0000	– 0x010

Virtual Address Translation

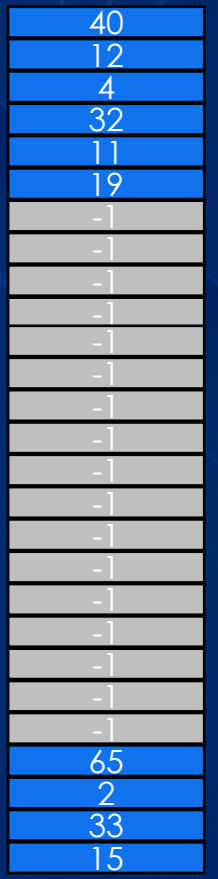


0000 0014 with respect to our address picture

0x00000 0x014

- Full virtual address of 5th instruction?
  - Virtual Page Number + Offset
  - VPN:
    - 0000 0000 0000 0000 0000 ; in hex?
    - 0x00000
  - Offset
    - 5 * 4 (bytes per instruction): 20
    - 20 in hex?
    - 0x014
  - 0000 0014
  - How many bits is 0000 0014?
    - 32, each hex digit is 4 bits
  - Translate the (virtual) address of the 5th instruction into the real (physical) address.

- What do we do with the VPN?
  - Use it to index into page table!
- PPN of physical address is (in hex):
  - 0x40
- Complete physical address:
  - 0x40 concatenated to ???
  - offset, which is the same for both virtual and physical address… so it is:
    - 0x014
  - 0x00400014
- Note, the hardware actually does something like:
  - 0x40 << 12 | 0x014



Avoid Simple Linear Page Table

- Use more complex page tables instead of just big array
  1. Make page tables fit more cleanly into memory
  2. PTE as small as they can be (don't waste physical memory)
- Any data structure is possible with software-managed TLB
  - Hardware looks for VPN in TLB on every memory access
  - If TLB does not contain VPN, TLB miss
    - Trap into OS and let OS find VPN to PPN translation
    - Store new translation back in TLB for future accesses.
- Better name for the TLB?
  - Address Translation Cache (in the MMU)

Other Possible Approaches

1. **Inverted Page Tables**
2. Segmented Page Tables
3. Multi-level Page Tables
   - Page the page tables
   - Page the page tables of page tables…

Inverted Page Tables

- Hash table indexed by VPN + shortened PID (ASID – Address Space Id)
  - Only one table for whole system rather than per process
  - Only contains a number of entries equal to allocated page frames (size of PT is independent of number of VPNs)
- Complex data structure (Not used all that often)
  - Only done under software-controlled TLB
- On TLB miss
  - OS handles trap
  - Looks up hash(VPN+ASID) to find PPN
  - Populates TLB entry for VPN, ASID, PPN combination
  - Returns from trap
- This would be very complicated to do in Hardware (in the CPU)…
  - For hw-controlled TLB, we need a well-defined, simple approach…

Other Approaches

1. Inverted Page Tables
2. **Segmented Page Tables**
3. Multi-level Page Tables
   - Page the page tables
   - Page the page tables of page tables…

Combine Paging and Segmentation

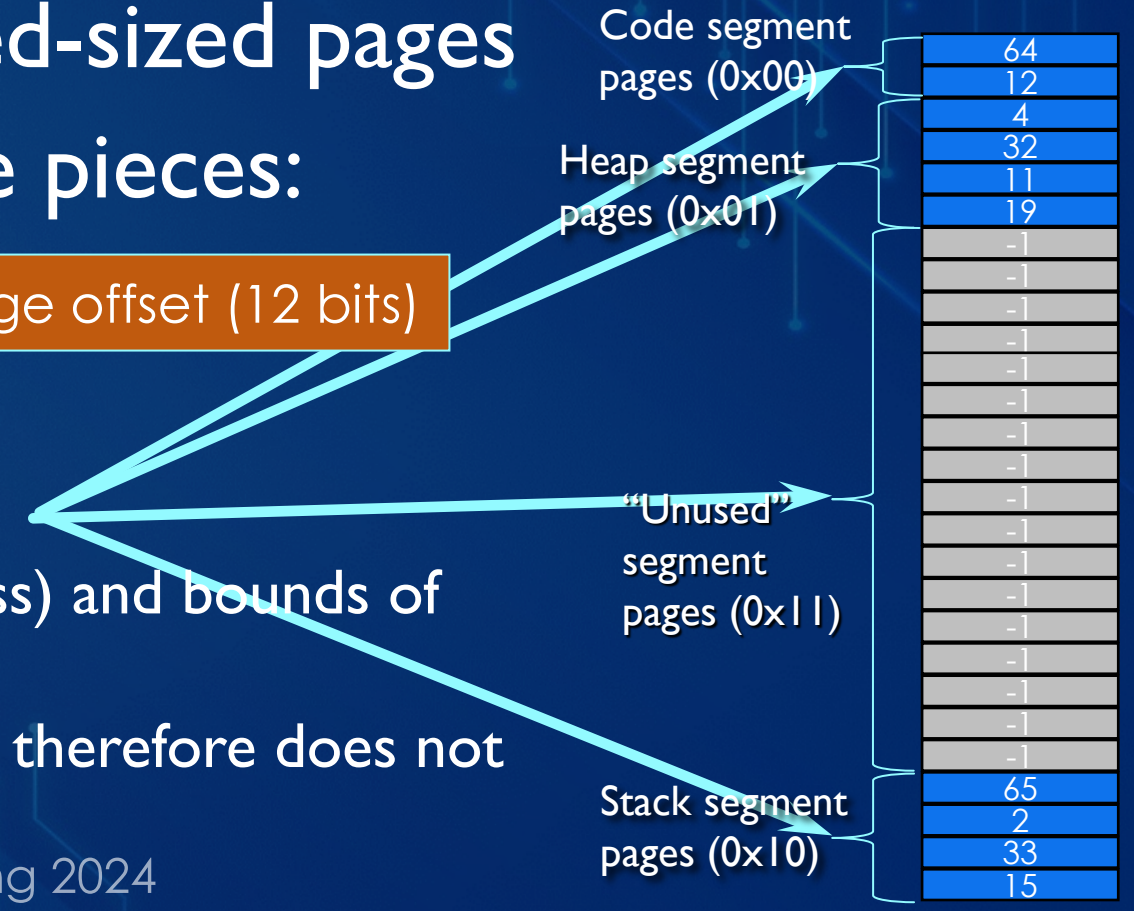
Divide address space into segments (code, heap, stack)

- Segments can be variable length
- Divide each segment into fixed-sized pages

Logical address divided into three pieces:

Implementation

- Each segment has a page table (4 Tables!)
- Each segment tracks base (physical address) and bounds of page table for that segment
- "Wasted" (Unused) portion of page table therefore does not have to be kept in memory…

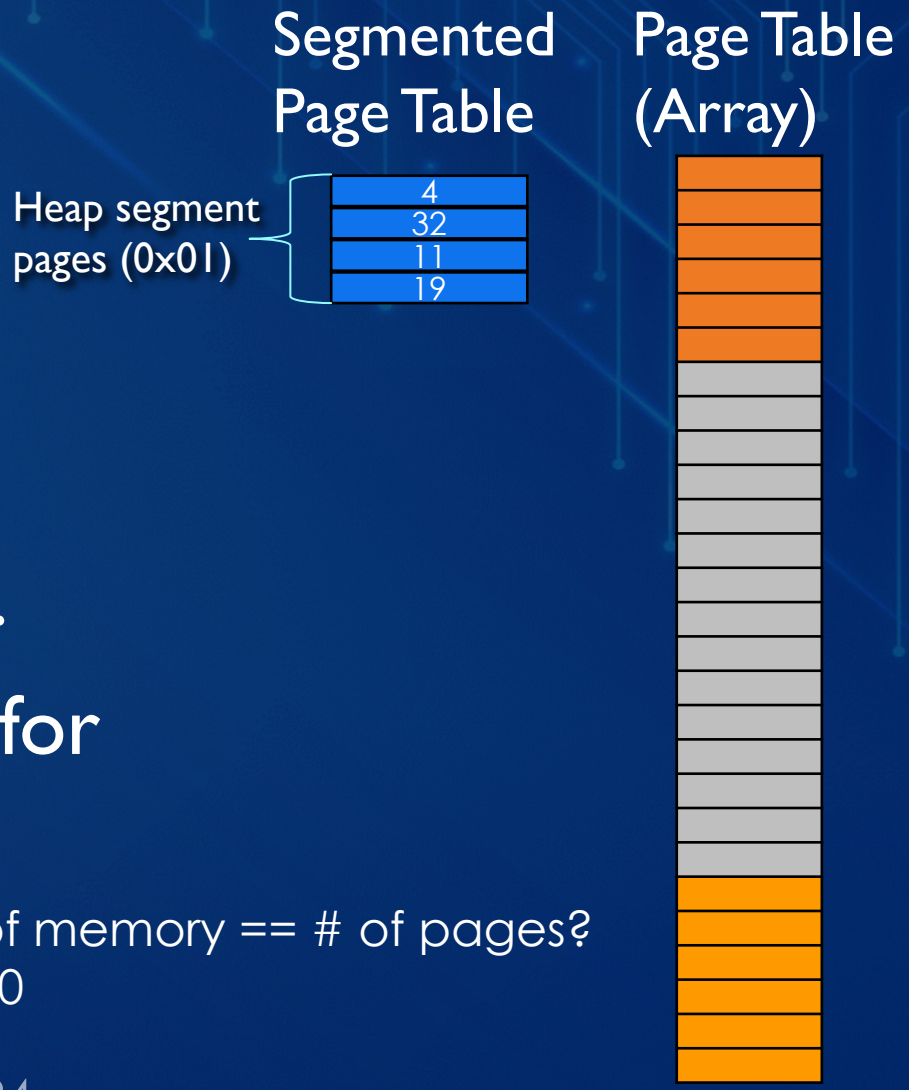


+/- of Segmentation & Paging

- Advantages of Segments
  - Supports sparse address spaces
  - Decreases size of page tables
    - If segment not used, no need for page table
- Advantages of Pages
  - No external fragmentation
  - Segments can grow without any reshuffling
  - Process can run when some pages are swapped to disk (next lecture)
- Advantages of Both
  - Increases flexibility of sharing
  - Share either single page or entire segment
- Still leaves us with external fragmentation of memory due to variable sized page tables
  - Don't forget page tables are stored in RAM!

Other Approaches

1. Inverted Page Tables
2. Segmented Page Tables
3. **Multi-level Page Tables**
   - Page the page tables
   - Page the pages of page tables…
   - Solve the problem of the need for contiguous memory…



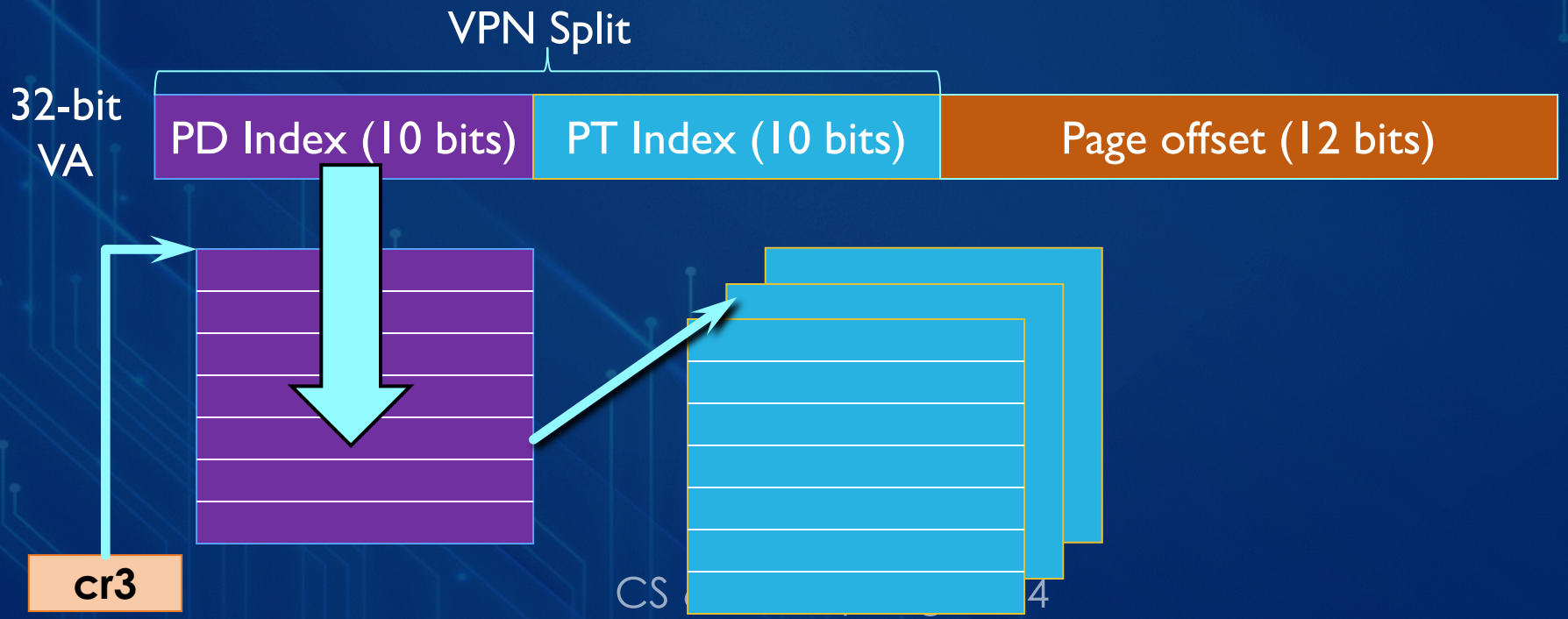
- 30 MB of memory == # of pages?
  - 7500

Multi-level Page Tables

Goal: Let page tables be allocated non-contiguously

Idea: Page the page tables

- Creates multiple levels of page tables; outer level "**page directory**"
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)

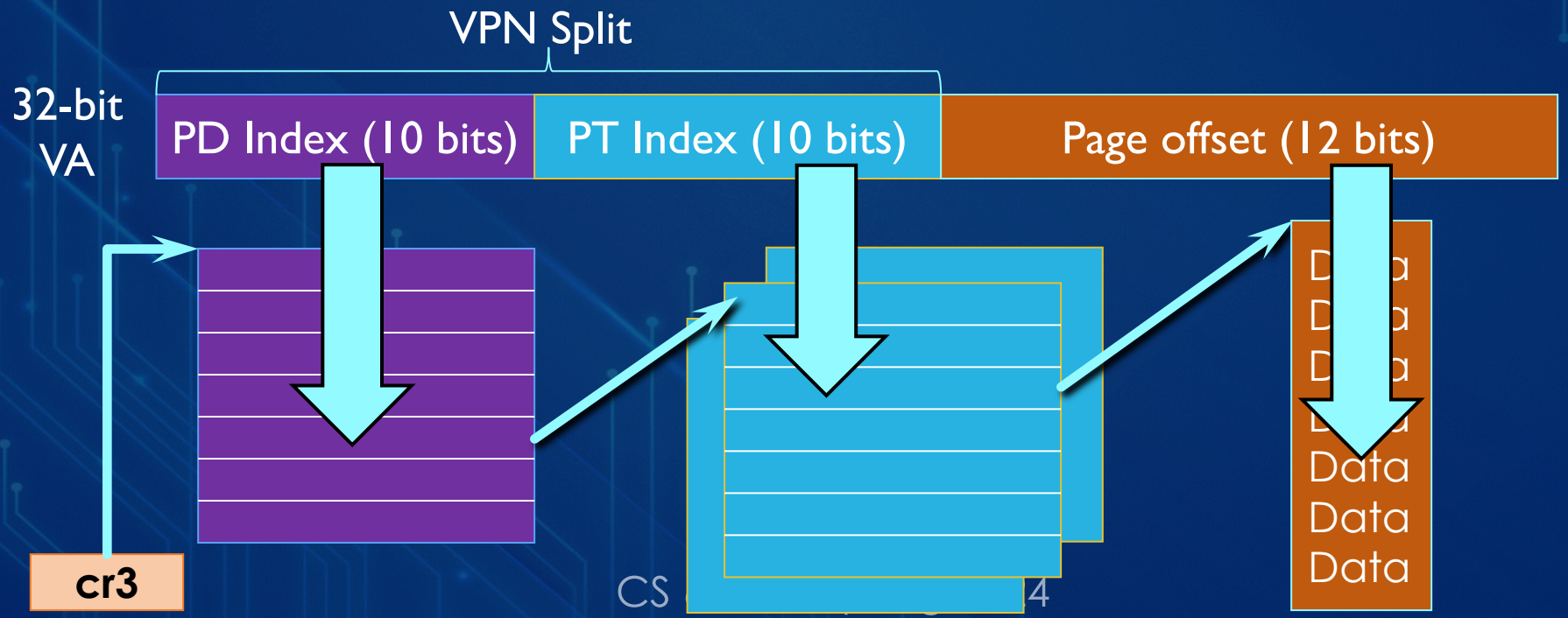


Multi-level Page Tables

Goal: Let page tables be allocated non-contiguously

Idea: Page the page tables

- Creates multiple levels of page tables; outer level "**page directory**"
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)



Address Format for Two-Level PT

How should logical address be structured?

- How many bits for each paging level?

Goal / Implementation…

- **Each page table fits within a page!**
- PTE size * number PTE = page size
  - Assume PTE size = 4 bytes [an address]
  - Page size = 2^{12} bytes = 4KB
  - 4 bytes * number PTE = 2^{12} bytes [Remember 4 == 2^2]
  - → number PTE per page = 2^{10}
- → # bits for selecting inner page = 10

Remaining bits for Page Directory:

- 30 – 10 – 12 = 8 bits

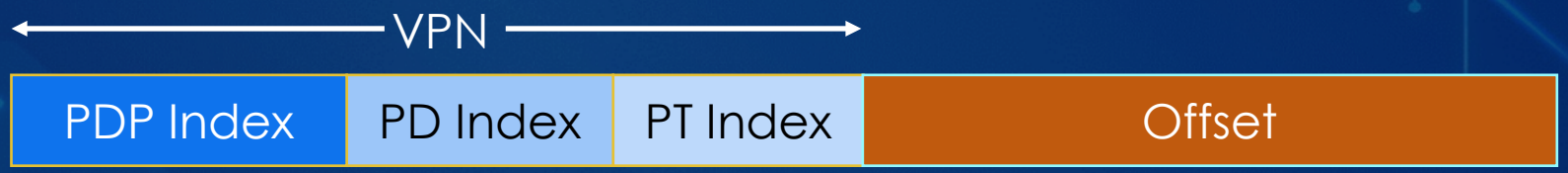
This is key to avoid external fragmentation!

Problem with 2 Levels?

Problem: page directories (outer level) may not fit in a page

Solution:

- Split page directories into pieces
- Have a directory of page directories of page tables – Multi-level Page Tables!



How large is virtual address space with 4 KB pages, 4 byte PTEs, each page table fits in page given 1, 2, 3 levels?

4 KB / 4 bytes → 1024 entries per level
1 level: 1024 * 4 KB = 2^{22} bytes = 4 MB
2 levels: 1024 * 4 MB = 2^{32} bytes = 4 GB
3 levels: 1024 * 4 GB = 2^{42} bytes = 4 TB

- In modern hardware, the virtual address space is much (>>>>) larger than the physical address space.
- Speed implications?

Multi-Level Page Tables

- Advantages:
  - Solves the wasted space problem
- There are some (minor?) disadvantages:
  - Complexity
  - On a TLB miss, multiple memory accesses are required to get the new Page Table information
    - *We assume most TLB lookups will be hits…

Page Tables Summary

- Linear page tables require too much contiguous (and active) memory
  - Wastes (a lot of) space with invalid entries
- Many options for efficiently organizing page tables
- If OS traps on TLB miss (ie, OS managess the TLB), OS can use any data structure
  - Eg: Inverted page tables (hashing)
- If hardware handles TLB miss, page tables must follow hw format
  - Multi-level page tables used in x86 architecture
  - Each page table fits within a page
  - Page directory indexes over a process' page tables.

Questions For The Viewer

- If an ASID uses 8-bits, how many unique processes can the TLB support?
- Yet, computer systems often have many more than that number of processes running at the same time…
  - How is this handled?
- If Caches are really fast, why don't we just make bigger caches and put everything in them?
- RAM – Implies you can access any (random) memory location as quickly as any other location. Why, in practice, is this not usually quite true?
- Increasing page size directly correlates to smaller page tables… Why is this not a good solution?
- What is the dirty bit for a Page Table Entry used for? Why is this important?

~ Fin ~