CS 153
Design of Operating Systems

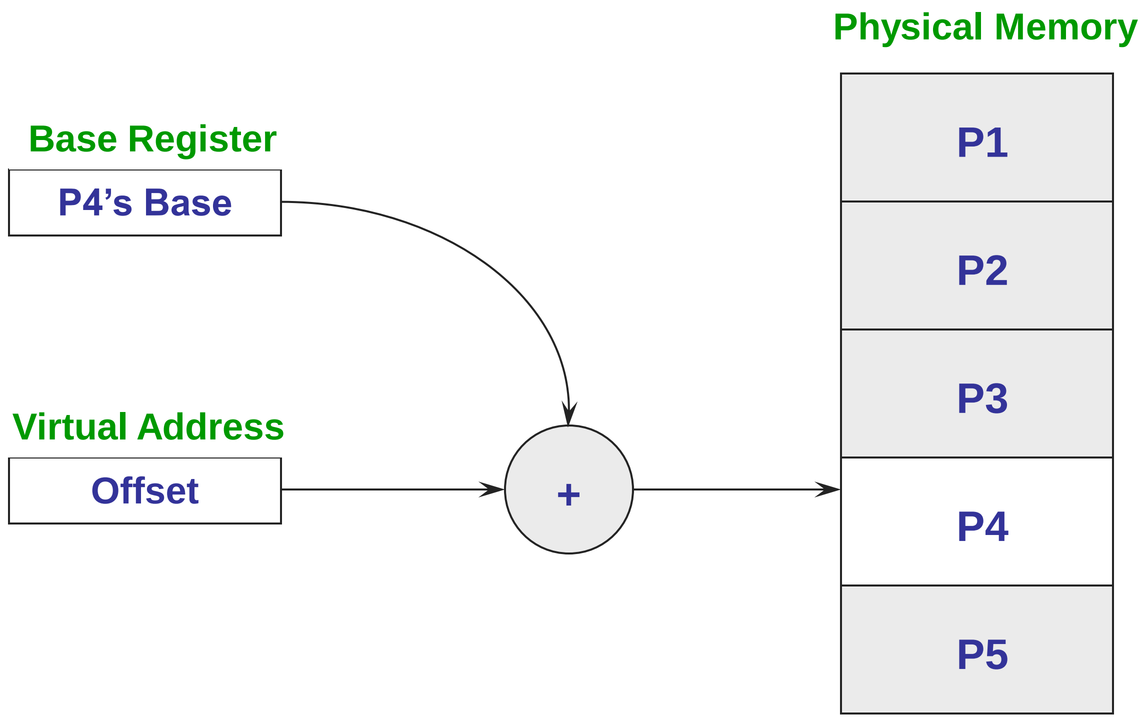
Winter 2016

Lecture 16: Memory Management and Paging

Announcement

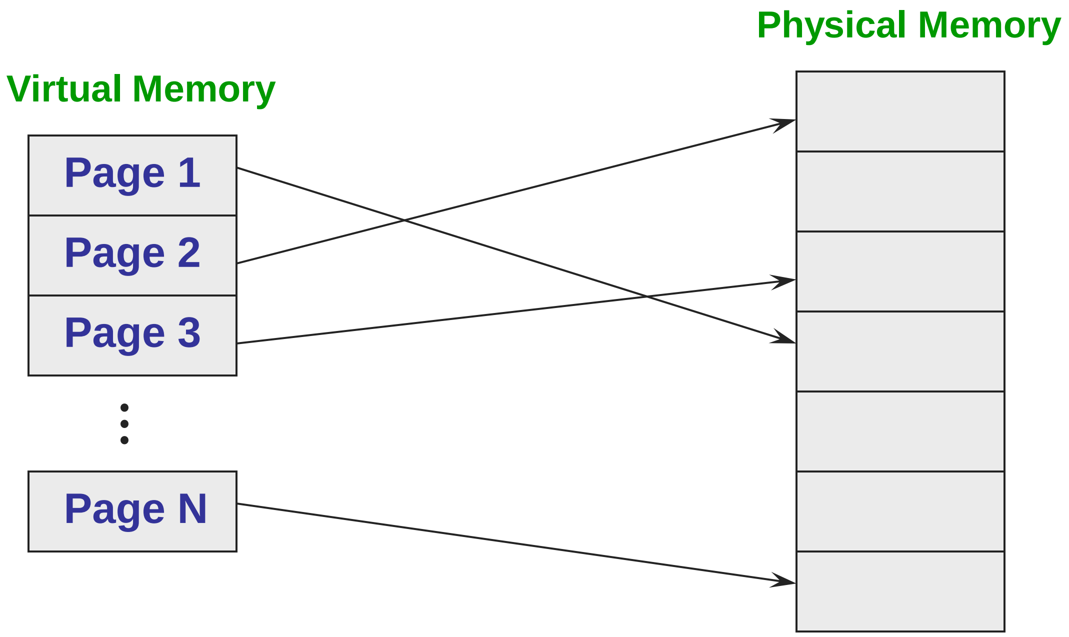
- Homework 2 is out
  - To be posted on ilearn today
  - Due in a week (the end of Feb 19th).

Recap: Fixed Partitions



Recap: Paging

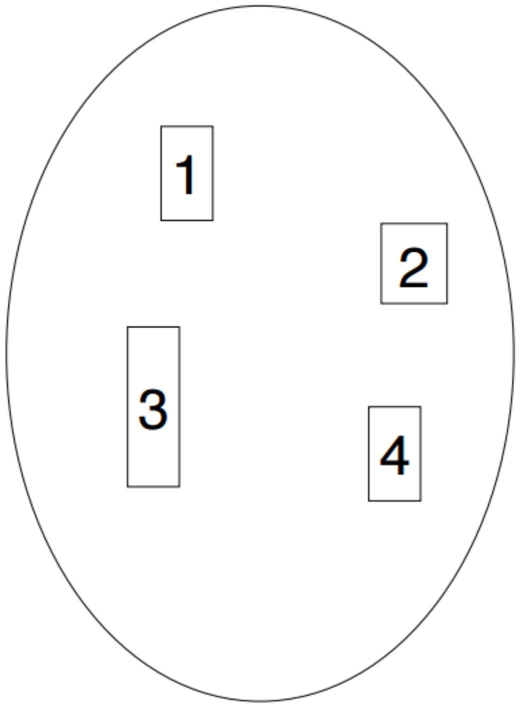
- Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory



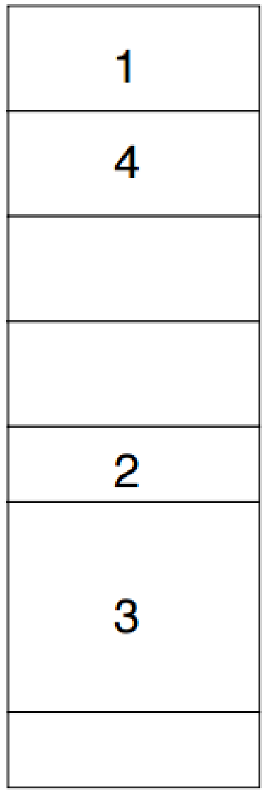
Segmentation

- Segmentation is a technique that partitions memory into logically related data units
  - Module, procedure, stack, data, file, etc.
  - Units of memory from user's perspective
- Natural extension of variable-sized partitions
  - Variable-sized partitions = 1 segment/process
  - Segmentation = many segments/process
  - Fixed partition : Paging :: Variable partition : Segmentation
- Hardware support
  - Multiple base/limit pairs, one per segment (segment table)
  - Segments named by #, used to index into table
  - Virtual addresses become <segment #, offset>

Segmentation

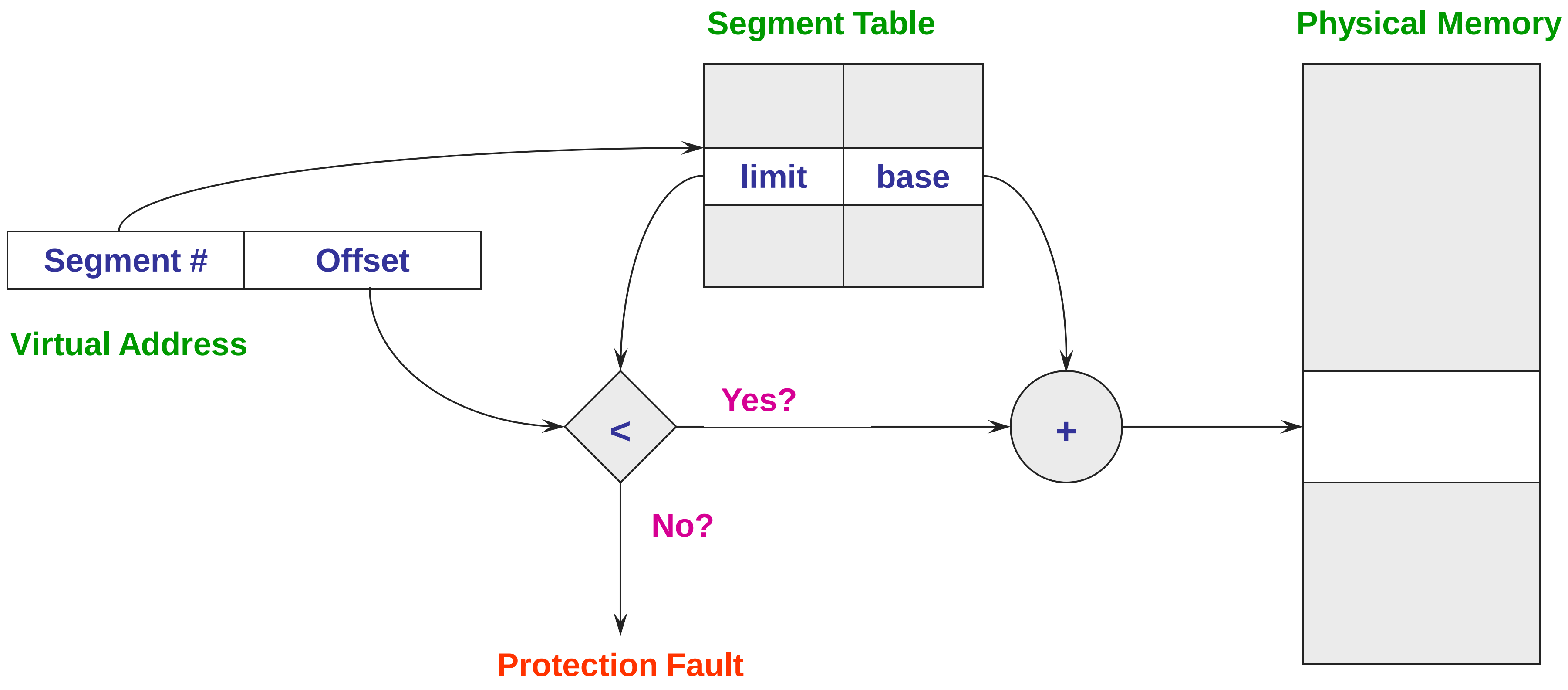


Program perspective



Physical memory

Segment Lookups



Segmentation Example

- Assuming we have 4 segments, we need 2 bits in virtual address to represent it

2	30
Seg#	**Offset**

- 0x01234567 goes to segment #0, whose base is 0x01000000, then physical address is 0x02234567

Segmentation fault?

Segmentation and Paging

- Can combine segmentation and paging
  - The x86 supports segments and paging
- Use segments to manage logically related units
  - Module, procedure, stack, file, data, etc.
  - Segments vary in size, but usually large (multiple pages)
- Use pages to partition segments into fixed size chunks
  - Makes segments easier to manage within physical memory
    - » Segments become “pageable” – rather than moving segments into and out of memory, just move page portions of segment
  - Need to allocate page table entries only for those pieces of the segments that have themselves been allocated
- Tends to be complex…

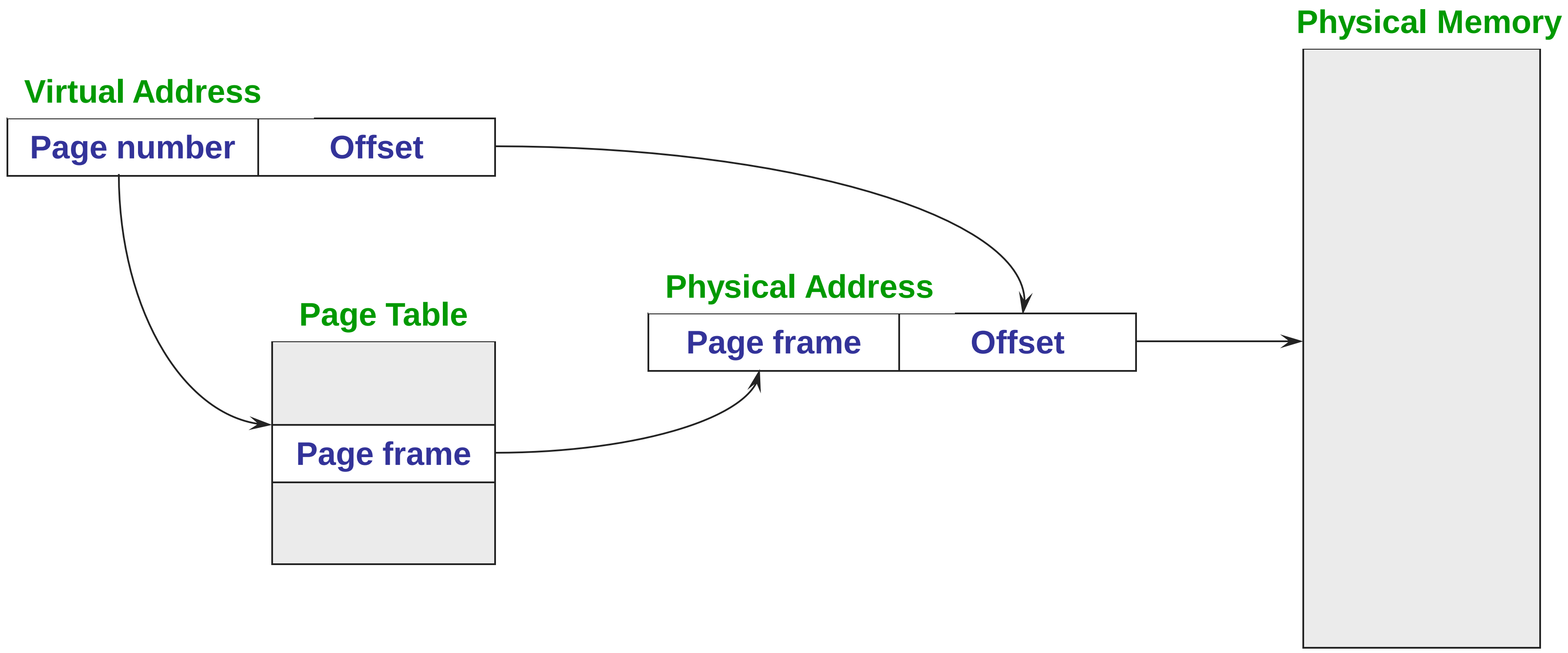
Managing Page Tables

- Last lecture we computed the size of the page table for a 32-bit address space w/ 4K pages to be 4MB
  - This is 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
- Use another level of indirection: two-level page tables

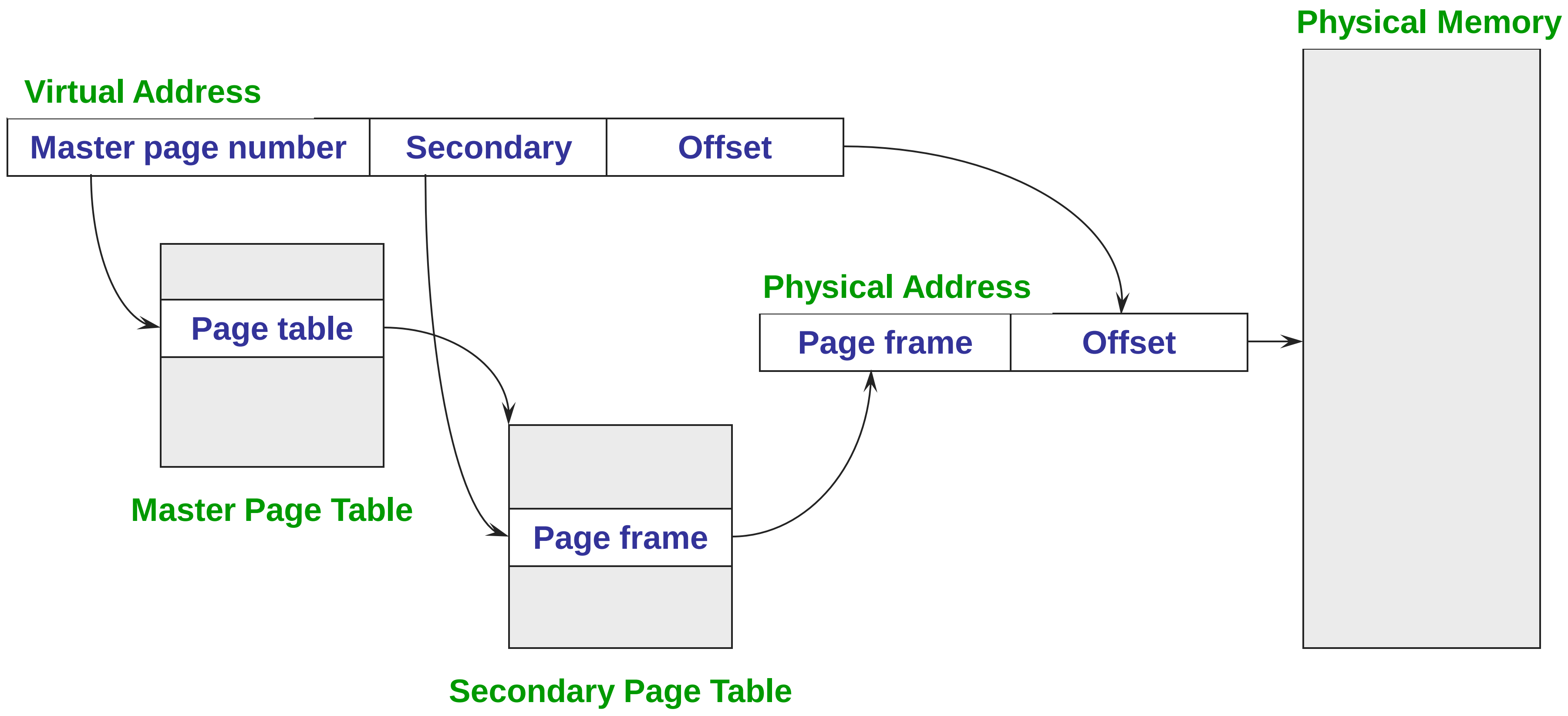
Two-Level Page Tables

- Two-level page tables
  - Virtual addresses (VAs) have three parts:
    - Master page number, secondary page number, and offset
  - Master page table maps VAs to secondary page table
  - Secondary page table maps page number to physical page
  - Offset indicates where in physical page address is located

Page Lookups



Two-Level Page Tables



Example

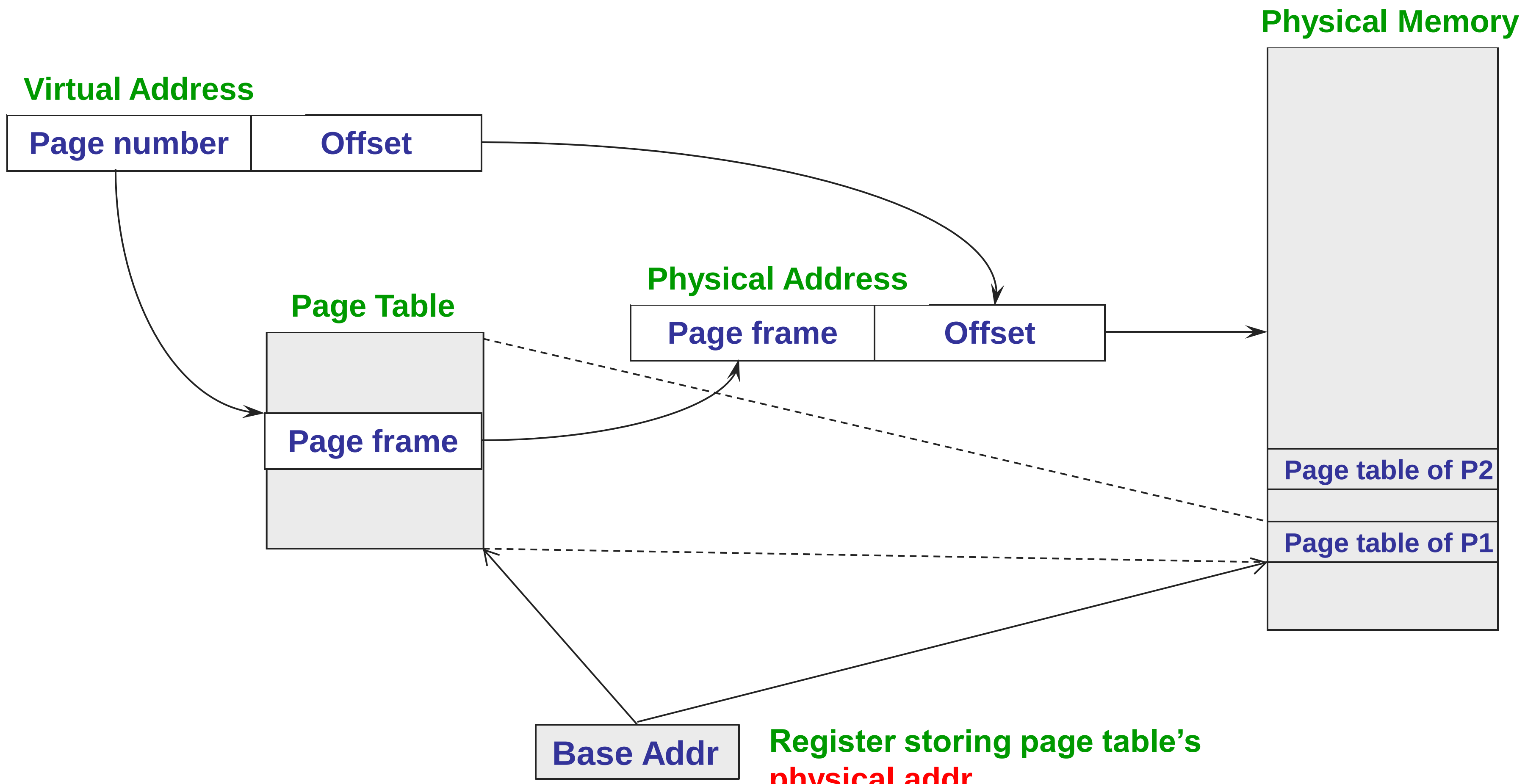
- How many bits in offset? 4K = 12 bits
- 4KB pages, 4 bytes/PTE
- Want master page table in one page: 4K/4 bytes = 1K entries
- Hence, 1K secondary page tables
- How many bits?
  - Master page number = 10 bits (because 1K entries)
  - Offset = 12 bits
  - Secondary page number = 32 – 10 – 12 = 10 bits

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)

Page Table in Physical Memory



Register storing page table's physical addr

Two-level Paging

- Two-level paging reduces memory overhead of paging
  - Only need one master page table and one secondary page table when a process begins
  - As address space grows, allocate more secondary page tables and add PTEs to master page table

- What problem remains?
  - Hint: what about memory lookups?

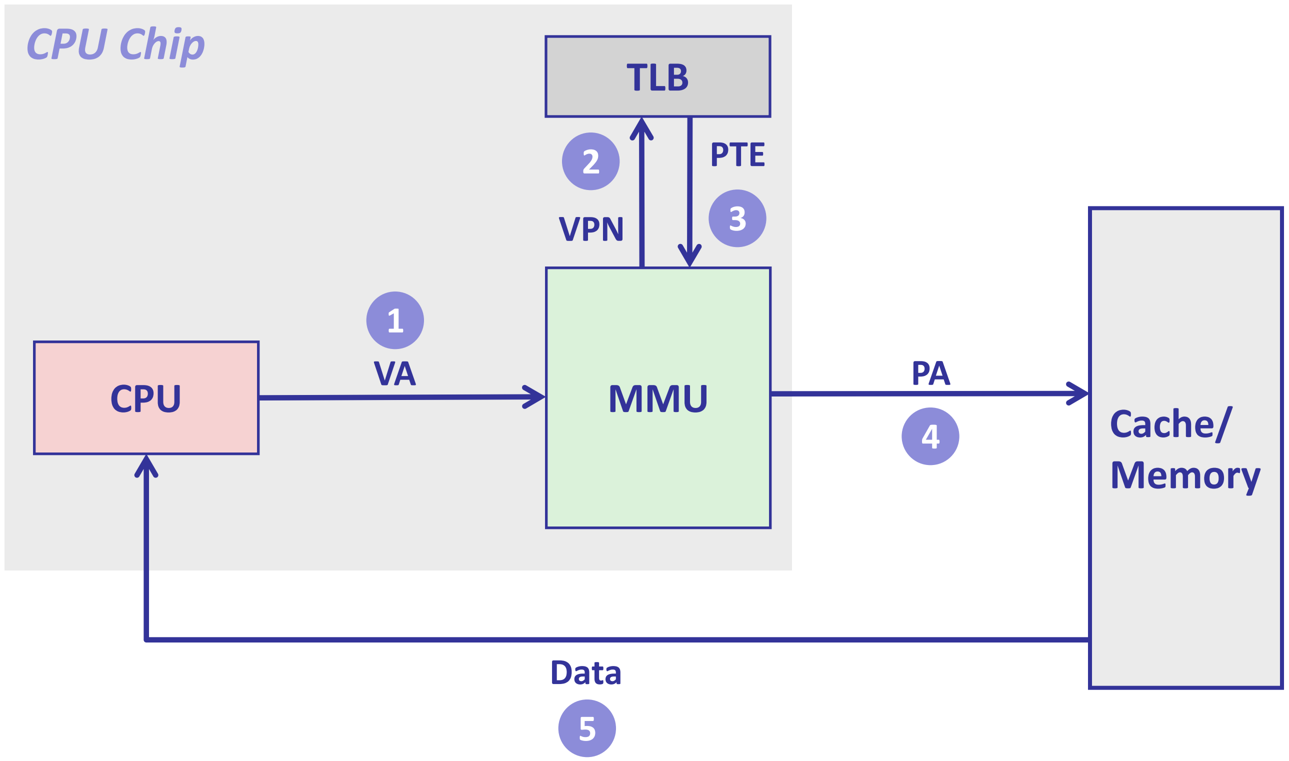
Efficient Translations

- Recall that our original page table scheme doubled the latency of doing memory lookups
  - One lookup into the page table, another to fetch the data
- Now two-level page tables triple the latency!
  - 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 have lookups cost about the same as fetching from memory?
  - Cache translations in hardware
  - Translation Lookaside Buffer (TLB)
  - TLB managed by Memory Management Unit (MMU)

TLBs

- Translation Lookaside Buffers
  - Translate virtual page #s into PTEs (not physical addrs)
  - Can be done in a single machine cycle
- TLBs implemented in hardware
  - Fully associative cache (all entries looked up in parallel)
    - » Keys are virtual page numbers
    - » Values are PTEs (entries from page tables)
  - With PTE + offset, can directly calculate physical address
- Why does this help?
  - Exploits locality: Processes use only handful of pages at a time
    - » 16-48 entries/pages (64-192K)
    - » Only need those pages to be "mapped"
  - Hit rates are therefore very important

TLB Hit



A TLB hit eliminates one or more memory accesses

Managing TLBs

- Hit rate: 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 (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 loaded 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 (special hardware instruction)
- Reload TLB on a process context switch
  - Invalidate all entries
  - Why? Who does 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., Least Recently Used (LRU)

Summary

- Virtual memory
  - Fixed partitions – easy to use, but internal fragmentation
  - Variable partitions – more efficient, but external fragmentation
  - Paging – use small, fixed size chunks, efficient for OS
  - Segmentation – manage in chunks from user’s perspective
  - Combine paging and segmentation to get benefits of both
- Optimizations
  - Managing page tables (space)
  - Efficient translations (TLBs) (time)

Next time...

- Read chapters 8 and 9 in either textbook