10. More Paging

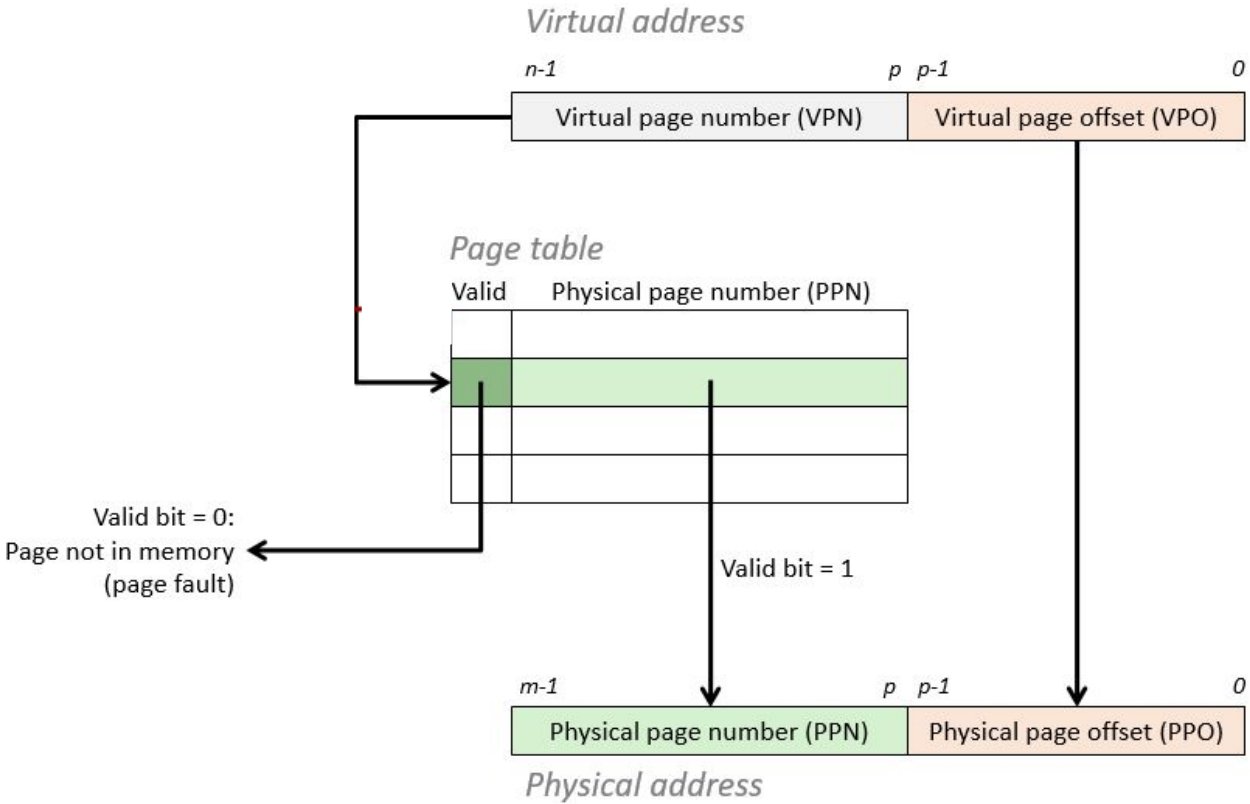
CS 4352 Operating Systems

Adopted from slides of Drs. Silberschatz, Huang, & Bryant

Review

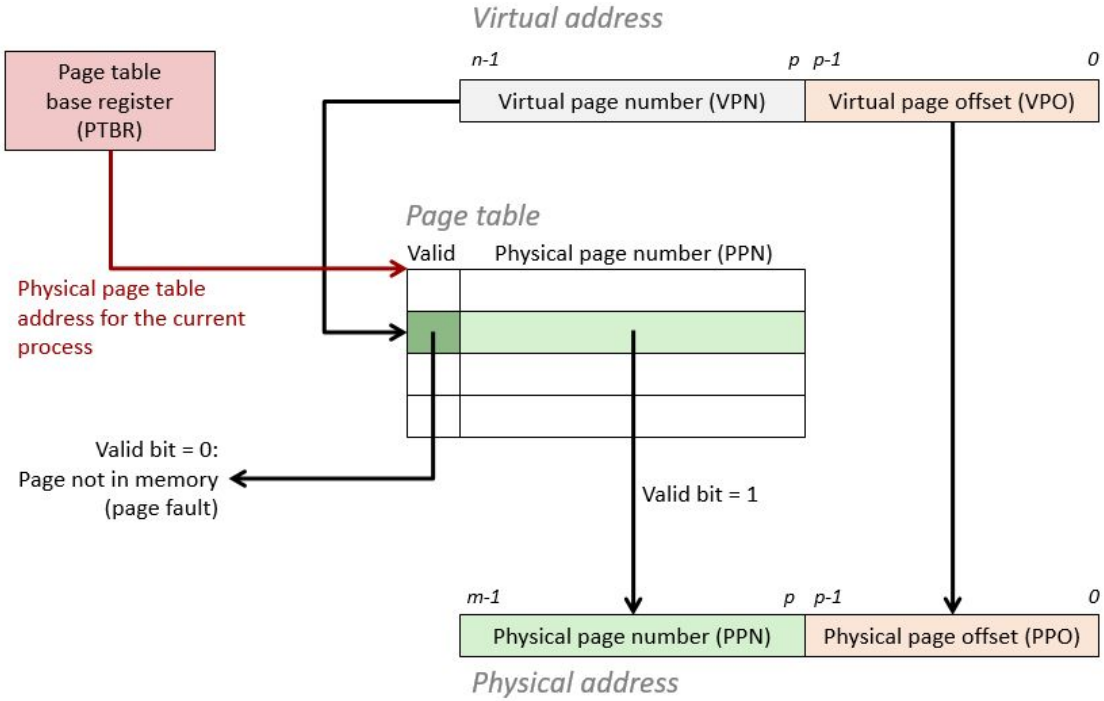
- Simple systems (e.g., elevators) might use physical addressing
  - The addresses used by a program correspond to real, physical addresses
  - Also called single memory, or flat address space
- All modern operating systems use virtual addressing
  - The addresses used by a program are translated to physical addresses
  - This translation is done mostly in hardware; the OS plays a big role in setting up data structures used by the hardware

Basic Address Translation



How to Find Page Table

- Use a special register (page table base register) to hold the physical address of the page table
  - On x86, CR3 register is used
  - Stored in PCB
- Context switch
  - PTBR is loaded a new value
    - Memory mapping



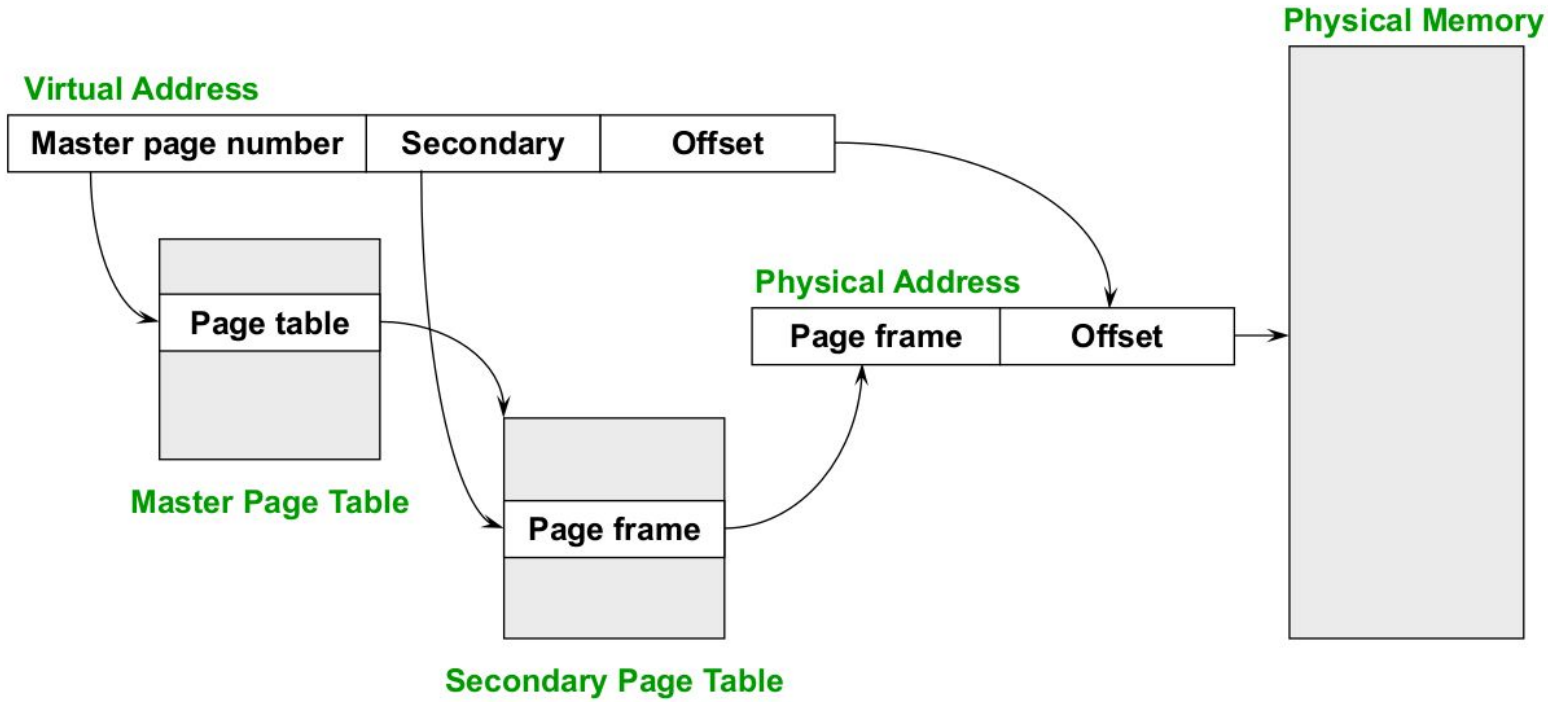
Managing Page Tables

- A single level page table usually takes up too much space
  - 32-bit address space, 4KB page size, 4-byte PTE
    - Need 2^20 PTEs; each is 4 bytes
    - 2^22 = 4MB -- not too bad, but still quite a bit
  - How about 48-bit address space, 4KB page size, 8-byte PTE
    - 2^39 bytes = 512 GB -- way too big!
- How can we reduce this overhead?
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire address space)
- How do we only map what is being used?
  - Can dynamically extend page table...
- Levels of indirection!

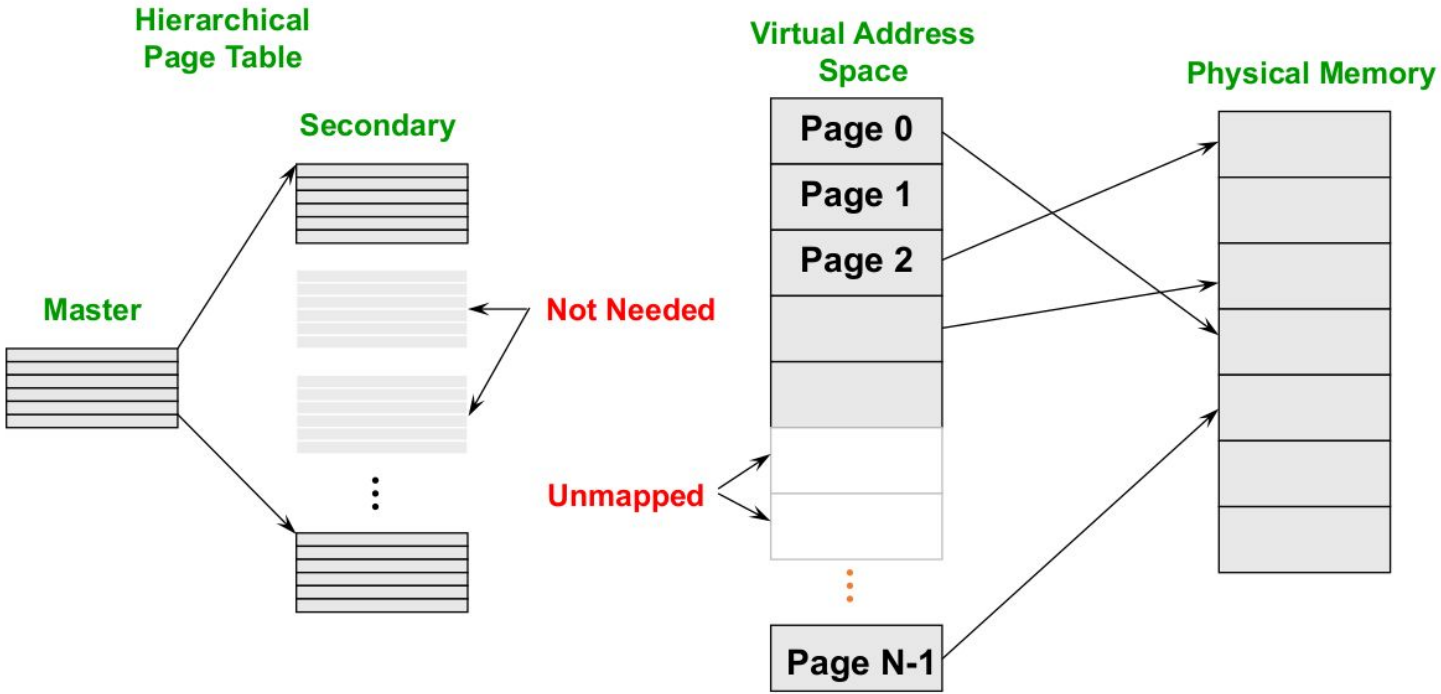
Two-Level Page Tables

- Two-level page tables
  - Virtual address 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 frame
  - Offset indicates where in physical page address is located
- Example
  - 4KB pages, 4 bytes/PTE, how many bits in offset?
    - 12 bits
  - Want master page table in one page, how many entries?
    - 4KB / 4 bytes = 1024 entries
  - 1024 secondary page tables. How many bits?
    - Master = 10, offset = 12, inner = 32 – 10 – 12 = 10 bits

Two-Level Page Tables

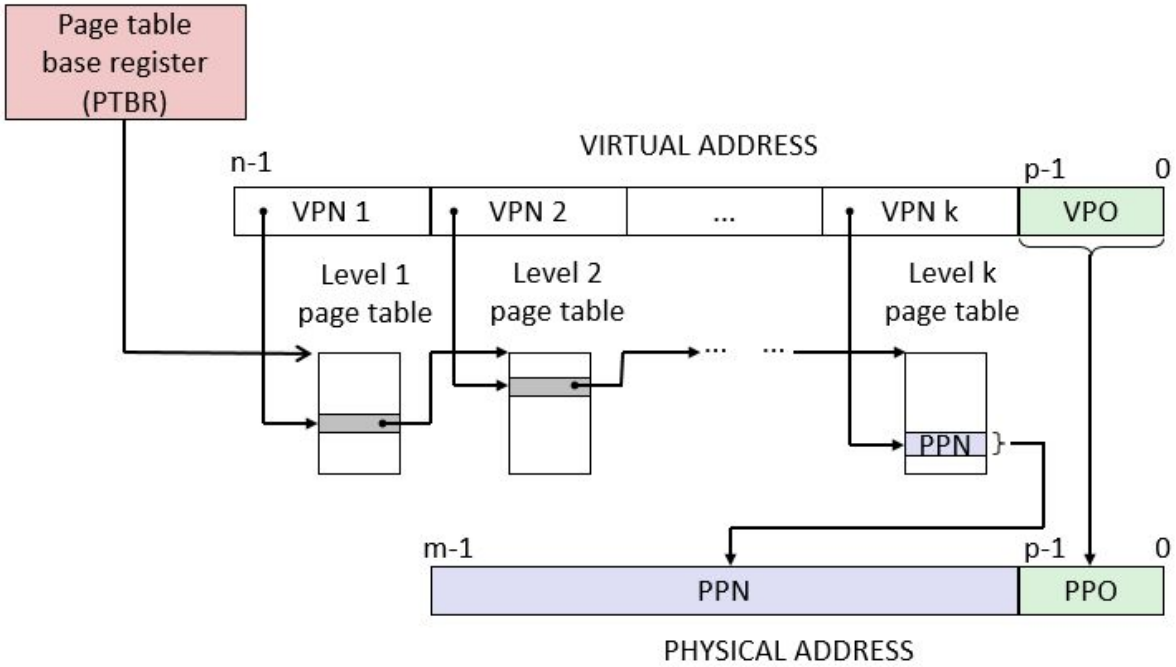


Page Table Evolution



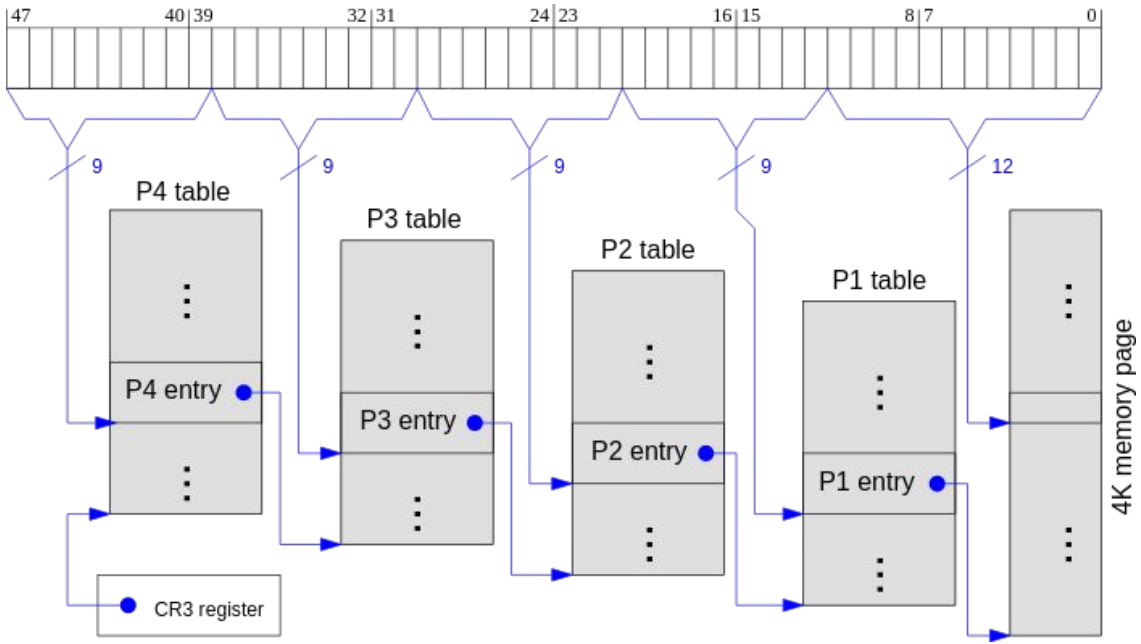
Generic Multi-Level Page Table

- Use PTBR to hold the physical address of the topmost table
  - This table normally occupy exactly one page frame



Modern OS + x86-64 Example

- 64-bit OS uses 48-bit virtual addresses and 4KB pages (normally)
  - 12 bits for offset; 9 bits for each lookup level



https://os.phil-opp.com/page-tables/

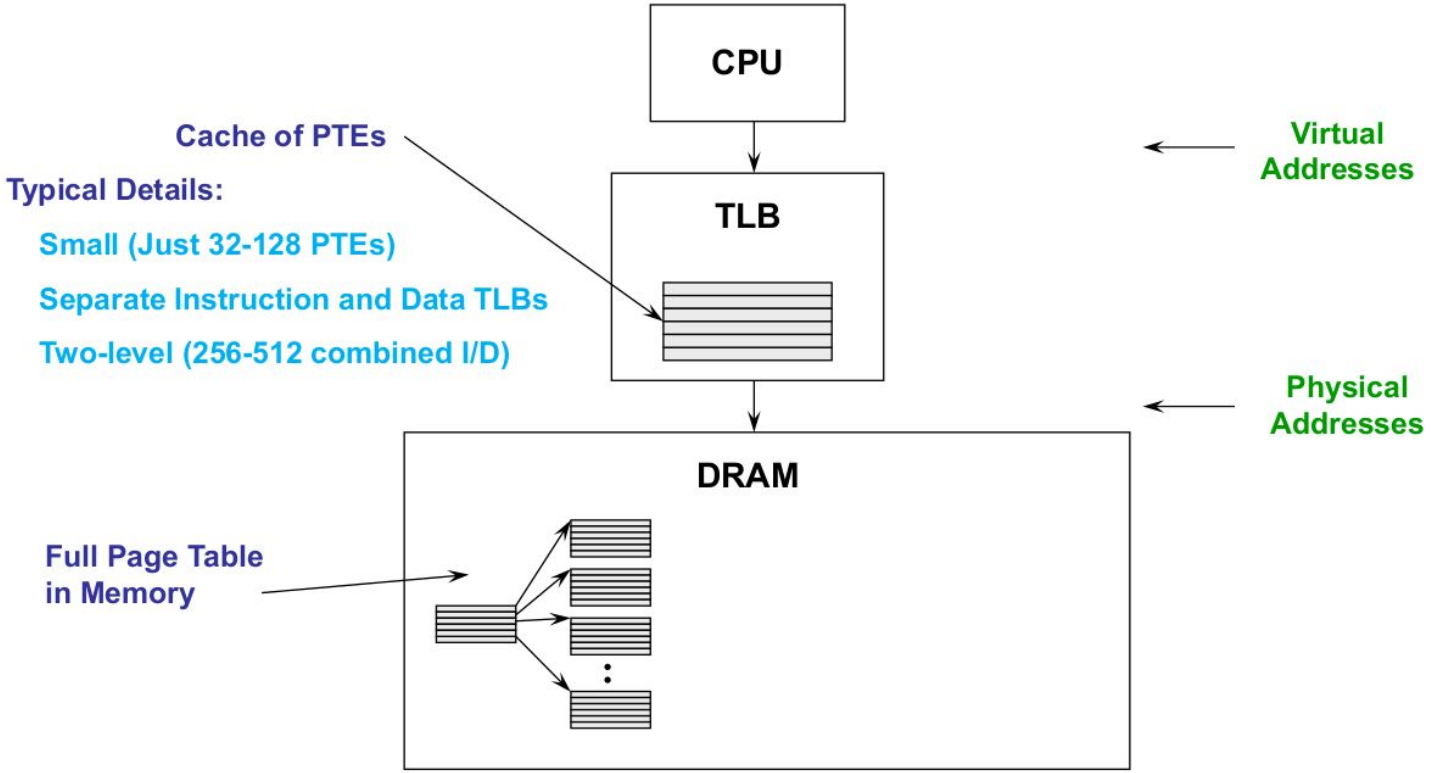
Efficient Translations

- Our original page table already doubled the cost of memory access
  - One lookup into the page table, another to fetch the data
- Now multi-level page tables increase the cost too much!
  - Several lookups into the page tables, then to fetch the data
- 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)

TLB

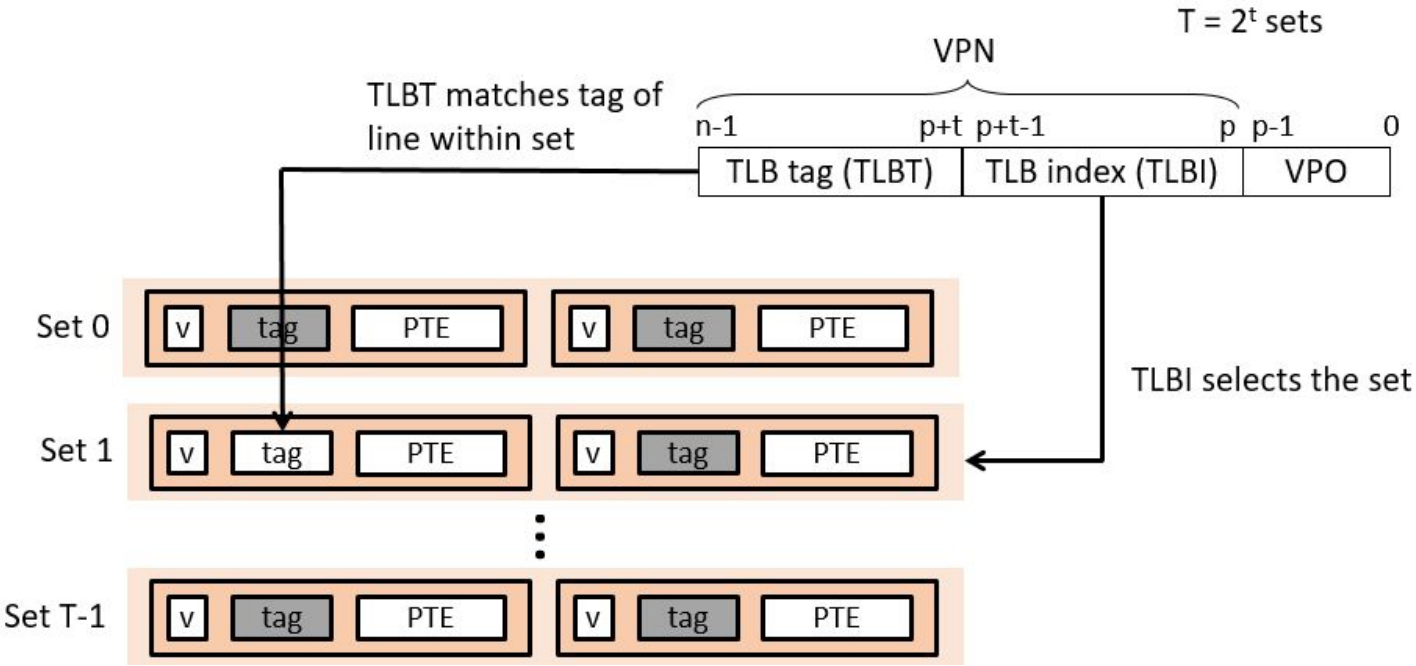
- Translation Lookaside Buffer (TLB)
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page frames
- TLBs exploit locality
  - Processes only use a handful of pages at a time (its working set)
- Some architectures have multiple TLBs:
  - Instruction TLB (ITLB)
  - Data TLB (DTLB)
- Can even have multiple levels of TLBs
  - Smaller ones are faster

TLB & Page Table



TLB in More Detail

- MMU uses the VPN portion of the virtual address to access the TLB



Managing TLBs

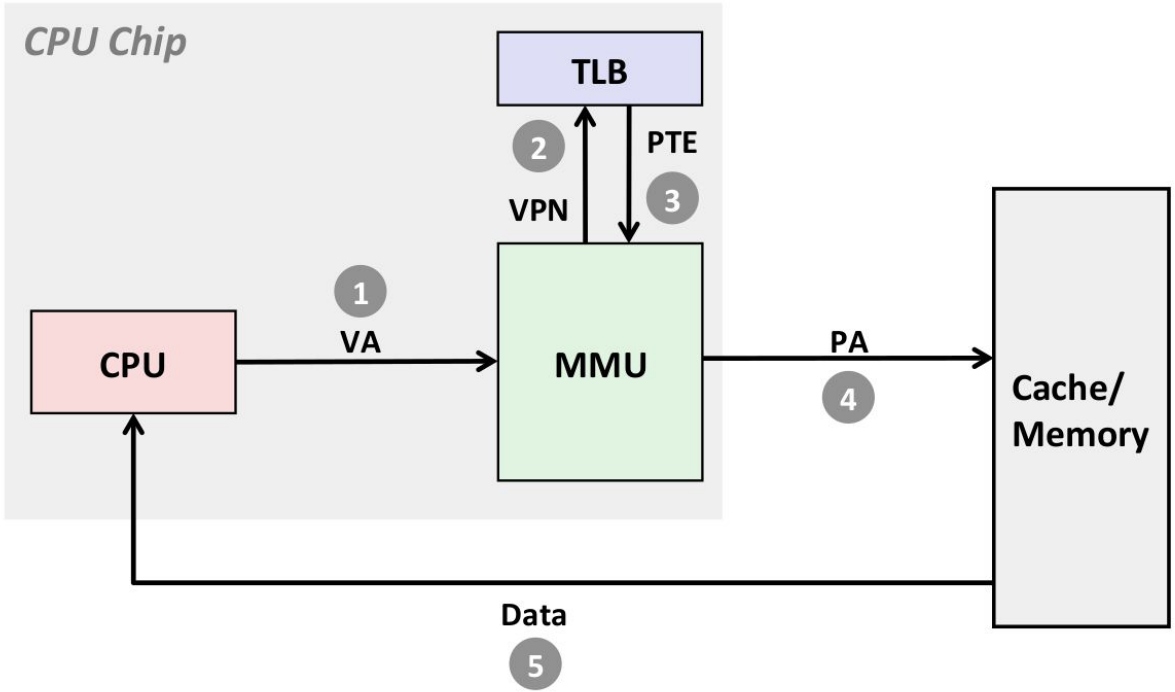
- Address translations for most memory accesses are handled using the TLB
  - But there are misses (TLB miss) ...
- Who places translations into the TLB (loads the TLB)?
  - MMU, e.g., x86
    - Knows where page tables are in main memory
    - OS maintains tables, HW accesses them directly
    - Tables have to be in HW-defined format (inflexible)
  - Software loaded TLB (OS), e.g., 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 (Cont.)

- OS ensures that TLB and page tables are consistent
  - When protection bits of a PTE are changed, it needs to invalidate the PTE if it is in the TLB
- When a context switch happens, the TLB entries are often invalid
  - On the x86, the kernel can flush the entire TLB or individual entries
  - Which is better depends; the kernel can’t always know ahead of time
- 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., LRU)

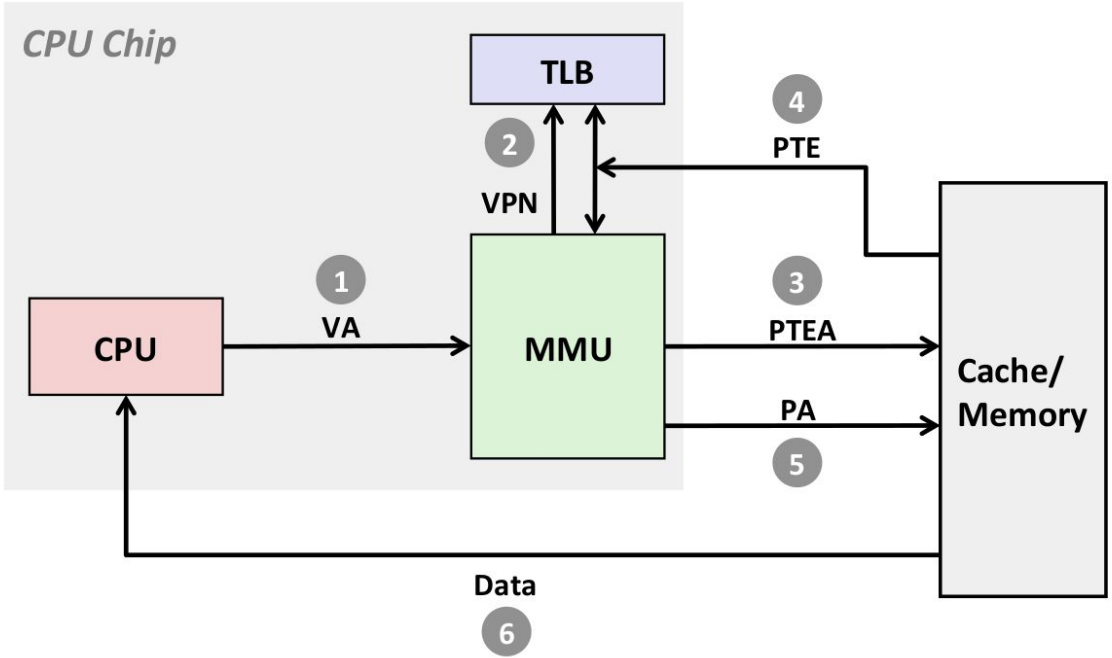
TLB Hit

- A TLB hit eliminates accesses to the page table



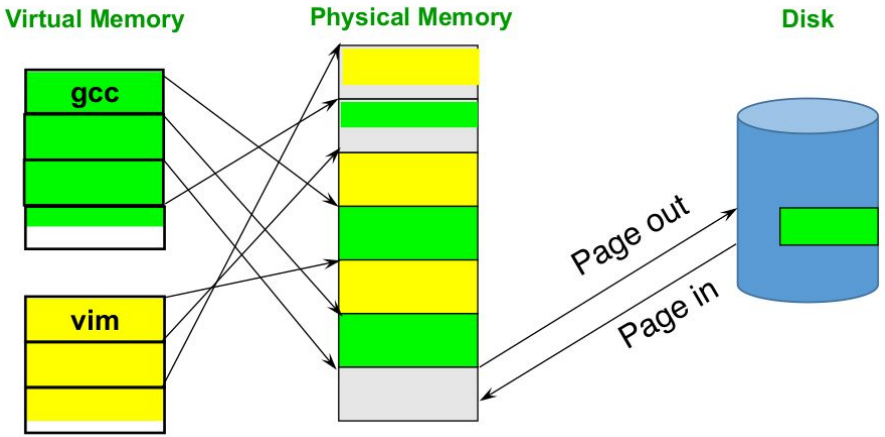
TLB Miss

- A TLB miss incurs additional memory accesses (the PTEs)
  - Fortunately, TLB misses are rare, why?



Paged Virtual Memory

- Pages can be moved between memory and disk
  - Use disk to simulate larger virtual than physical mem
  - This process is called 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 (where? the swap file/backing store)
  - Done by the OS, and transparent to the application



Demand Paging

- Pages are only brought into memory when a reference is made to a location on that page
  - In other words, the page isn't resident until it's accessed (read from or written to)
- Demand paging is really good!
  - The OS avoids wasting time/space loading parts of your program that are never referenced
  - Example: if you never make it past level 1 in a video game, the OS can avoid loading up the code for level 2

Next Lecture

Page faults and replacement policies