CSE 351 Section 10

Virtual Memory + final review

Virtual Memory

Virtual Memory

Very powerful layer of indirection on top of physical memory addressing

- We never actually use physical addresses when writing programs
- Every address, pointer, etc. you come across is a virtual address

Why? There are several benefits:

- Shared memory spaces
- Memory isolation
- Memory R/W/X protection
- Illusion of large address space despite limited physical memory

Address Translation

How do we convert virtual addresses to physical addresses?

- Create a table mapping virtual pages to physical pages
- A page table is an array of page table entries (PTEs)

An entry exists for every virtual page number (VPN)

- Each entry stores a physical page number (PPN)
- Each process gets its own pages table
- The page table is software-defined! It exists in DRAM

Quick note about page tables

- If we have 4 KB pages and 32-bit addresses, then…
  - We have 2^20 PTEs per page table–that's ~4 MB per page table
  - Pull up task manager–we have hundreds of processes running on a computer. That's multiple GBs of memory used, just for page tables
    - Keep in mind we have 64-bit addresses now as well..!
  - Solution in real-world operating systems: page the page table (and potentially page those paged page tables, etc…)

Visualizing virtual memory

Virtual memory is an enormous, contiguous region for storage

- It is broken up into fixed-size “pages”
- When a process needs more memory, the OS will allocate a page of physical memory and insert an entry into that process’ page table
- If we run out of physical memory for that process, then only the most recently-used pages are left in memory
    - The rest are kept on disk, and “swapped” in when needed

Visualizing virtual memory

Thus, we can think of virtual memory as a cache for disk

- Imagine that the virtual memory space is initially mapped to disk
- Recently-used pages of virtual memory are cached in physical memory

Page Faults

- When a page table entry points to a page that is on disk, a process will generate a page fault
  - Transfers control to the OS
  - The OS will copy the page of data from disk into memory

Acronyms for Reference (there's a lot oof)

VA	Virtual Address
VPN	Virtual Page Number
VPO	Virtual Page Offset
PT	Page Table
PTBR	Page Table Base Register
TLBT	TLB Tag
MMU	Memory Management Unit

PA	Physical Address
PPN	Physical Page Number
PPO	Physical Page Offset
PTE	Page Table Entry
TLB	Translation Lookaside Buffer
TLBI	TLB Index

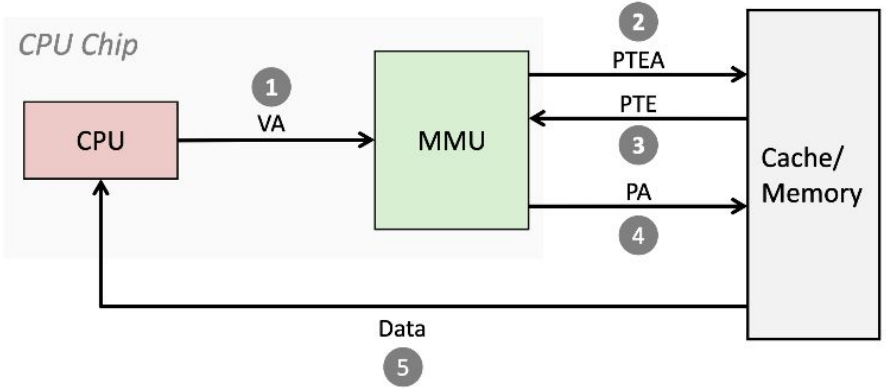
Address Translation: Page Hit

1) Processor send *virtual* address to MMU

2-3) MMU fetches PTE from page table in cache/memory (use PTBR to find beginning of page table for current process)

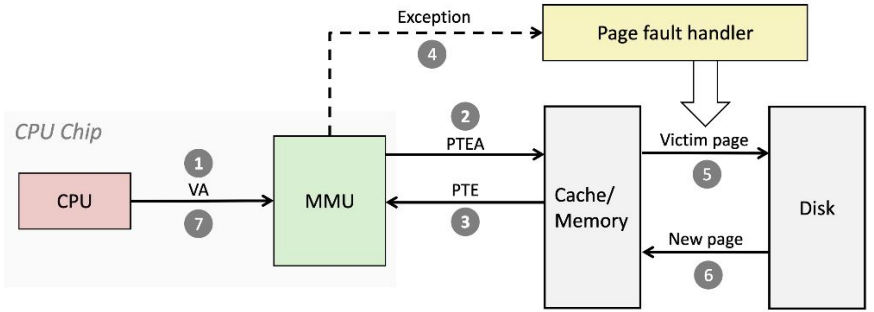
4) MMU sends *physical* address to cache/memory requesting data

5) Cache/memory sends data (~1 word) to processor



Address Translation: Page Fault

1. Processor send virtual address to MMU
2. MMU fetches PTE from page table in cache/memory
3. Valid bit is 0, so MMU triggers page fault exception
4. Handler identifies victim (and, if dirty, pages it out to disk)
5. Handler pages in new page and updates PTE in memory
6. Handler returns to original process, restarting faulting instruction



Other Benefits

Shared memory is easy to implement

- In each process' page table, simply point to the same PPN (physical page number)

Memory protection

- In addition to valid bits in the page table, also store bits for R/W/X
- If a process attempts to use a page incorrectly, it will generate a segmentation fault

Speeding up virtual memory

Reading memory in order to read memory seems… slow

- It is!

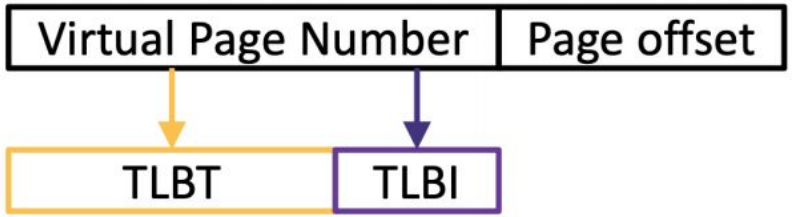
How have we sped up memory reads already?

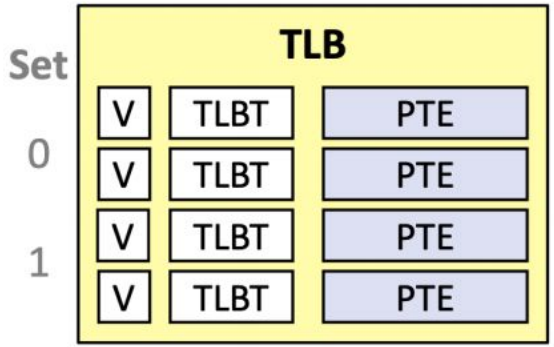
- Caching!

We use a *Translation Lookaside Buffer* to speed up virtual memory access

Address Translation: Translation Lookaside Buffer

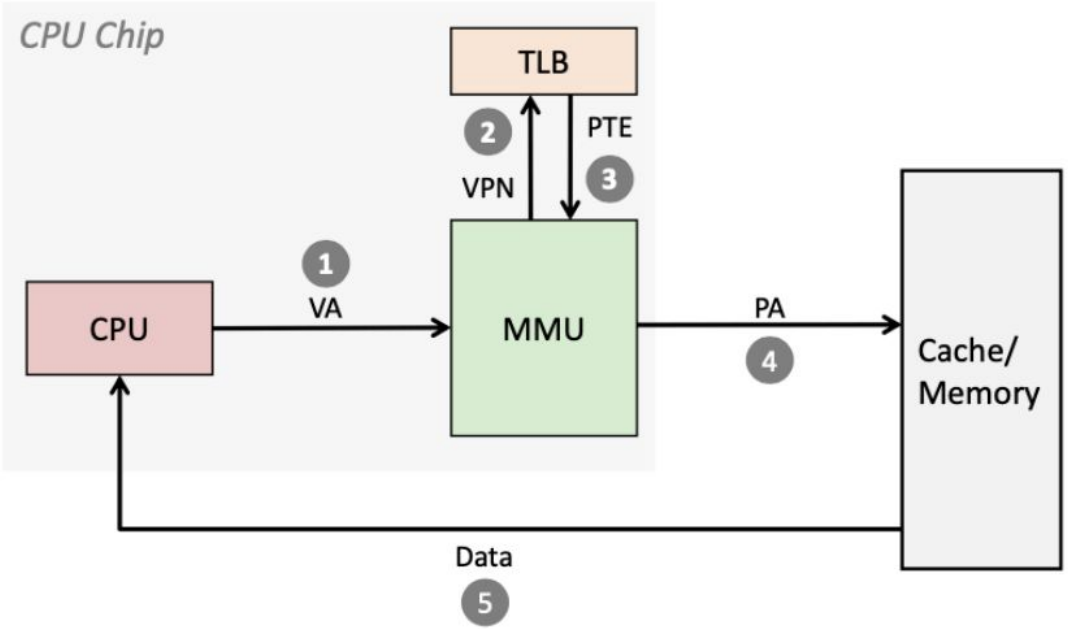
- Hardware cache that stores recent VPN → PPN mappings
- Eliminate one memory access that looks up page table in memory on TLB Hit
  - Now MMU only queries DRAM for TLB misses
- TLB misses are rare!





TLB Hit

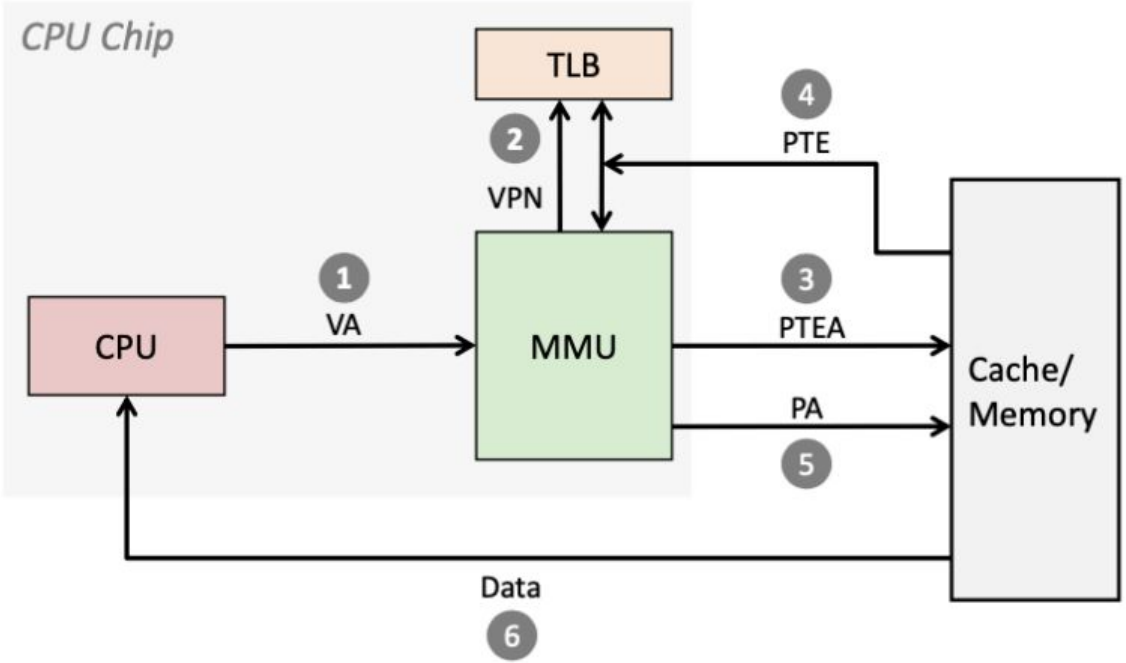
1. Processor sens virtual address to MMU
2. MMU uses VPN to find PTE in TLB
3. MMU fetches PTE from TLB
4. MMU sends physical address to cache/memory requesting data



A TLB hit eliminates a memory access

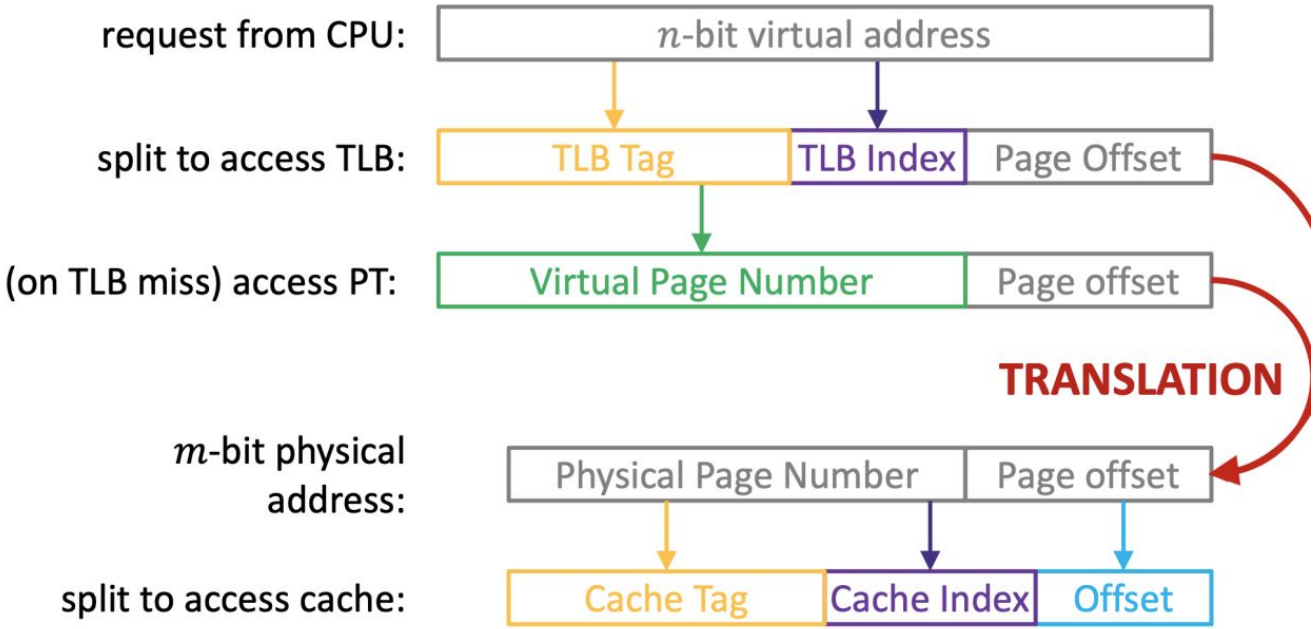
TLB Miss

1. Processor sends virtual address to MMU
2. MMU checks TLB and notices a TLB miss
3. MMU fetches PTE from page table in cache/memory
4. PTE is loaded into TLB
5. MMU sends physical address to cache/memory requesting data



A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare

Address Manipulation



Formulas for Reference

n-bit virtual address, virtual address space *N* = 2^*n*
m-bit physical address, physical address space *M* = 2^*m*
Page size *P* = 2^*p* , page offset is *p* bits

Number of virtual pages is *N* / *P*, so VPN width = *n* - *p*
Number of physical pages is *M* / *P*, so PPN width = *m* - *p*

VPN splits into TLBT and TLBI:

- *S* = number of TLB sets
- TLBI bits: log_2(*S*)
- TLBT bits: VPN bits - TLBI bits

Exercise 3 (not on worksheet)

Page offset bits = log2(_____)

Virtual address bits = _____+ page offset bits

Physical address bits = physical page number bits + _____

Virtual page number bits = log2(_____)

Entries in a page table = _____

Final Review

What do y’all want practice on?