Administivia

- Project 2 ongoing
  - Extended, now due Friday 5/19
- Homework #3 ongoing
  - Due Tuesday 5/23
Paging with Single-Level Page Tables

- Divide physical and virtual memory into fixed-sized chunks calls **pages**
- Translate from virtual to physical addresses using a page table

![Diagram showing linear page table and page table entries mapping to physical memory](image)
Multi-Level Page Tables

- Represent page tables hierarchically
- Each page map fits in one page
- Omit empty page maps
- Reduces the amount of memory needed to store page tables
Today’s Outline

- Efficient translations
  - TLBs
- The illusion of more memory than is physically present
  - Demand-paged virtual memory
- Advanced functionality
  - Shared memory
  - Copy on write
Inefficient Translations

• Programs must translate every virtual address to a physical address
  ♦ On every load or store operation
• How many memory accesses are required per program memory access?
  ♦ Linear page table
    » 2: page table + data itself
  ♦ N-level page table
    » N + 1: N levels of page maps + data itself
Translation Lookaside Buffer (TLB)

• Solution: take advantage of locality
  ♦ In a short period of time, programs tend to access the same page(s) repeatedly

• Translation Lookaside Buffer
  ♦ A small hardware cache of recently used translations
  ♦ Each cache entry stores a virtual page number and the corresponding PTE

• Implementation
  ♦ In hardware in the MMU
  ♦ Fully associative cache
  ♦ Typically 64-2048 entries (small!)
  ♦ TLB hits are very fast (~1 CPU cycle)
Address Translation with a TLB

- Compare VPN with every VPN in the TLB
  - Execute all comparisons in parallel
- If there’s a hit, use the corresponding PFN
- If there’s a miss, consult the page table
  - Look up the PTE
  - Save the PTE in the TLB
Most address translations are **TLB hits**, handled by the TLB (> 99%)

- **TLB misses** do still occur
  - Need to walk the page table and update the TLB
Managing TLBs – Handling Misses

• What component handles TLB misses?
  ♦ **Hardware** (MMU) [x86, ARM, RISC-V]
    » Knows where page tables are in main memory
    » HW parses page tables and loads PTE into TLB
    » Page tables have to match a HW-defined format (inflexible)
  ♦ **Software** (OS) [MIPS, Alpha, Sparc, PowerPC]
    » TLB faults to the OS, OS finds the PTE and loads it into the TLB
    » CPU ISA has instructions for manipulating the TLB
    » Tables can be in any format (flexible)

• Policy for replacing TLB entries
  ♦ Random, Least Recently Used
Managing TLBs

• Handling context switches
  ♦ Invalidate all entries
    » Lots of TLB misses afterward
  ♦ Tag each entry with an Address Space Identifier (ASID)
    » Check each TLB entry against a register containing the process id of the currently executing process
    » Update this register on a context switch

• Maintaining consistency between the TLBs and page tables
  ♦ When the OS modifies a PTE (e.g., changes protection bits) it needs to invalidate the PTE if it is in the TLB
  ♦ On multi-core CPUs, must invalidate PTE entries on all cores (TLB shoot-down)
Today’s Outline

• Efficient translations
  ♦ TLBs

• The illusion of more memory than is physically present
  ♦ Demand-paged virtual memory

• Advanced functionality
  ♦ Shared memory
  ♦ Copy on write
Limited Space in Physical Memory

- What if all of our data doesn’t fit in memory at once?
- DRAM
  - Low latency
  - High bandwidth
  - Limited capacity (tens of GB)
- Disk
  - Much larger capacity (a few TB)
  - 100,000x higher latency
  - 1,000x less bandwidth
- Goal: use disk to make physical memory appear larger than it is
Demand Paging

- Store data on disk and use main memory as a cache
  - Take advantage of locality – programs typically spend most of their time using a small fraction of their data
  - Keep data in physical memory while it is used
  - Keep unused data on disk in a swap file
    » Also called a backing store, swap space, etc.
- Demand paging: load pages on demand
  - Each virtual page is either in memory or on disk
  - Allocate memory when first accessed
  - Swap pages in from disk when they are accessed (if not present)
- Ideal: memory system behaves as though it has the performance of main memory but the cost and capacity of disk
Paged Virtual Memory

Virtual Memory

Page 0
Page 1
Page 2
Page 3
Page 4
...
Page N-1

Physical Memory

Disk
Page Faults - Overview

• What happens when a process references a page that is in the swap file?
  ♦ Process references memory (e.g., LDR or STR)
  ♦ Valid/present bit in PTE is set to 0
    » Indicates that the page is not accessible
  ♦ Triggers an exception (page fault) and a trap to the OS
  ♦ OS runs the page fault handler
    » Finds a free page frame in physical memory
    » Reads the page in from the swap file to the page frame
    » OS updates the PTE, sets valid=1
    » Returns to the process
  ♦ Process re-executes the memory reference
Page Faults - Details

- How does the OS know which page caused the page fault?
  - Hardware saves the faulting address in a special register
- What if there are no available page frames?
  - Evict a different page to memory
- How does the OS know where the page is located in the swap file?
  - Store this information in the PTE (it’s invalid anyway)
- Where should we resume process execution?
  - Restart the faulting instruction
Paging Policies

• **Page fetching**: when to bring pages into memory
  - Demand fetching – don’t load a page until it is referenced
  - Prefetching – try to predict when pages will be needed and load them in advance
    » Simple approach: during a page fault, bring in the next N pages

• **Page replacement**: which pages to evict from memory
  - Next lecture
Address Translation Redux

- **virtual address**
- **VPN**
- **offset**

Is VPN in the TLB?

- yes
- no

Do bits in the PTE allow access?

- yes
- no

- Find the PTE in the page table, reload TLB
- Page fault, fetch page, reload TLB
- Combine PFN from PTE with offset

PFN

offset

physical address

hardware or software
Causes of Page Faults

- PTE can indicate the type of a page fault:
  - **Invalid**
    - Page not in physical memory (in swap file)
    - Virtual page allocated by program but not yet accessed
    - Virtual page not allocated (invalid memory access)
  - **Read/write/execute** – operation not permitted on page
- OS approaches to handling page faults:
  - Fetch page from the swap file
  - Allocate a physical page
  - Send a fault back to process (e.g., segmentation fault)
  - Other advanced functionality
Today’s Outline

- Efficient translations
  - TLBs
- The illusion of more memory than is physically present
  - Demand-paged virtual memory
- Advanced functionality
  - Shared memory
  - Copy on write
Sharing Memory

- By default, processes have private virtual address spaces
  - Disjoint sets of physical pages
- Sometimes processes want to share memory
  - Cache shared by a parent and child in a forking web server
  - Tabs in your browser
Use **shared memory** to allow multiple processes to access the same memory

- Use a system call to set up shared memory
  - Unix: `shm_open`, `shm_unlink`

- Have PTEs in both page tables map to the same physical frame

- Can map shared memory at the same or at different virtual addresses in each process address space
Copy on Write

- OSes spend a lot of time copying data
  - E.g., entire address spaces to implement fork()
- We would like to avoid copying until it’s necessary
- Copy-on-Write
  - Share pages as read-only, using shared mappings
  - Only copy when a process tries to write the page
    - Protection fault, trap to the OS, copy page, update page mapping, restart write instruction
Today’s Summary

- Efficient translations
  - TLBs – cache PTEs
- The illusion of more memory than is physically present
  - Demand-paged virtual memory
- Advanced functionality
  - Shared memory
  - Copy on write
For next class…

• Read chapters 17 and 22