CS61C : Machine Structures

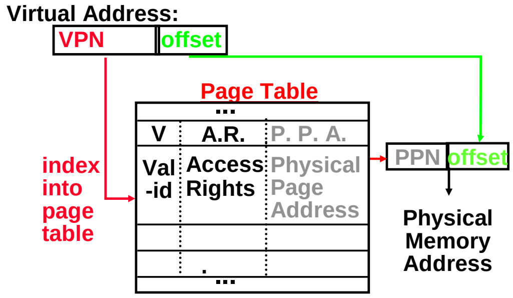
Lecture 7.1.2
VM II

2004-08-03

Kurt Meinz

inst.eecs.berkeley.edu/~cs61c

Address Mapping: Page Table



Page Table located in physical memory

Page Table

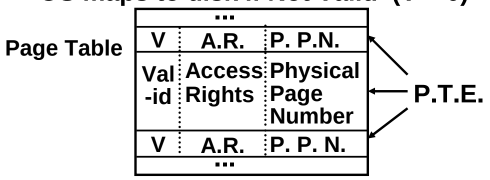
- A page table: mapping function
  - There are several different ways, all up to the operating system, to keep this data around.
  - Each process running in the operating system has its own page table
    - Historically, OS changes page tables by changing contents of Page Table Base Register
      - Not anymore! We'll explain soon.

Requirements revisited

- Remember the motivation for VM:
- Sharing memory with protection
  - Different physical pages can be allocated to different processes (sharing)
  - A process can only touch pages in its own page table (protection)
- Separate address spaces
  - Since programs work only with virtual addresses, different programs can have different data/code at the same address!

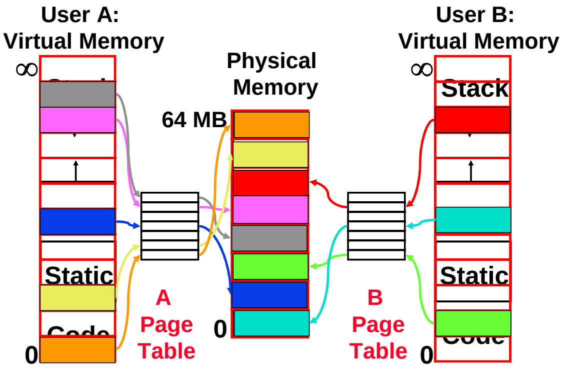
Page Table Entry (PTE) Format

- Contains either Physical Page Number or indication not in Main Memory
- OS maps to disk if Not Valid (V = 0)



- If valid, also check if have permission to use page: Access Rights (A.R.) may be Read Only, Read/Write, Executable

Paging/Virtual Memory Multiple Processes



Comparing the 2 levels of hierarchy

Cache Version	Virtual Memory vers.
Block or Line	Page
Miss	Page Fault
Block Size: 32-64B	Page Size: 4K-8KB
Placement: Direct Mapped, N-way Set Associative	Fully Associative
Replacement: LRU or Random	Least Recently Used (LRU)
Write Thru or Back	Write Back

Notes on Page Table

- OS must reserve "Swap Space" on disk for each process
- To grow a process, ask Operating System
  - If unused pages, OS uses them first
  - If not, OS swaps some old pages to disk
  - (Least Recently Used to pick pages to swap)
- Will add details, but Page Table is essence of Virtual Memory

VM Problems and Solutions

- TLB
- Paged Page Tables

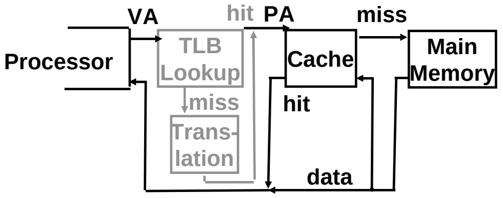
Virtual Memory Problem #1

- Map every address ⇒ 1 indirection via Page Table in memory per virtual address ⇒ 1 virtual memory accesses = 2 physical memory accesses ⇒ SLOW!
- Observation: since locality in pages of data, there must be locality in virtual address translations of those pages
- Since small is fast, why not use a small cache of virtual to physical address translations to make translation fast?
- For historical reasons, cache is called a Translation Lookaside Buffer, or TLB

Translation Look-Aside Buffers (TLBs)

•TLBs usually small, typically 32 - 256 entries

• Like any other cache, the TLB can be direct mapped, set associative, or fully associative



On TLB miss, get page table entry from main memory

Typical TLB Format

Virtual Address	Physical Address	Dirty	Ref	Valid	Access Rights

• TLB just a cache on the page table mappings

• TLB access time comparable to cache (much less than main memory access time)

• Dirty: since use write back, need to know whether or not to write page to disk when replaced

•Ref: Used to help calculate LRU on replacement
  • Cleared by OS periodically, then checked to see if page was referenced

What if not in TLB?

- Option 1: Hardware checks page table and loads new Page Table Entry into TLB
- Option 2: Hardware traps to OS, up to OS to decide what to do
- MIPS follows Option 2: Hardware knows nothing about page table

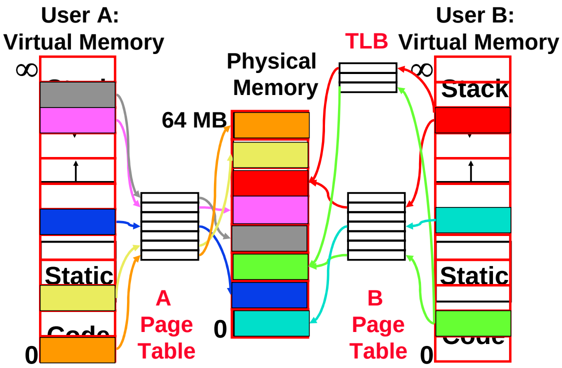
What if the data is on disk?

- We load the page off the disk into a free block of memory, using a DMA (Direct Memory Access – very fast!) transfer
  - Meantime we switch to some other process waiting to be run
- When the DMA is complete, we get an interrupt and update the process's page table
  - So when we switch back to the task, the desired data will be in memory

What if we don't have enough memory?

- We chose some other page belonging to a program and transfer it onto the disk if it is dirty
  - If clean (disk copy is up-to-date), just overwrite that data in memory
  - We chose the page to evict based on replacement policy (e.g., LRU)
- And update that program's page table to reflect the fact that its memory moved somewhere else
- If continuously swap between disk and memory, called Thrashing

Paging/Virtual Memory Review



Virtual Memory Problem #1 Recap

- Slow:
  - Every memory access requires:
    - 1 access to PT to get VPN->PPN translation
    - 1 access to MEM to get data at PA
- Solution:
  - Cache the Page Table
    - Make common case fast
    - PT cache called “TLB”
  - “block size” is just 1 VPN->PN mapping
  - TLB associativity?

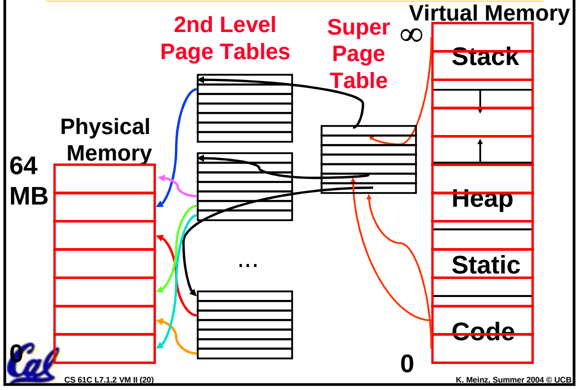
Virtual Memory Problem #2

- Page Table too big!
  - 4GB Virtual Memory ÷ 1 KB page
    ⇒ ~ 4 million Page Table Entries
    ⇒ 16 MB just for Page Table for 1 process, 8 processes ⇒ 256 MB for Page Tables!
- Spatial Locality to the rescue
  - Each page is 4 KB, lots of nearby references
  - But large page size wastes resources
- No matter how big program is, at any time only accessing a few pages
  - “Working Set”: recently used pages

Solutions

- Page the Page Table itself!
  - Works, but must be careful with never-ending page faults
  - Pin some PT pages to memory
- 2-level page table
- Solutions tradeoff in-memory PT size for slower TLB miss
  - Make TLB large enough, highly associative so rarely miss on address translation
  - CS 162 will go over more options and in greater depth

2-level Page Table



Page Table Shrink :

- Single Page Table

Page Number	Offset
20 bits	12 bits

- Multilevel Page Table

Super Page No.	Page Number	Offset
10 bits	10 bits	12 bits

- Only have second level page table for valid entries of super level page table
  - Exercise 7.35 explores exact space savings