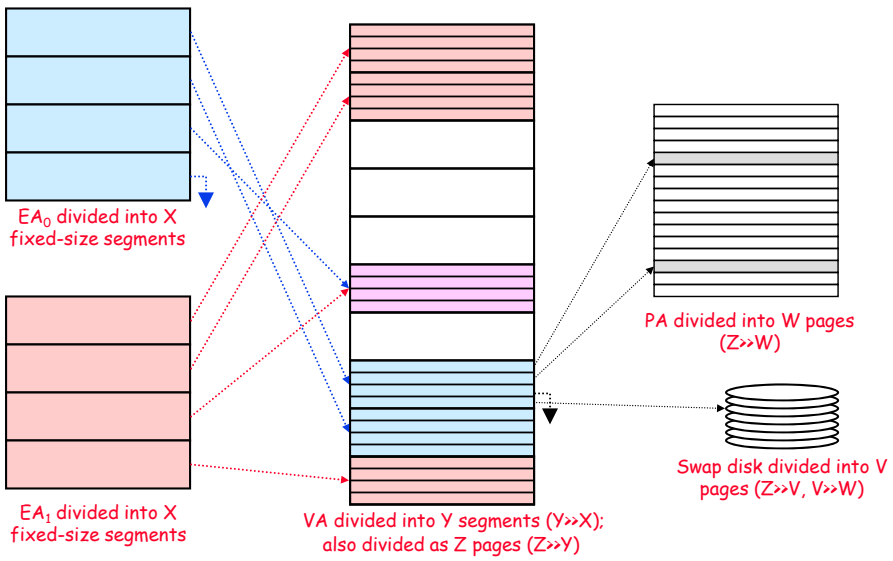
18-447 Lecture 21: Virtual Memory: Page Tables and TLBs

James C. Hoe
Dept of ECE, CMU
April 13, 2009

Announcements: Read Jacob&Mudge for Wed

Handouts: Don't forget Proj 4 and HW 4

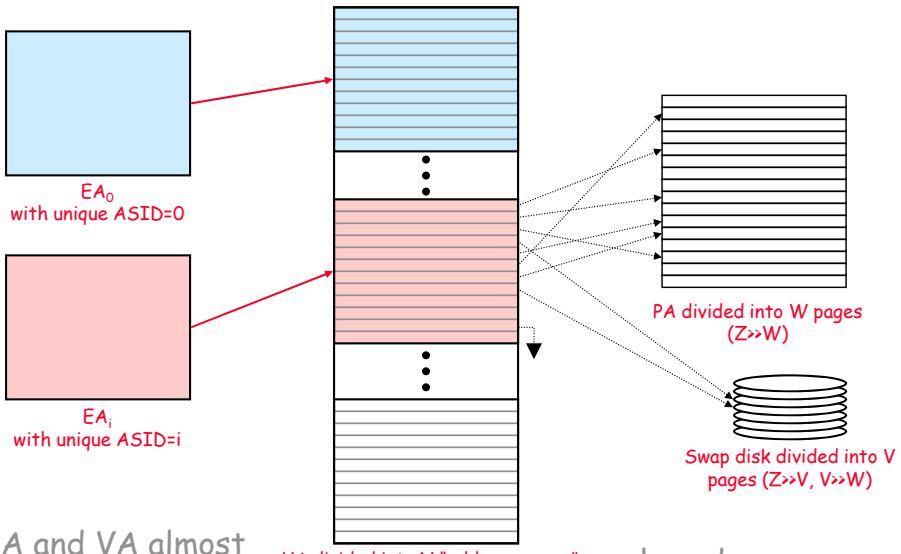
EA, VA and PA (IBM's view)



segmented EA: private, contiguous + sharing

demand paged VA: size of swap, speed of DRAM

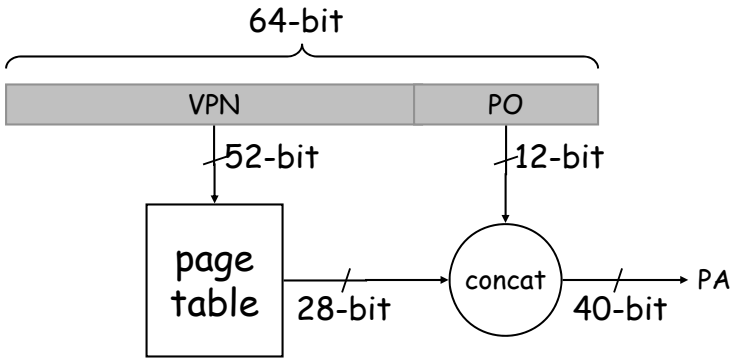
EA, VA and PA (almost everyone else)



EA and VA almost synonymous

how do processes share pages?

How large is the page table?



- A page table holds mapping from VPN to PPN
- Suppose 64-bit VA and 40-bit PA, how large is the page table? 2^{52} entries x ~4 bytes ≈ 16×10^{15} Bytes

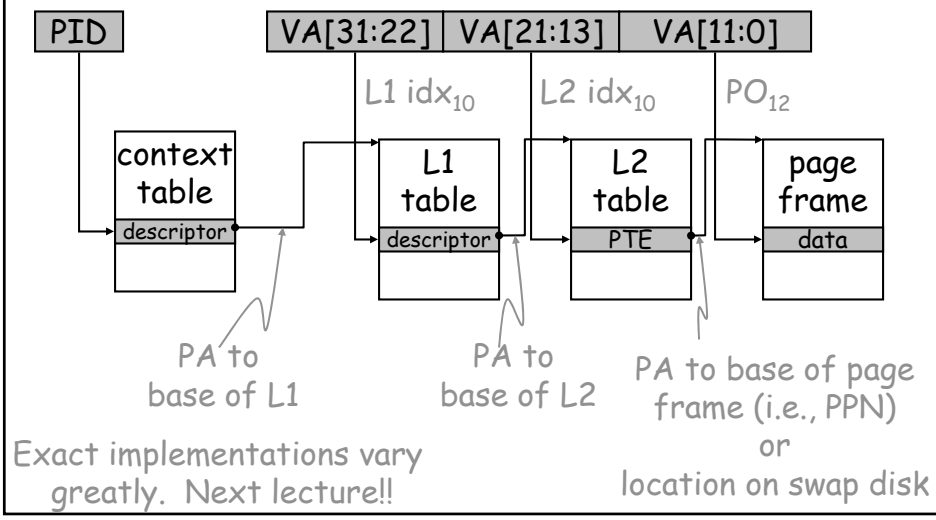
and that is for just one process!!?

How large is the page table?

- Don't need to keep track of the entire VA space
  - the total allocated VA space in a system is 2^{64} bytes x # processes, but most of which is not alive
  - the system can't possibly use more memory locations than the physical storage (DRAM and swap disk)
- A clever page table scales "linearly" with the size of physical storage (and not the size of the VA space)
- Also cannot be too convoluted
  - a page table must be "walkable" by HW
  - a page table is accessed not infrequently
- Two basic themes in use today
  - hierarchical page tables
  - hashed (inverted) page tables

Hierarchical Page Tables

- Hierarchical page table is a "tree" data structure in DRAM



Exact implementations vary greatly. Next lecture!!

Hierarchical Page Tables

- Hierarchical page table is a "tree" graph,
  - for example on previous page
    - L1 table has 1024 decedents (L2 tables) indexed by VA[31:22]
    - each L2 table has 1024 decedents (physical page frames) indexed by VA[21:12]
  - more levels can be used to accommodate larger VA space
  - assume 4-byte descriptors and PTEs, each table is 4KByte (size of page frames) such that the tables themselves can be demand paged between DRAM and disk
- Hierarchical page table is a "sparse" tree graph
  - if none of the virtual page frames associated with a L2 table is in used, the L2 table does not need to exist (corresponding L1 entry simply points to null)
  - in general, an entire unused sub-tree can avoided
  - considering typical size ratio of VA to PA, the tree should be quite sparse

How sparse?

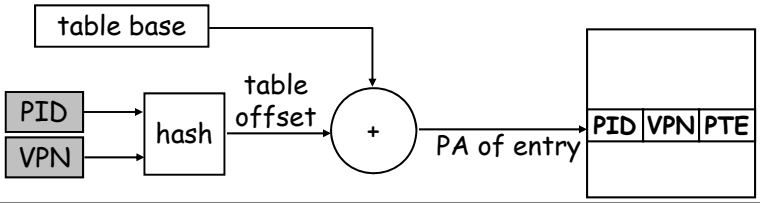
How large is the hierarchical table?

- Assume 32-bit VA with 4 MByte in use
- Best Case: one contiguous 4-MByte region in VA aligned on 4MByte boundaries
  - 1K physical page frames
  - needs 1 L2 table + 1 L1 table=2 x 4KBytes,
  - overhead ≈ sizeof(PTE)/page_size per physical page
- Worst Case: 1K 4-KByte regions in VA; each is 4MByte aligned
  - 1K physical page frames
  - needs 1K L2 tables (only 1 entry per L2 table in use)
  - 1025 x 4KBytes
  - overhead ≈ 200% per physical page
- Locality says we should be close to the best case

4 bytes/4Kbytes ≈ 0.1%

Hashed Page Tables

- Choose an appropriate page table storage overhead
  - at least 1 entry per physical page, but probably more to avoid "hash" conflicts
  - e.g. 1GB DRAM ⇒ 256K frames ⇒ 256K PTEs
- Page table works like a hash table
  - to lookup a translation, hash VPN and PID into a index e.g. (VPN⊕PID)%table_size (note: overly simplified)
  - assumes the PTE was inserted according to the same hash
  - each entry must be "tagged" by PID and VPN to detect collision



How large is the hashed page table?

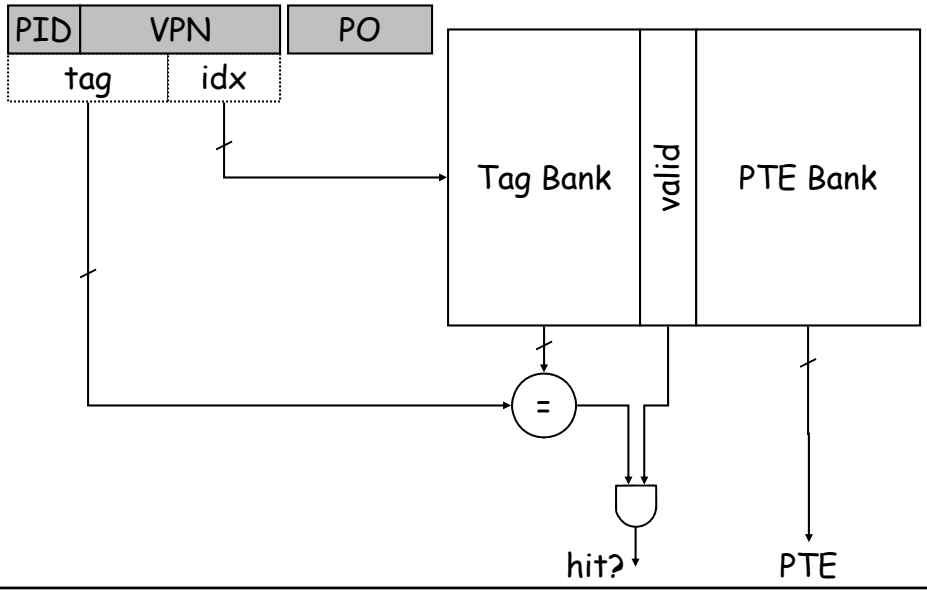
- Size of hashed page table is a function of physical memory size
- The exact proportion is an engineering choice

  large enough to reduce hash collisions
- Often hashed page table only stores translation for pages currently in DRAM; on a miss, must consult a complete table structure to determine if the VPN is on swap disk or if the VPN is non-existent
- The original "inverted" page table (a historical note)
  - allocate exactly 1 entry per physical page frame
  - hashed location in table corresponds exactly to page frame in main memory (the table entries do not need to hold PPN)
  - viewing the table by itself, it is indexed by PPN and returns VPN

Translation Look-Aside Buffer (TLB)

- Every user memory reference (code or data) requires a translation
  - how many memory accesses per translation?
    hierarchical vs. hashed
  - what good is it to hit in the cache if translation takes forever
- TLB: a "cache" of most recently used translations
  - same type of "tagged" lookup structure as caches and BTBs
  - given a VPN, returns a PTE (PPN & protections)
  - TLB entry:
    tag: address tag (from VA), PID
    PTE: PPN, protection bits
    misc: valid, dirty, etc.
  - similar design considerations as caches
    capacity, block size, associativity, replacement policy

Direct-Mapped TLB (bad example)



TLB Design

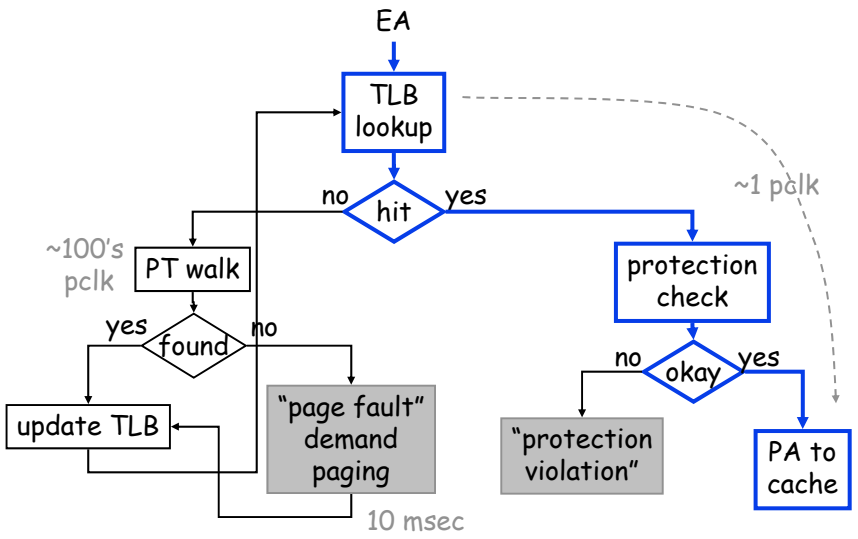
- Separate I and D-TLB, multi-level TLBs make sense as in caches
- C: if the L1 I-cache is 64KB, what's the I-TLB size?
  - should cover the same 64KB footprint
  - a minimum of 16 TLB entries × some safety factor (2~8)
  - in the old days 32~64 entries; nowadays a few hundred
- B: after accessing a page, how likely is it to access the next page? (coarse grain spatial locality)
  - typically one PTE per TLB entry
  - MIPS stores 2 consecutive pages' translations per entry
- a: what associativity to minimize collision?
  - in the old days, fully-associative is the norm
  - nowadays, 2~4-way-associative is more common Why?

On a TLB Miss

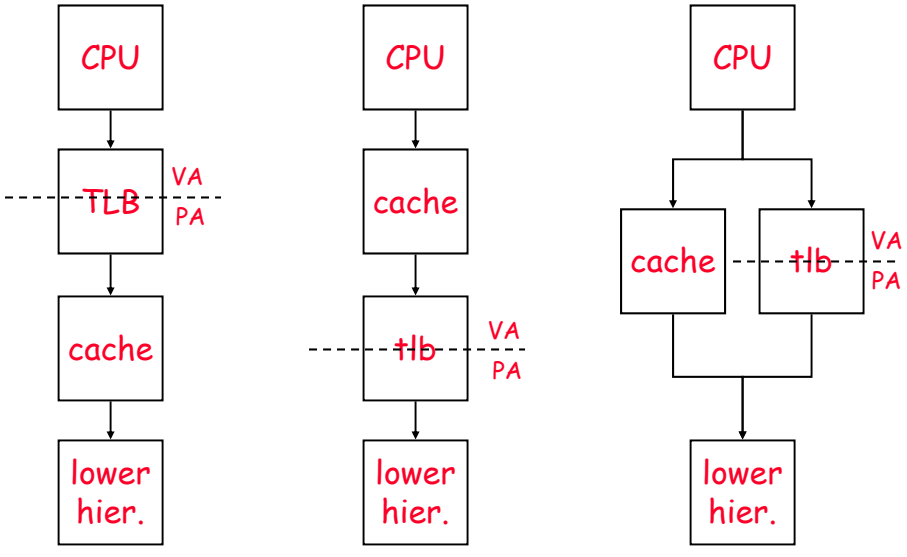
- Most address translation resolved in ~1 cycle in the TLB
- On a TLB miss
  - must "walk" the page table to determine translation
  - walk usually done by HW (MIPS walks in SW)
  - can take 100's of cycles to complete
  - if PTE is found and page is in memory, then replace TLB with new PTE and continue
  - if PTE is found but the page is on disk, then trigger "page fault" exception to initiate kernel handler for demand paging
  - if PTE is not found, trigger "segmentation fault" exception to initiate kernel handler

    What to do now?

VA to PA Translation



How should VM and Caches Interact?



Virtual Caches

- Even with TLB, translation takes time
- Naively, memory access time in the best case is

  TLB hit time + cache hit time
- Why not access cache with virtual addresses and only translate on a cache miss to DRAM

  make sense if TLB hit time >> cache hit time
- Virtual caches in SUN SPARC, circa 1990
  - CPU has gotten fast enough that off-chip a SRAM access takes multiple cycles
  - dies size has gotten large enough to integrate L1 caches
  - MMU and TLB still on a separate chip

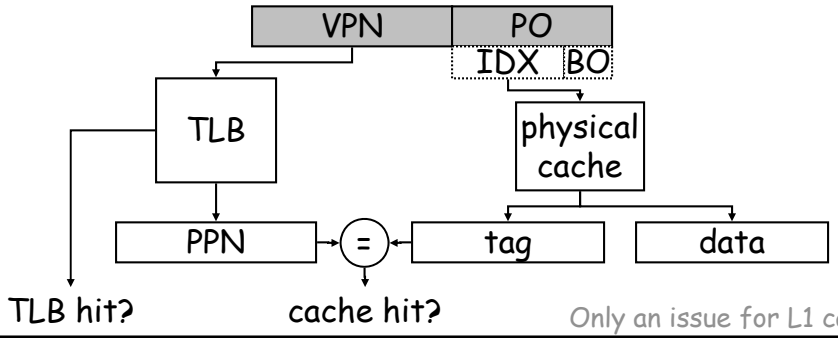
  the conditions no longer hold

Managing Virtual Caches: Synonyms and Homonyms

- Homonyms (same sound different meaning)
  - same EA (in different processes) points to different PAs
  - flush virtual cache between context; or include PID in cache tag
- Synonyms (different sound same meaning)
  - different EAs (from the same or different processes) point to the same PA
  - in a virtually addressed cache
    - a PA could be cached twice under different EAs
    - updates to one cached copy would not be reflected in the other cached copy
    - solution: make sure synonyms can't co-exist in the cache, e.g., OS can forces synonyms to have the same index bits in a direct mapped cache

Virtually-Indexed Physically-Tagged
(a misnomer)

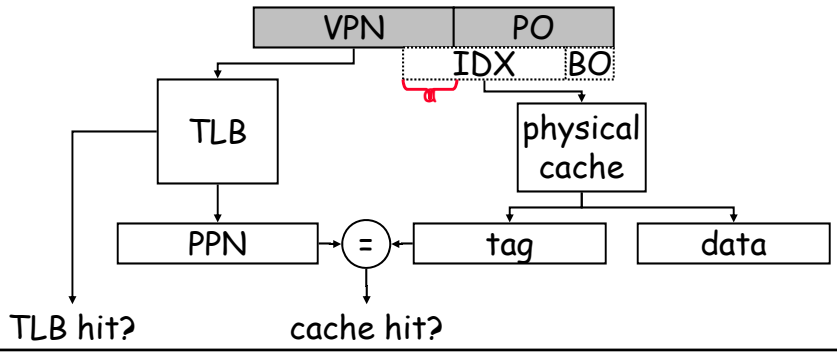
- If C≤(page_size × associativity), the cache index bits come only from page offset (same in VA and PA)
- If both cache and TLB are on chip
  - index both arrays concurrently using VA bits
  - check cache tag (physical) against TLB output at the end



Only an issue for L1 caches

Large Virtually-Indexed Caches

- If C>(page_size × associativity), the cache index bits include VPN ⇒ Synonyms can cause problems
- Solutions
  - increase associativity
  - increase page size
  - MIPS R10K



R10000's Virtually Index Caches

- 32KB 2-Way Virtually-Indexed L1
  - needs 10 bits of index and 4 bits of block offset
  - page offset is only 12-bits ⇒ 2 bits of index are VPN[1:0]
- Direct-Mapped Physical L2
  - L2 is inclusive of L1
  - VPN[1:0] is appended to the "tag" of L2
- Given two virtual addresses VA and VB that differs in a and both map to the same physical address PA
  - Suppose VA is accessed first so blocks are allocated in L1&L2
  - What happens when VB is referenced?
    1. VB indexes to a different block in L1and misses
    2. VB translates to PA and goes to the same block as VA in L2
    3. Tag comparison fails (VA[1:0]≠VB[1:0])
    4. L2 detects that a synonym is cached in L1 ⇒ VA's entry in L1 is ejected before VB is allowed to be refilled in L1