18-447
Virtual Memory, Protection and Paging

James C. Hoe and José F. Martínez
Dept of ECE, CMU

March 22-24, 2010

Assigned Reading
"Virtual memory in contemporary microprocessors."
B. L Jacob and T. N. Mudge. IEEE Micro, July/August 1998.
(on Blackboard)

2 Parts to Modern VM

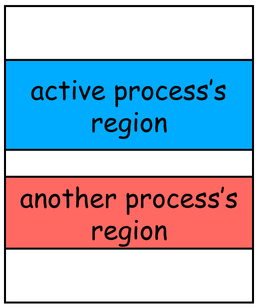
- In a multi-tasking system, VM provides each process with the illusion of a <u>large</u>, <u>private</u>, and <u>uniform</u> memory
- Ingredient A: Naming and Protection
  - each process sees a large, contiguous memory segment without holes ("virtual" address space is much bigger than the available physical storage)
  - each process's memory space is private, i.e. protected from access by other processes
- Ingredient B: demand paging
  - capacity of secondary storage (swap space on disk)
  - at the speed of primary storage (DRAM)

Mechanism: Address Translation

- Protection, VM and demand paging enabled by a common HW mechanism: address translation
- User process operates on "effective" addresses
- HW translates from EA to PA on every memory reference
  - controls which physical locations (DRAM and/or swap disk) can be named by a process
  - allows dynamic relocation of physical backing store (DRAM vs. swap disk)
- Address translation HW and policies controlled by the OS and protected from users, i.e., priviledged

Evolution of Memory Protection

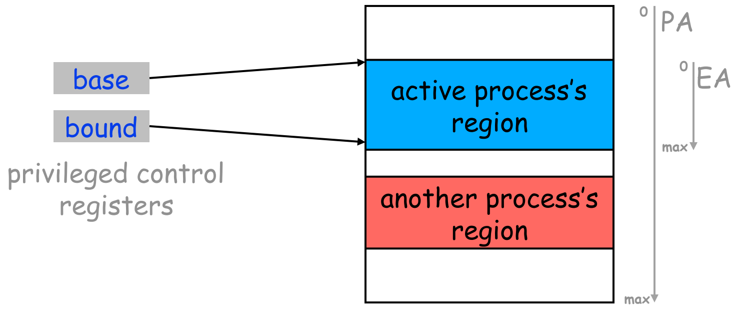
- Earliest machines did not have the concept of protection and address translation
  - no need---single process, single user
    automatically "private and uniform" (but not very large)
  - programs operated on physical addresses directly
    cannot support multitasking protection
- Multitasking 101
  - give each process a non-overlapping, contiguous physical memory region
  - everything belonging to a process must fit in that region
  - how do you keep one process from reading or trashing another process' data?



Base and Bound

- A process's private memory region can be defined by
  - **base**: starting address of the region
  - **bound**: ending address of the region
- User process issue "effective" address (EA) between 0 and the size of its allocated region

private and uniform



Base and Bound Registers

- When switching user processes, OS sets base and bound registers
- Translation and protection check <u>in hardware</u> on <u>every</u> user memory reference
  - PA = EA + **base**
  - if (PA < **bound**) then okay else violation
- User processes cannot be allowed to modify the base and bound registers themselves

  ⇒ requires 2 privilege levels such that the OS can see and modify certain states that the user cannot

  privileged instructions and state

Segmented Address Space

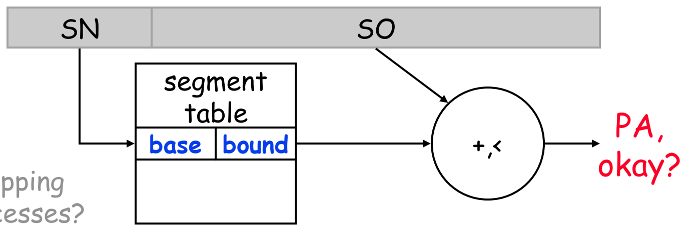
- Limitations of the base and bound scheme
  - large contiguous space is hard to come by after the system runs for a while---free space may be fragmented
  - how do two processes shared some memory regions but not others?
- A "base&bound" pair is the unit of protection

  ⇒ give user multiple memory "segments"
  - each segment is a continuous region
  - each segment is defined by a base and bound pair
- Earliest use, separate code and data segments
  - 2 sets of base-and-bound reg's for inst and data fetch
  - allow processes to share read-only code segments

  became more elaborate later: code, data, stack, etc

Segmented Address Translation

- EA comprise a segment number (SN) and a segment offset (SO)
  - SN may be specified explicitly or implied (code vs. data)
  - segment size limited by the range of SO
  - segments can have different sizes, not all SOs are meaningful
- Segment translation table
  - maps SN to corresponding base and bound
  - separate mapping for each process
  - must be a privileged structure



how to change mapping
when swapping processes?

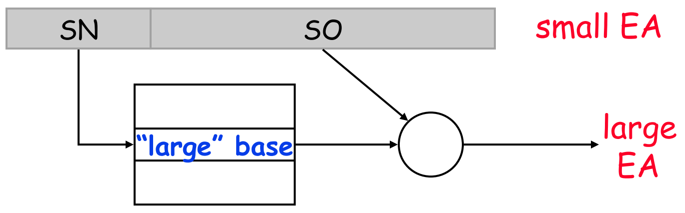
Access Protections

- In addition to naming, finer-grain access protection can be associated with each segment as extra bits in the segment table
- Generic options include
  - readable?
  - writeable?
  - executable?

  also misc. options such as cacheable? which level?
- Normal data pages ⇒ RW(!E)
- Static shared data pages ⇒ R(!W)(!E)
- Code pages ⇒ R(!W)E   What about self modifying code?
- Illegal pages ⇒ (!R)(!W)(!E)   Why would any one want this?

Another Use of Segments

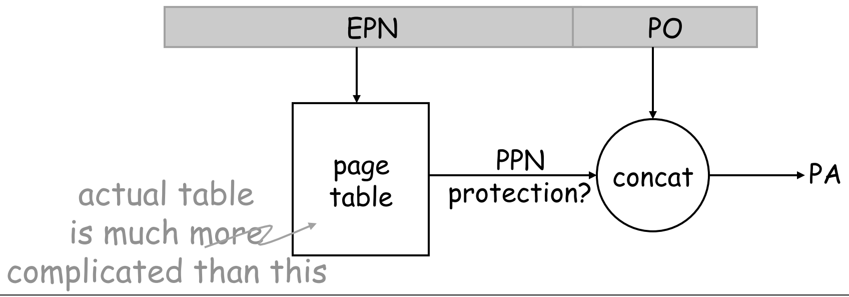
- How to extend an old ISA to support larger addresses for new applications while remain compatible with old applications?
- User-level segmented addressing
  - old applications use identity mapping in table
  - new applications reload segment table at run time with "active segments" to access different regions in memory
  - complications of a "non-linear" address space: dereferencing some pointers (aka long pointers) requires more work if it is not in an active segment



can be orthogonal from protection considerations

Paged Address Space

- Divide PA and EA space into fixed size segments known as "page frames", historically 4KByte
- EAs and PAs are interpreted as page number (PN) and page offset (PO)
  - page table translates EPN to PPN
  - VPO is the same as PPO, just concatenate to PPN to get PA



Fragmentation

- External Fragmentation
  - a system may have plenty of unallocated DRAM, but they are useless in a segmented system if they do not form a contiguous region of a sufficient size
  - paged memory management eliminates external fragmentation
- Internal Fragmentation
  - with paged memory management, a process is allocated an entire page (4KByte) even if it only needs 4 bytes
  - a smaller page size reduces likelihood for internal fragmentation
  - modern ISA are moving to larger page sizes (Mbytes) in addition to 4KBytes Why?

Demand Paging

- Use main memory and "swap" disk as automatically managed levels in the memory hierarchies

  analogous to cache vs. main memory

- Early attempts
  - von Neumann already described manual memory hierarchies
  - Brookner's interpretive coding, 1960
    - a software interpreter that managed paging between a 40KByte main memory and a 640KByte drum
  - Atlas, 1962
    - hardware demand paging between a 32-page (512 word/page) main memory and 192 pages on drums
    - user program believes it has 192 pages

Demand Paging vs. Caching

- Drastically different size and time scale

  ⇒ drastically different design considerations

	L1 Cache	L2 Cache	Demand Paging
Capacity	10~100KByte	MByte	GByte
Block size	~16 Byte	~128 Byte	4K~4M Byte
hit time	a few cyc	a few 10s cyc	a few 100s cyc
miss penalty	a few 10s cyc	a few 100s cyc	10 msec
miss rate	0.1~10%	(?)	0.00001~0.001%
hit handling	HW	HW	HW
miss handling	HW	HW	SW

Hit time, miss penalty and miss rate are not really independent variables!!

Same Fundamentals

- Potentially $M=2^m$ bytes of memory, how to keep the most frequently used ones in C bytes of fast storage where $C << M$

DRAM in the case of demand paging

- Basic issues

  (1) where to "cache" a virtual page in DRAM?

  (2) how to find a virtual page in DRAM?

  (3) granularity of management

  (4) when to bring a page into DRAM?

  (5) which virtual page to evict from DRAM to disk to free-up DRAM for new pages?

  (5) is much better covered in an OS course

  BTW, architects should take OS and compiler

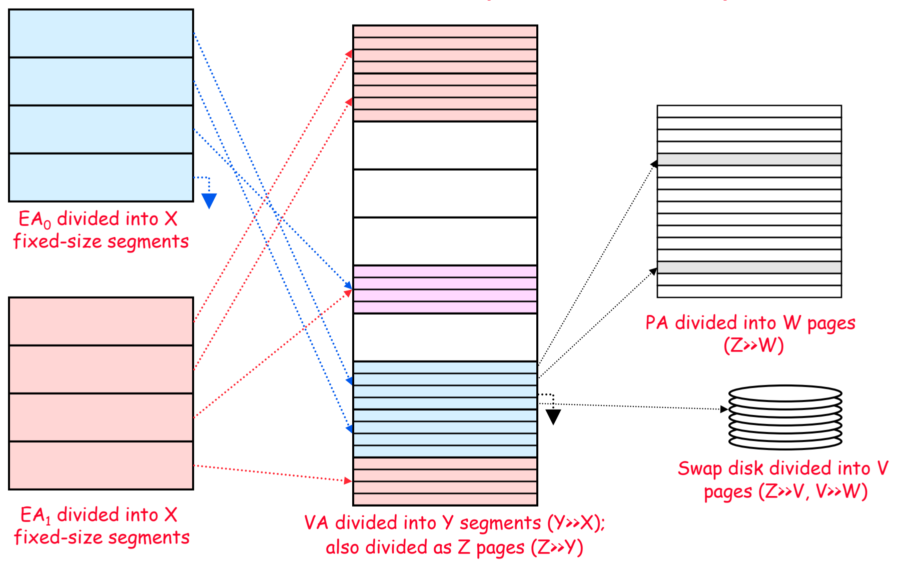
Terminology and Usage

- Physical Address
  - addresses that refer directly to specific locations on DRAM or on swap disk
- Effective Address
  - addresses emitted by user applications for data and code accesses
  - usage is most often associated with "protection"
- Virtual Address
  - addresses in a large, linear memory seen by user app's

    often larger than DRAM and swap disk capacity
  - virtual in the sense that some addresses are not backed up by physical storage (hopefully you don't try to use it)
  - usage is most often associated with "demand paging"

Modern memory system always have both protection and demand paging; usage of EA and VA is sometimes muddled

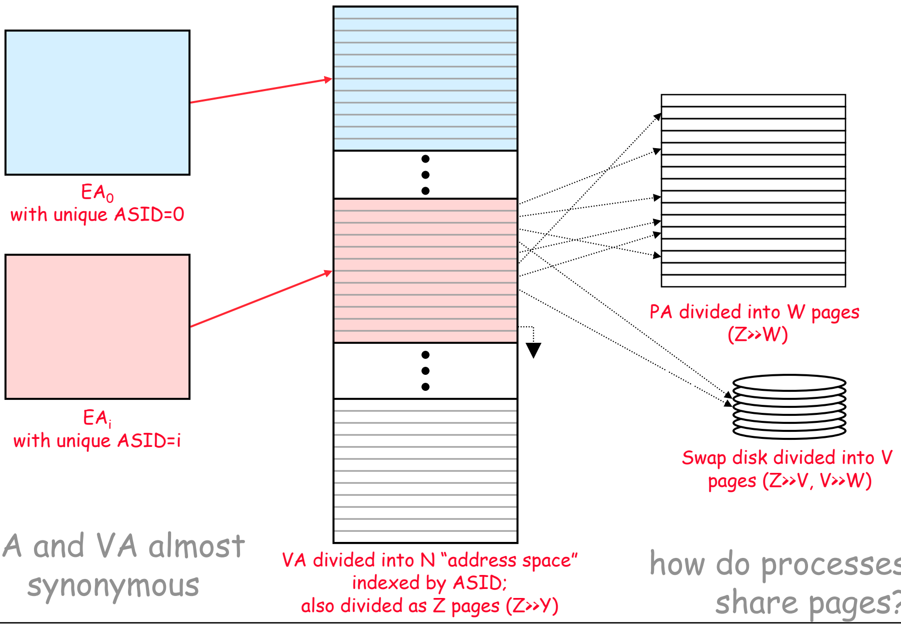
EA, VA and PA (IBM's view)


EA$_0$ divided into X fixed-size segments
EA$_1$ divided into X fixed-size segments
VA divided into Y segments (Y>>X); also divided as Z pages (Z>>Y)
PA divided into W pages (Z>>W)
Swap disk divided into V pages (Z>>V, V>>W)

segmented EA:
private, contiguous + sharing

demand paged VA:
size of swap, speed of DRAM

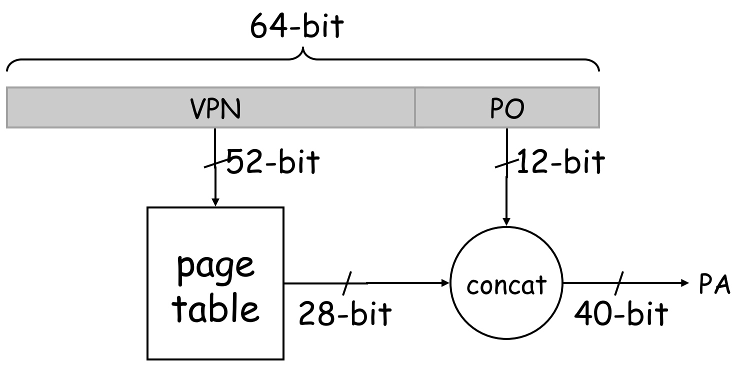
EA, VA and PA (almost everyone else)


EA$_0$ with unique ASID=0
EA$_i$ with unique ASID=i
VA divided into N "address space" indexed by ASID; also divided as Z pages (Z>>Y)
PA divided into W pages (Z>>W)
Swap disk divided into V pages (Z>>V, V>>W)

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 ≈ 16x10^{15} Bytes

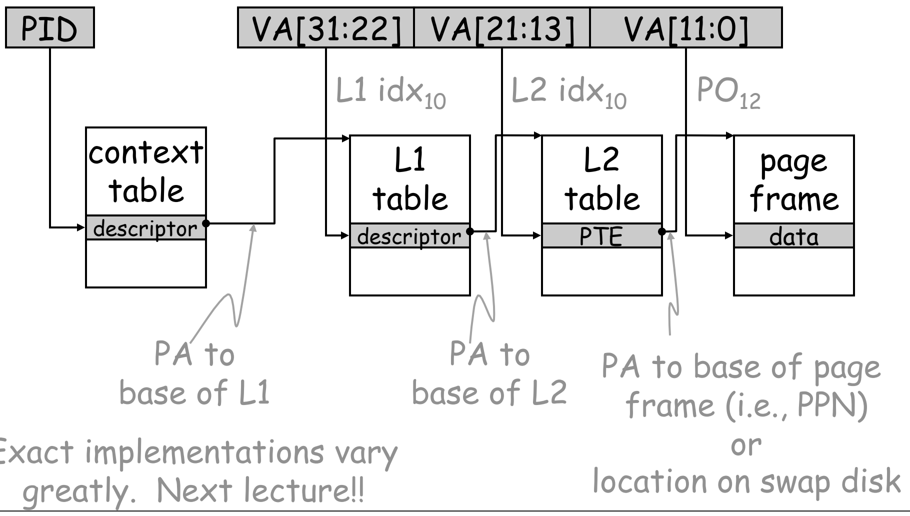
and that is for just one process!!?

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?

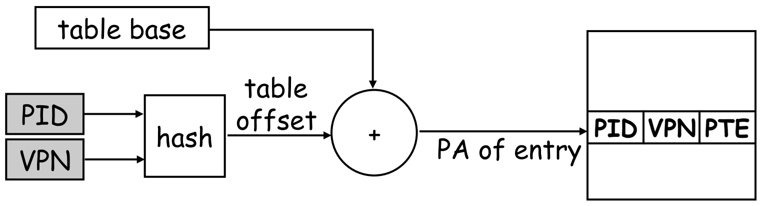
How large is the hierarchical table?

- 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 ≈ 1 page per data 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:

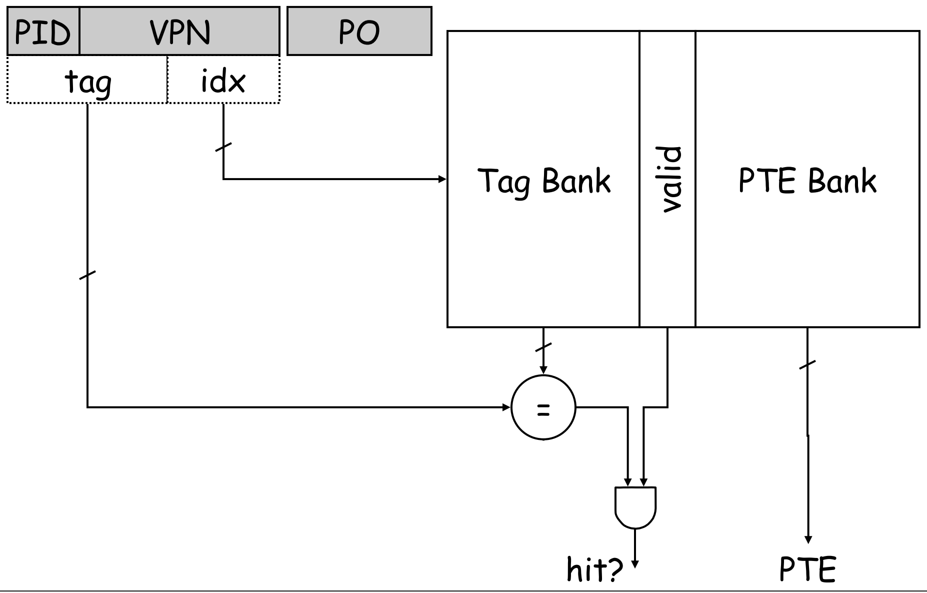
    tag: address tag (from VA), PID

    PTE: PPN, protection bits

    misc: valid, dirty, etc.
  - similar design considerations as caches

    capacity, block size, associativity, replacement policy

Direct-Mapped TLB (bad example)



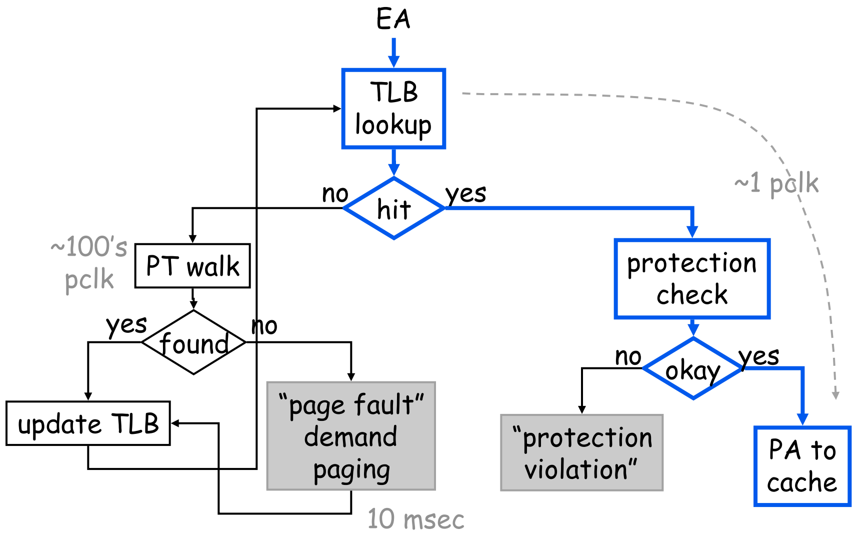
TLB Design

- 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?

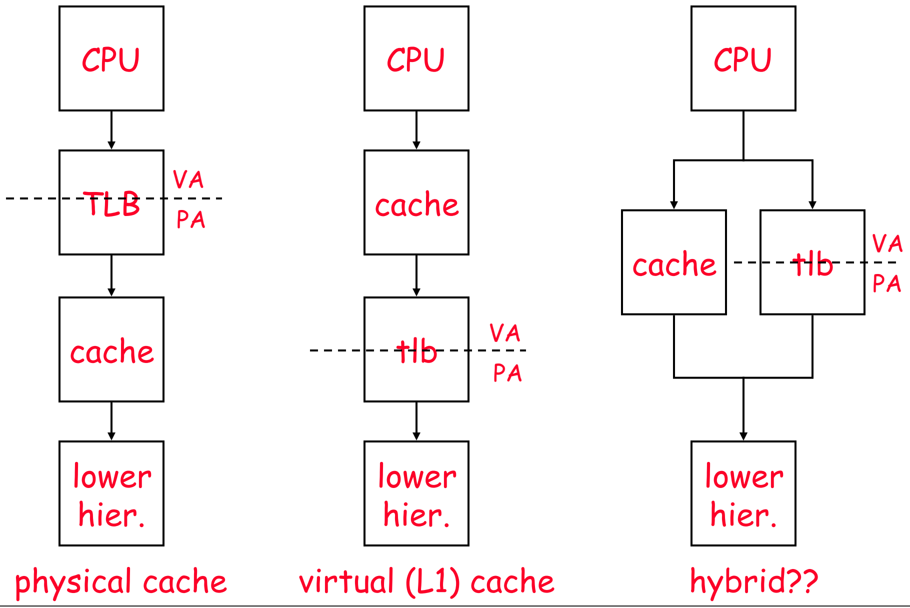
On a TLB Miss

- 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

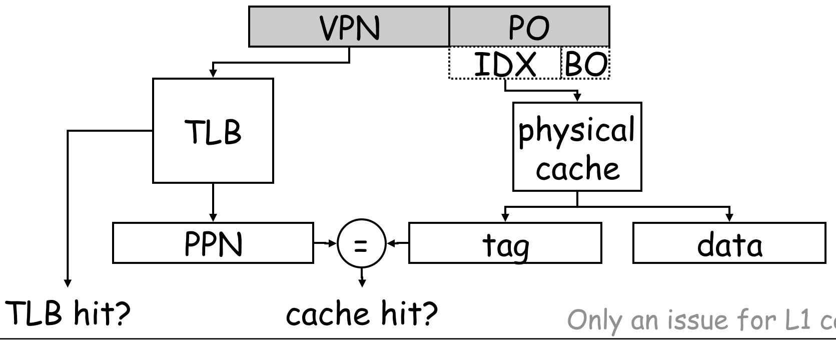
the conditions no longer hold

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)

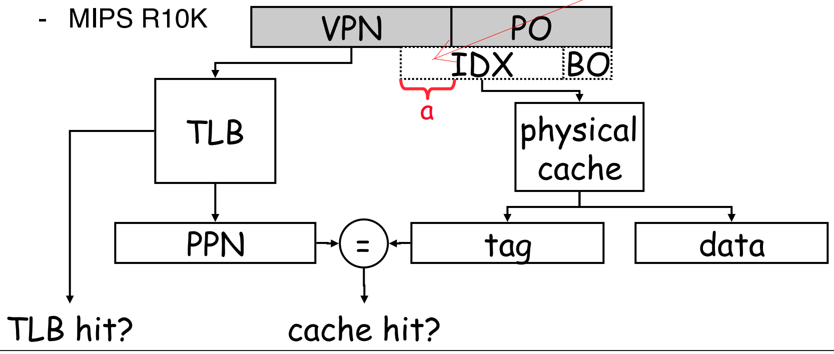
- If C≤(page_size × associativity), the cache index bits come only from page offset (same in VA and PA)
- If both cache and TLB are on chip
  - index both arrays concurrently using VA bits
  - check cache tag (physical) against TLB output at the end



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
  - page coloring
  - MIPS R10K



Two VAs may have different bits in "a" and therefore map to different cache sets, yet translate into the same PA -> must force IDX to stay within PO to guarantee conflict between synonyms in the cache

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