TLB Performance

- TLB Reach = # TLB entries * Page size
  = 64 * 4KB = 256KB << L2 cache size

Solution #1: Big pages (e.g., 4MB)

TLB Reach = 256MB, but internal fragmentation

How to support both big and small pages?

Solution #2: Two-level TLB

L1: 64-128 entries, L2: 512-2048 entries

Solution #3: Software TLB (aka TSB)

in memory TLB: 32K entries (or more)

low-associativity (e.g., 2-way), longer hit time

Much faster than page table access

TLB Misses and Miss Handling

- **TLB miss:** requested PTE not in TLB, search page table
  - **Software routine**, e.g., Alpha, SPARC, MIPS -- **virtual** PT
    - Special instructions for accessing TLB directly
    - Latency: one or two memory accesses + trap
  - **Hardware finite state machine (FSM),** e.g., x86 – **physical** PT
    - Store page table root in hardware register
    - Page table root and table pointers are physical addresses
    + Latency: saves cost of OS call
  - In both cases, reads use the the standard cache hierarchy
    + Allows caches to help speed up search of the page table
- **Nested TLB miss**: miss handler itself misses in the TLB
  - Solution #1: Allow recursive TLB misses (very tricky)
  - Solution #2: Lock TLB entries for handler into TLB
  - Solution #3: Avoid problem using physical address in page table

Page Faults

- **Page fault**: PTE not in page table
  - Page is simply not in memory
  - Starts out as a TLB miss, detected by OS handler/hardware FSM

- **OS routine**
  - Choose a physical page to replace
    - **"Working set"**: more refined software version of LRU
      - Tries to see which pages are actively being used
      - Balances needs of all current running applications
    - If dirty, write to disk
  - Read missing page from disk
    - Takes so long (~10ms), OS schedules another task
  - Treat like a normal TLB miss from here

Page Table Size

- How big is a page table on the following machine?
  - 4B page table entries (PTEs)
  - 32-bit machine
  - 4KB pages

  - 32-bit machine → 32-bit VA → 4GB virtual memory
  - 4GB virtual memory / 4KB page size → 1M VPs
  - 1M VPs * 4B PTE → 4MB

- How big would the page table be with 64KB pages?
- How big would it be for a 64-bit machine?

- Page tables can get big
  - There are ways of making them smaller
  - PA = f(VA) ➔ many different data structures possible

Page Table Size

- There has to be some property that we can exploit to cut page table size
- What property?
- How would you exploit it?

Multi-Level Page Table

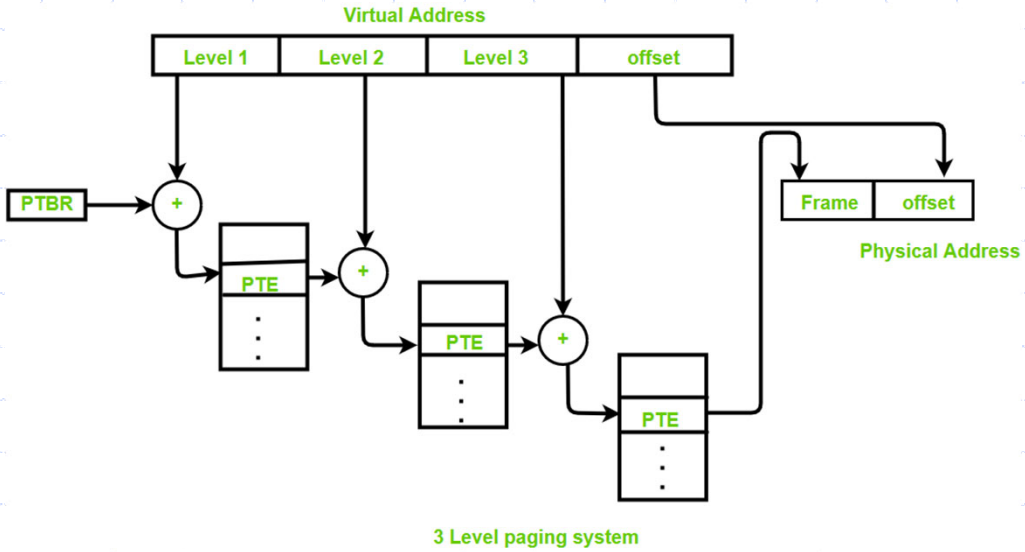
- One way: **multi-level page tables**
  - Tree of page tables
  - Lowest-level tables hold PTEs
  - Upper-level tables hold pointers to lower-level tables
  - Different parts of VPN used to index different levels

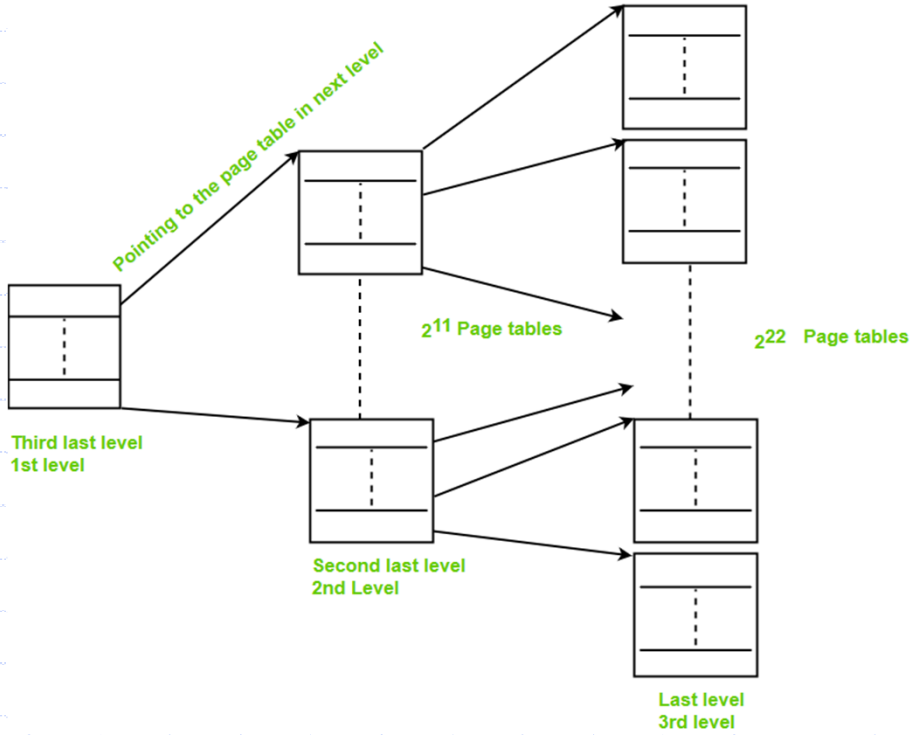
- Example: two-level page table for machine on last slide
  - Compute number of pages needed for lowest-level (PTEs)
    - 4KB pages / 4B PTEs → 1K PTEs/page
    - 1M PTEs / (1K PTEs/page) → 1K pages
  - Compute number of pages needed for upper-level (pointers)
    - 1K lowest-level pages → 1K pointers
    - 1K pointers * 32-bit (4B) VA → 4KB → 1 upper level page

Multi-Level Page Table option 1: virtual

- multi-level page table (virtual)
  - 1st-level PT in physical memory
  - 2nd-level PT in kernel virtual space
  - not all 2nd-level pages have to exist
    - 1st-level points to disk if 2nd-level page not exist
    - else 1st-level points to 2nd-level page in memory
- hardware support
  - base registers to point to PTs
- Eg Sparc

Multi-Level Page Table option 1: virtual





Multi-Level Page Table option 1: virtual

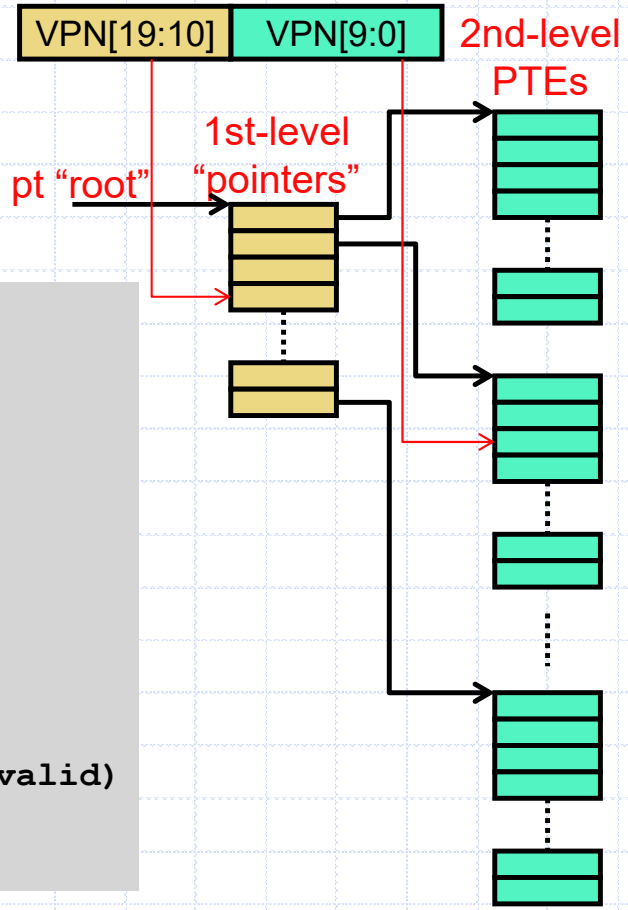
- Think of 1st level as "page table" for the process's page table
- Page table is accessed on a TLB miss (TLB hit => no access)
- On a TLB miss, an exception is raised and OS comes in
- OS uses process's base register to access 1st level (in memory)
    - if 1st-level points to disk then a page fault
        - 2nd-level page not in memory, and is brought
        - actual data page may or may not exist in memory
            - yes => use 2nd-level to update TLB
            - no => recursive fault (update 2nd level + TLB)

Multi-Level Page Table option 2: physical

- multi-level page table (physical)
  - all levels in physical memory
- hardware support
  - base register points to root PT
- Eg x86

Physical Page Table

- 20-bit VPN
  - Upper 10 bits index 1st-level table
  - Lower 10 bits index 2nd-level table



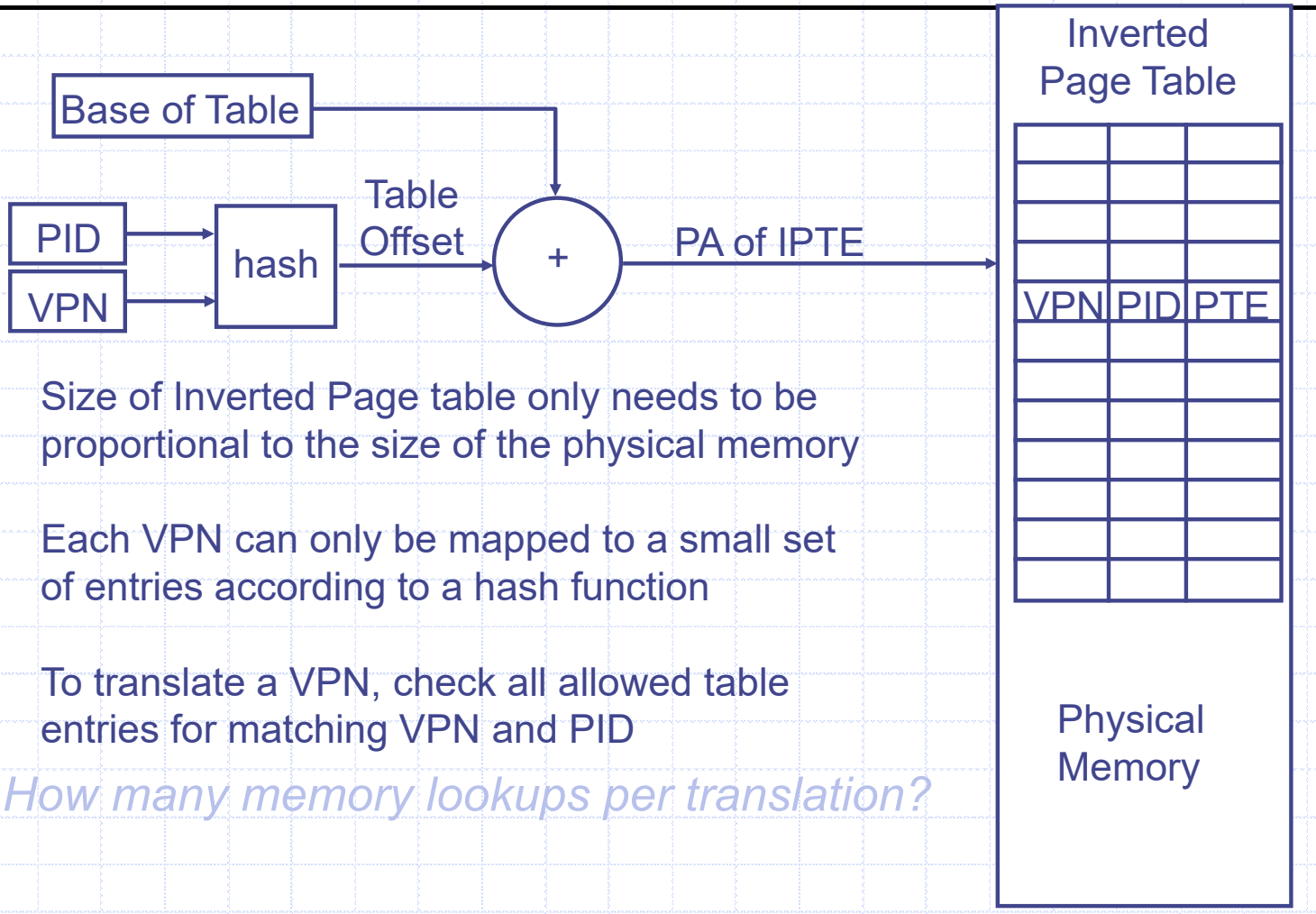

```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty;
} PTE;
struct {
   struct PTE ptes[1024];
} L2PT;
struct L2PT *pt[1024];

int translate(int vpn) {
  struct L2PT *l2pt = pt[vpn>>10];
  if (l2pt && l2pt->ptes[vpn&1023].is_valid)
     return l2pt->ptes[vpn&1023].ppn;
}
```

option 3: hashed

- inverted
  - hashed virtual address points to hash table
  - hash table entry points to linked list of PTEs
- size of page table =
  - (phys mem size / page size) x table entry size x safety factor for hash collisions
- hash function
  - typically XOR upper and lower bits of VPN and PID (process ID)
- Eg IBM POWER1

Alternative: Inverted/Hashed Page Tables



Size of Inverted Page table only needs to be proportional to the size of the physical memory

Each VPN can only be mapped to a small set of entries according to a hash function

To translate a VPN, check all allowed table entries for matching VPN and PID

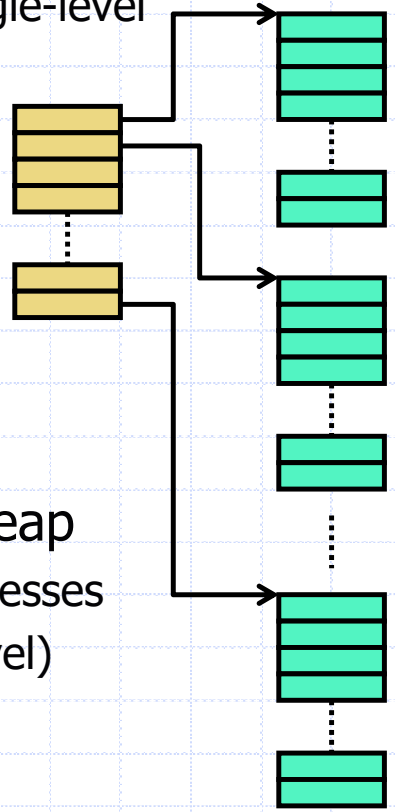
How many memory lookups per translation?

Virtual vs. Physical Page Table

- upon TLB miss (you do not access page table as long as TLB hits)
  - option 1 - virtual: because 2nd level is in OS's VA =>
    - page table access will require OS intervention
    - => TLB miss is handled in s/w (OS)
    - => slow but because 2nd level in OS's VA => needs less physical memory
  - option 2- physical: because all levels are in physical mem
    - page table access needs no OS
    - => TLB miss handled in h/w
    - => fast but needs all levels in memory => needs more physical memory

Multi-Level Page Table (PT)

- Have we saved any space?
  - Isn't total size of 2nd level tables same as single-level table (i.e., 4MB)?
  - Yes, but…
- Large virtual address regions unused
  - Corresponding 2nd-level tables need not exist
  - Corresponding 1st-level pointers are null
- Example: 2MB code, 64KB stack, 16MB heap
  - Each 2nd-level table maps 4MB of virtual addresses
  - 1 for code, 1 for stack, 4 for heap, (+1 1st-level)
  - 7 total pages = 28KB (much less than 4MB)



Page table and TLB interaction

- you access page tables ONLY on TLB miss
  - you do NOT have multi-level entries in TLB like page table
    - instead the page table is traversed (in h/w or s/w)
    - and the FINAL physical address is put in TLB
    - and NOT the intermediate levels
- page table changes due to page replacement from physical memory
  - replacement => page table change => TLB MAY be stale
  - will need OS to invalidate TLB called TLB shootdown
  - And invalidate blocks in cache (translation has changed so blocks cannot stay in cache with old phy addr
  - so unlike hardware caching TLB/caches visible to OS

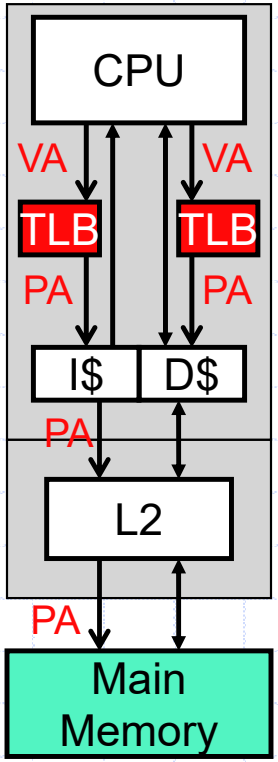
Context-switch- TLB interaction

- TLB holds VA to PA translation and is indexed by VA
- Context switches: all VAs look alike => so new process will use old process's translations
    - flush the TLB at context switch (slow)
    - put PIDs (process IDs) in TLB entries (better)

Cache- TLB interaction

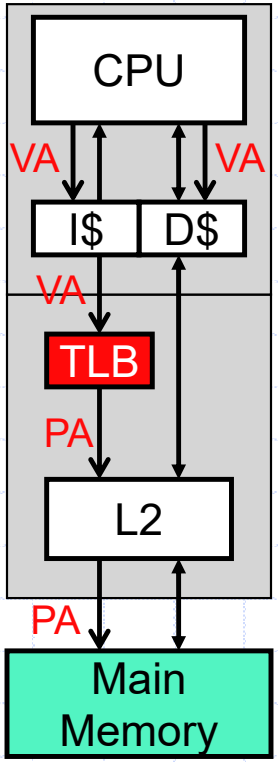
- With TLB, one TLB access instead of memory access before data access
- But most data access is cache hit → TLB access before cache access
  - adds at least 1-2 cycles to L1 $

Physical (Address) Caches



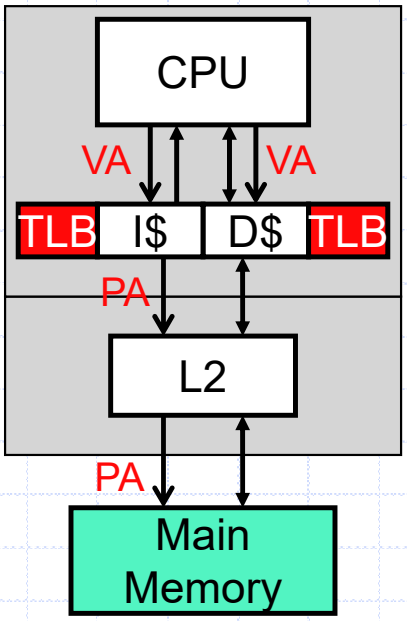
- Memory hierarchy so far: **physical caches**
  - Indexed and tagged by Pas
    - Physically Indexed (PI)
    - Physically Tagged (PT)
  - Translate to PA to VA at the outset
  - \+ Cached inter-process communication works
    - Single copy indexed by PA
  - – Slow: adds 1-2 cycles to t_{hit}

Virtual Address Caches (VI/VT)



- Alternative: **virtual caches**
  - Indexed and tagged by VAs (VI and VT)
  - Translate to PAs only to access L2 (L1 miss)
  - \+ Fast: avoids translation latency in common case
  - – Problem: VAs from ***different processes*** are distinct physical locations (with different values) (call homonyms)
- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags
- Does inter-process communication work?
  - **Synonyms**: multiple VAs map to same PA
    - Can't allow same PA in the cache twice
  - Can be handled, but very complicated

Parallel TLB/Cache Access (VI/PT)



Compromise: **access TLB in parallel**

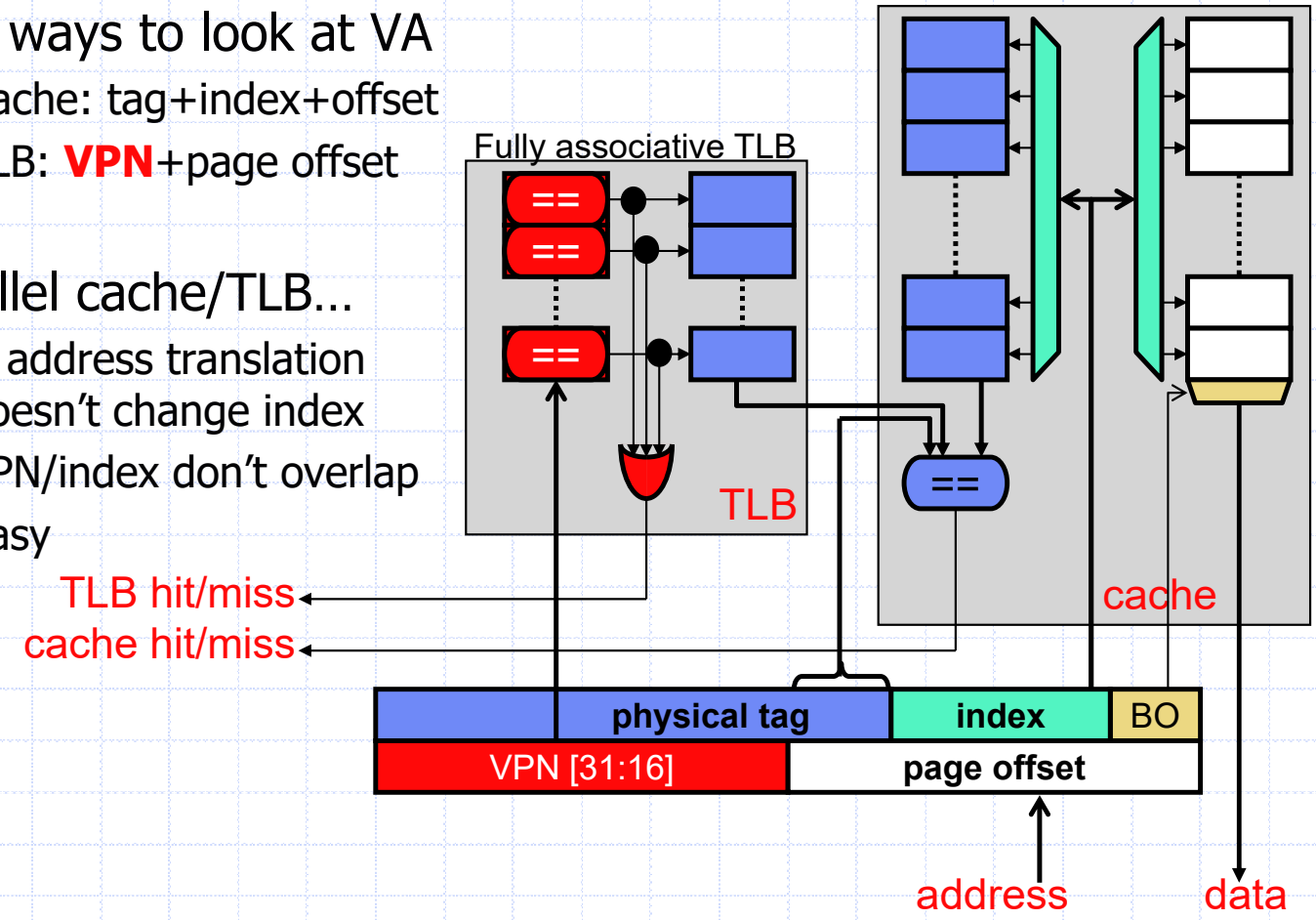
- *Ok for small caches*
- *In small caches, index of VA and PA the same*
  - *VI == PI*
- Use the VA to index the cache
- Tagged by PA
- Virtually-indexed physically-tagged
- Cache access and address translation in parallel

+ No context-switching/aliasing problems
+ Fast: no additional t_{hit} cycles

- Common organization in processors today

Parallel Cache/TLB Access – small cache

- Two ways to look at VA
  - Cache: tag+index+offset
  - TLB: **VPN**+page offset
- Parallel cache/TLB…
  - If address translation doesn't change index
  - VPN/index don't overlap
  - Easy



Parallel Cache/TLB Access – large cache

- Difficult
- VPN-index overlap means synonyms can go to different sets
  - Different VA but same PA
  - Different VA ➔ different overlap bits
  - Synonyms won't work correctly

Parallel Cache/TLB Access – large cache

- limit cache size to page size times associativity
  - Fit index within page offset
- search more sets in parallel
  - 64K 4-way cache, 4K pages - search 4 sets (16 entries)
- restrict page placement in OS ("page coloring" in Solaris)
  - guarantee that index(VA) == index(PA)
- eliminate by OS convention
  - single virtual space
  - restrictive sharing model

Common today

- L1 caches - Virtually indexed physically tagged
  - cannot do physical indexing because TLB in critical path
  - so, virtual indexing and physical tagging
    - TLB gives physical tag in parallel with L1 access
    - need to solve synonyms using above schemes
- L2 caches - physically indexed physically tagged
  - physical index no problem because TLB done during L1

Virtual Memory

- Virtual memory ubiquitous today
  - Certainly in general-purpose (in a computer) processors
  - But even many embedded (in non-computer) processors support it

- Several forms of virtual memory
  - **Paging** (aka flat memory): equal sized translation blocks
    - Most systems do this
  - **Segmentation**: variable sized (overlapping?) translation blocks
    - x86 used this rather than 32-bits to break 16-bit (64KB) limit
    - Makes life hell
  - **Paged segments**: don't ask

Memory Protection and Isolation

- Most important role of virtual memory today
- Virtual memory protects applications from one another
  - OS uses indirection to isolate applications
  - One buggy program should not corrupt the OS or other programs
  + Comes "for free" with translation
  – However, the protection is limited
- What about protection from…
  - Viruses and worms?
    - Stack smashing
  - Malicious/buggy services?
    - Other applications with which you want to communicate
    - These exploit bugs in code to circumvent VM

Page-Level Protection

```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty, permissions;
} PTE;
```

- **Page-level protection**
  - Piggy-backs on translation infrastructure
  - Each PTE associated with permission bits: **R**ead, **W**rite, e**X**ecute
    - **Read/execute (RX)**: for code
    - **Read (R)**: read-only data
    - **Read/write (RW)**: read-write data
  - TLB access traps on illegal operations (e.g., write to **RX** page)
  - To defeat stack-smashing? Set stack permissions to **RW**
    - Will trap if you try to execute `&buf[0]`

  \+ **X** bits added to x86 for this specific purpose

  – Unfortunately, hackers have many other tricks