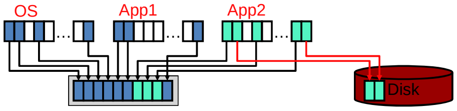
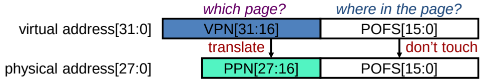
Virtual Memory: The Basics

- **Programs use virtual addresses (VA)**
  - VA size (N) aka machine size (e.g., Core 2 Duo: 48-bit)
- **Memory uses physical addresses (PA)**
  - PA size (M) typically M<N, especially if N=64
  - 2^M is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap) or nowhere (if not yet touched)



Address Translation



- **VA→PA mapping called address translation**
  - Split VA into virtual page number (VPN) & page offset (POFS)
  - Translate VPN into physical page number (PPN)
  - POFS is not translated
  - VA→PA = [VPN, POFS] → [PPN, POFS]

Example above

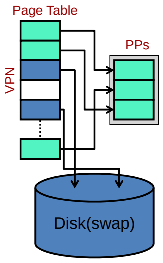
64KB per page → 16-bit POFS	(2^{16}=64K)
32-bit machine → 32-bit VA → 16-bit VPN	32-16=16
Max. 256MB memory → 28-bit PA → 12-bit PPN	28-16=12
	(2^{28}=256M

Address Translation Mechanics I

- How are addresses translated?
  - In sw (for now) but with hardware acceleration (a little later)
- Each process allocated a **page table (PT)**
  - **Software data structure constructed by OS**
  - Maps VPs to PPs or to disk (swap) addresses
    - VP entries empty if page never referenced
  - Translation is table lookup

```
struct {
   int ppn;
   int is_valid, is_dirty, is_swapped;
} PTE;
struct PTE page_table[NUM_VIRTUAL_PAGES];

int translate(int vpn) {
  if (page_table[vpn].is_valid)
     return page_table[vpn].ppn;
}
```



Page Table Size

How big is a page table on the following machine?

Given:

- 32-bit machine
- 4KB per page
- 4B page table entries (PTEs) (see struct definition, prev slide)

Can determine:

- 32-bit machine → 32-bit VA → 4GB virtual memory (2^{32}=4G)
- 4GB virtual memory / 4KB page size → 1M VPs
- Each VP needs a PTE: 1M VPs → 1M PTEs
- 1M PTEs x 4B-per-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* (see next slide)

Multi-Level Page Table (PT)

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 page size / 4B-per-PTE → can hold 1K PTEs per page
  - 1M PTEs / (1K PTEs/page) → 1K pages
- Compute # of pages needed for upper-level (pointers)
  - 1K lowest-level pages → 1K pointers
  - 1K pointers x 32-bit VA → 4KB → 1 upper level page

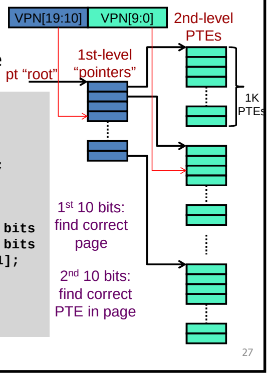
Multi-Level Page Table (PT)

20-bit VPN

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

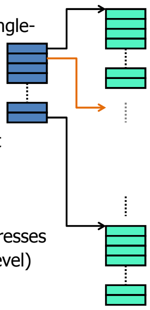
```
struct {
   int ppn;
   int is_valid, is_dirty, is_swapped;
} PTE;
struct { struct PTE ptes[1024]; } L2PT;
struct L2PT *page_table[1024];

int translate(int vpn) {
  index1 = (vpn >> 10);    // upper 10 bits
  index2 = (vpn & 0x3ff);  // lower 10 bits
  struct L2PT *l2pt = page_table[index1];
  if (l2pt != NULL &&
      l2pt->ptes[index2].is_valid)
    return l2pt->ptes[index2].ppn;
}
```



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)



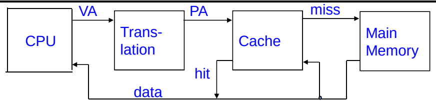
Page-Level Protection

- **Page-level protection**
  - Piggy-back page-table mechanism
  - Map VPN to PPN + Read/Write/Execute permission bits
  - Attempt to execute data, to write read-only data?
    - Exception → OS terminates program
  - Useful (for OS itself actually)

```
struct {
   int ppn;
   int is_valid, is_dirty, is_swapped, permissions;
} PTE;
struct PTE page_table[NUM_VIRTUAL_PAGES];

int translate(int vpn, int action) {
   if (page_table[vpn].is_valid &&
       !(page_table [vpn].permissions & action)) kill;
   …
}
```

Integrating VM and Cache



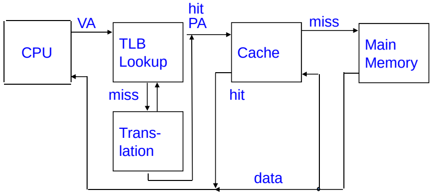
- Most Caches "Physically Addressed"
  - Accessed by physical addresses
  - Allows multiple processes to have blocks in cache at same time
  - Allows multiple processes to share pages
  - Cache doesn't need to be concerned with protection issues
    - Access rights checked as part of address translation
- Perform Address Translation Before Cache Lookup
  - But this could involve a memory access itself (of the PTE)
  - Of course, page table entries can also become cached

Address Translation Mechanics II

- Conceptually
  - Translate VA to PA before every cache access
  - Walk the page table before every load/store/insn-fetch
  - Would be terribly inefficient (even in hardware)
- In reality
  - Translation Lookaside Buffer **(TLB)**: cache translations
  - Only walk page table on TLB miss
- Hardware truisms
  - Functionality problem? Add indirection (*e.g.*, VM)
  - Performance problem? Add cache (*e.g.*, TLB)

Speeding up Translation with a TLB

- "Translation Lookaside Buffer" (TLB)
  - Small hw cache in MMU (memory management unit)
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages



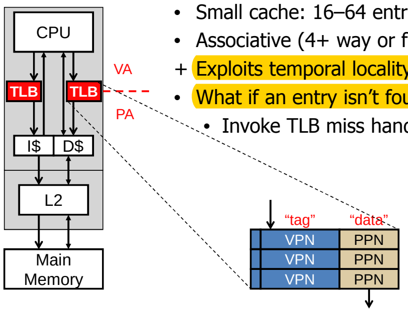
Translation Lookaside Buffer

Translation lookaside buffer (TLB)

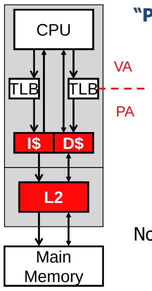
- Small cache: 16–64 entries
- Associative (4+ way or fully associative)

+ Exploits temporal locality in page table

- What if an entry isn't found in the TLB?
  - Invoke TLB miss handler



Serial TLB & Cache Access



"Physical" caches
- Indexed and tagged by **physical addresses**
- + Natural, "lazy" sharing of caches between apps/OS
  - VM ensures isolation (via **physical addresses**)
  - No need to do anything on context switches
  - Multi-threading works too
- + Cached inter-process communication works
  - Single copy indexed by physical address
- – Slow: adds at least one cycle to t_{hit}

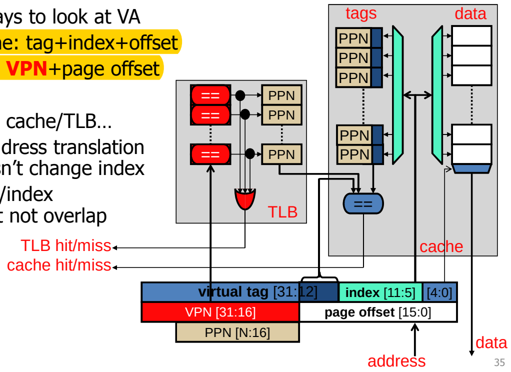
Note: **TLBs are by definition "virtual"**
- Indexed and tagged by **virtual addresses**
- Flush across context switches
- Or extend with process identifier tags (x86)

Parallel TLB & Cache Access

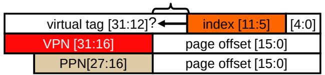
Two ways to look at VA
- Cache: tag+index+offset
- TLB: **VPN**+page offset

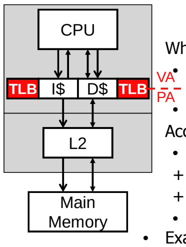
Parallel cache/TLB…
- If address translation doesn't change index

→ VPN/index must not overlap



Parallel TLB & Cache Access





What about parallel access?
- Only if…
- **(cache size) / (associativity) ≤ page size**
  - Index bits same in virtual & physical addresses!

Access TLB in parallel with cache
- Cache access needs tag only at very end
- + Fast: no additional t_{hit} cycles
- + No context-switching/aliasing problems
- Dominant organization used today

- Example: Core 2, 4KB pages, 32KB, 8-way SA L1 data cache
  - Implication: *associativity allows bigger caches*

TLB Organization

- **Like caches**: TLBs also have ABCs
  - Capacity
  - Associativity (At least 4-way associative, fully-associative common)
  - What does it mean for a TLB to have a block size of two?
    - Two consecutive VPs share a single tag
  - **Like caches**: there can be L2 TLBs

- Example: AMD Opteron
  - 32-entry fully-assoc. TLBs, 512-entry 4-way L2 TLB (insn & data)
  - 4KB pages, 48-bit virtual addresses, four-level page table

- **Rule of thumb**: TLB should "cover" L2 contents
  - In other words: (#PTEs in TLB) * page size ≥ L2 size
  - Why? Consider relative miss latency in each…

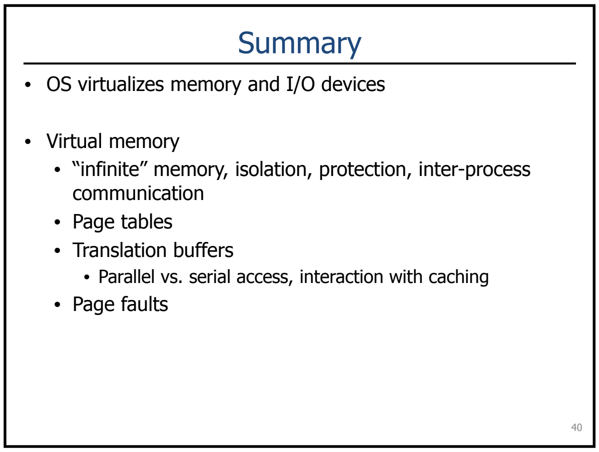
TLB Misses

- **TLB miss:** translation not in TLB, but in page table
  - Two ways to "fill" it, both relatively fast
- **Software-managed TLB**: *e.g.*, Alpha, MIPS, ARM
  - Short (~10 insn) OS routine walks page table, updates TLB
  - + Keeps page table format flexible
  - – Latency: one or two memory accesses + OS call (pipeline flush)
- **Hardware-managed TLB**: *e.g.*, x86
  - Page table root in hardware register, hardware "walks" table
  - + Latency: saves cost of OS call (avoids pipeline flush)
  - – Page table format is hard-coded
- Trend is towards hardware TLB miss handler

Page Faults

Page fault: PTE not in TLB or page table → page not in memory
- Or no valid mapping → segmentation fault
- Starts out as a TLB miss, detected by OS/hardware handler

OS software routine:
- Choose a physical page to replace
  - **"Working set"**: refined LRU, tracks active page usage
- 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



Summary

- OS virtualizes memory and I/O devices
- Virtual memory
  - "infinite" memory, isolation, protection, inter-process communication
  - Page tables
  - Translation buffers
    - Parallel vs. serial access, interaction with caching
  - Page faults