Virtual Memory II

Jo, Heeseung

Today's Topics

How to reduce the size of page tables?

How to reduce the time for address translation?

Page Tables

Space overhead of page tables

- The size of the page table for a 32-bit address space with 4KB pages = about 4MB (per process)
- For example
  - Virtual address: 32 bits (4G), Page size: 4KB (=2^{12})
  - Page table entries: 2^{20}, 4bytes/PTE

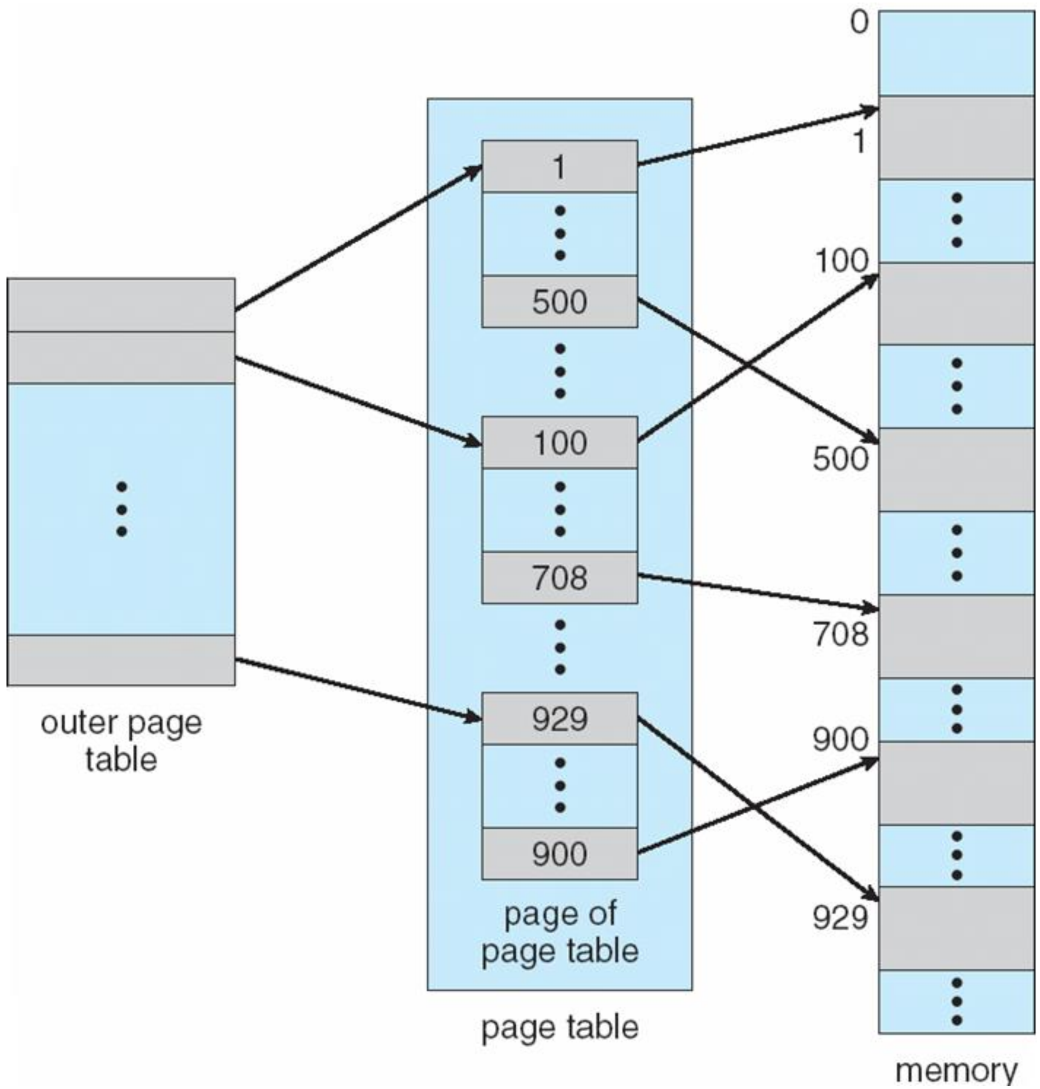
How can we reduce this overhead?

- Observation: Only need to map the portion of the address space actually being used
  - Tiny fraction of entire address space

How do we only map what is being used?

- Make the page table structure dynamically extensible
  - Linked list or tree?
- Use another level of indirection:
  - Two-level, hierarchical, hashed, etc.

Two-level Page Tables (1)



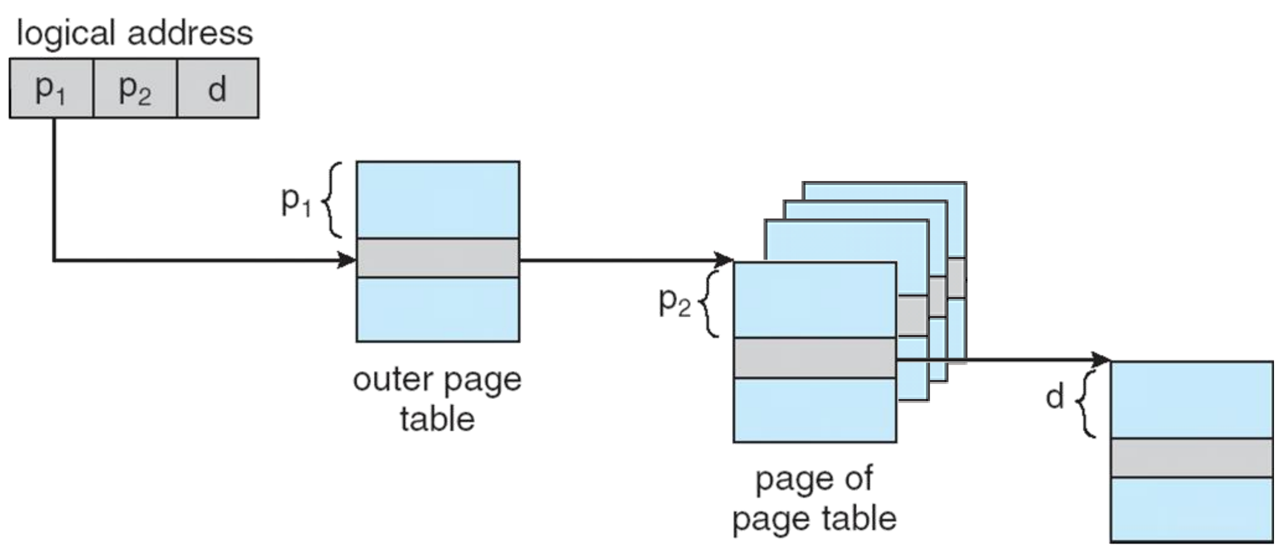
Two-level Page Tables (2)

Two-level page tables

- Virtual addresses have 3 parts:

Master page #	Secondary page #	Offset

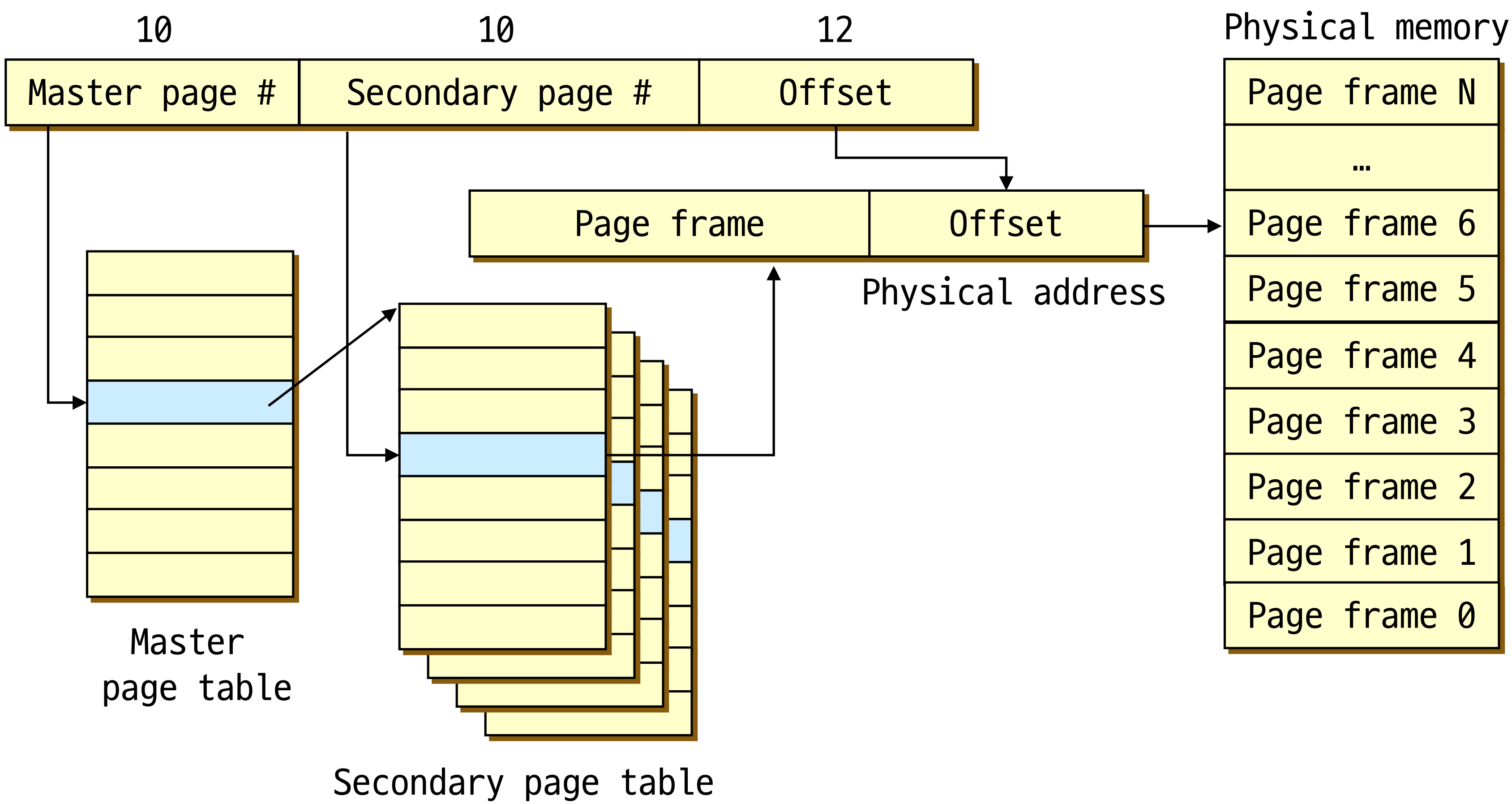
- Master page table
  - master page number -> secondary page table
- Secondary page table
  - secondary page number -> page frame number



Two-level Page Tables (3)

Example

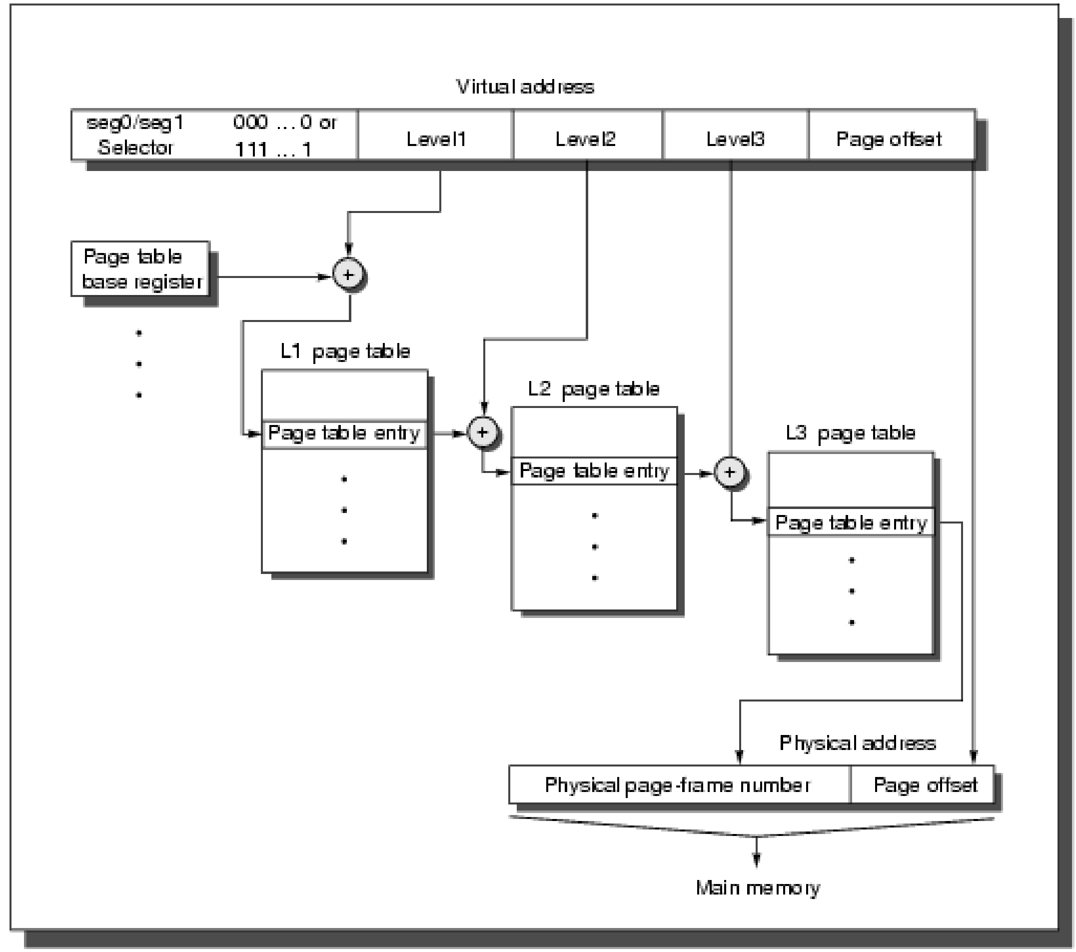
- 32-bit address space, 4KB pages, 4bytes/PTE
- Want master page table in one page (4KB)



Multi-level Page Tables

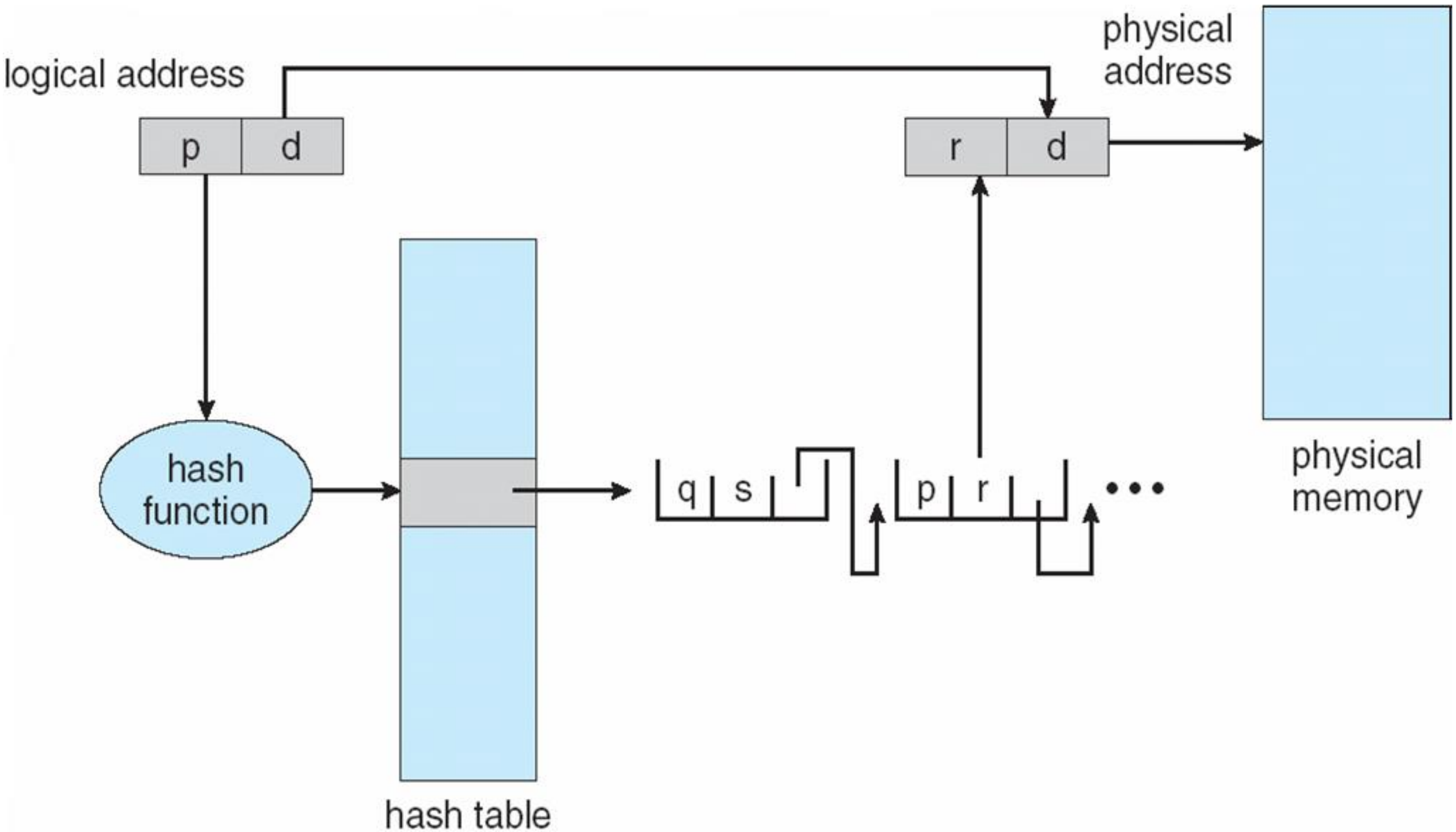
Address translation in Alpha AXP Architecture

- Three-level page tables
- 64-bit address divided into 3 segments
  - seg0 (0x): user code
  - seg1 (11): user stack
  - kseg (10): kernel
- Alpha 21064
  - Page size: 8KB
  - Virtual address: 43bits
  - Each page table is one page long



Hashed Page Tables (1)

Example



Hashed Page Tables (2)

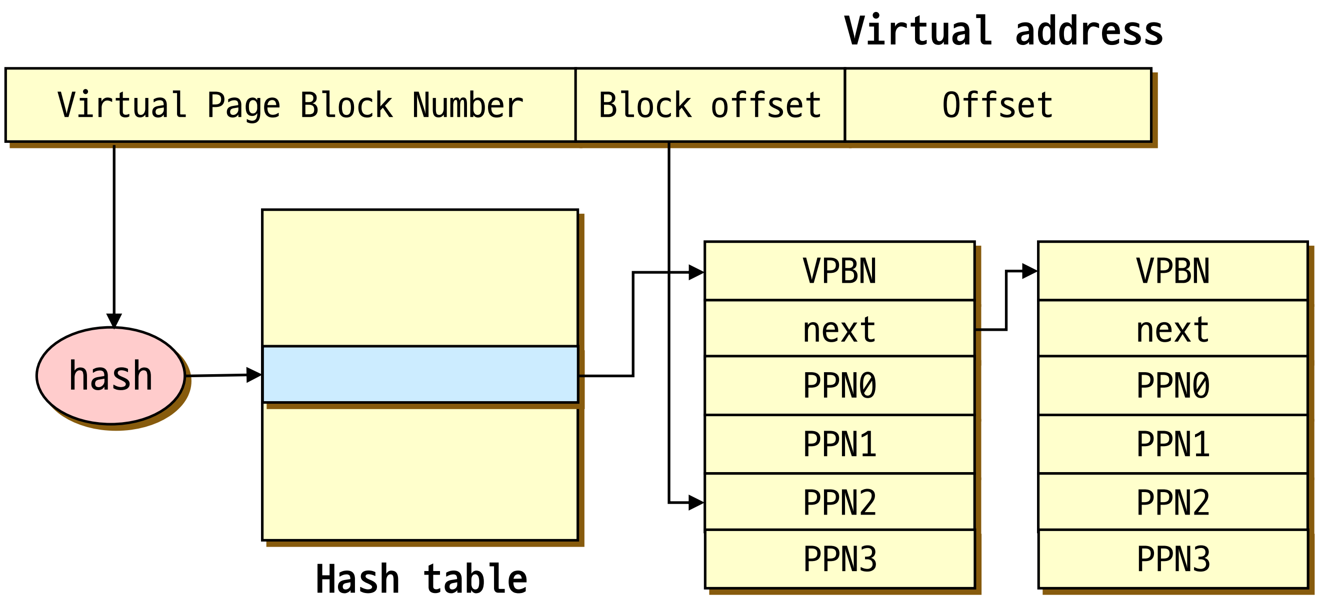
Hashed page tables

- Virtual page number is hashed into the hash table
- Each hash table entry contains a linked list of elements that hash to the same location (in case of collision)
- Each elements contains:
  - The virtual page number
  - The value of the mapped page frame
  - A pointer to the next element in the linked list

Hashed Page Tables (3)

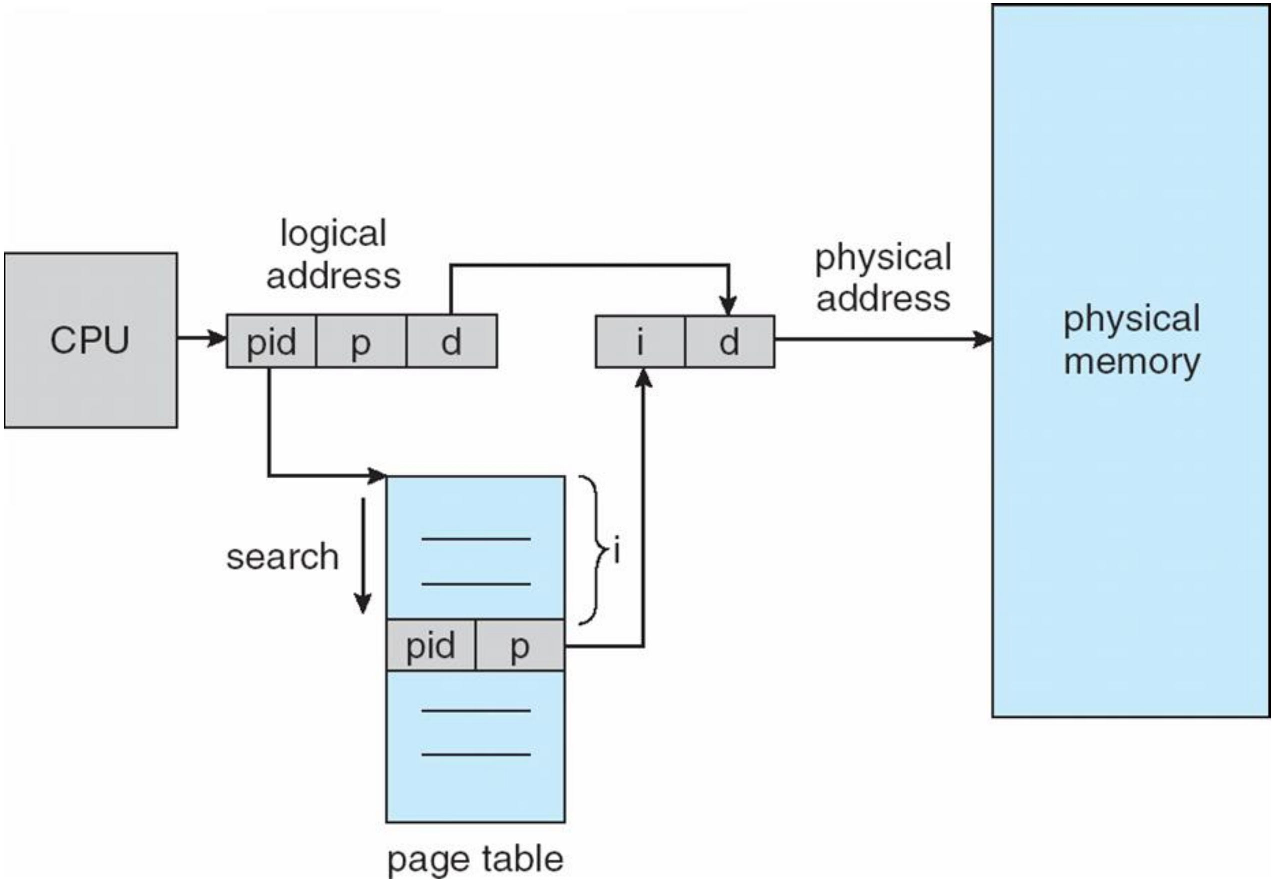
Clustered page tables

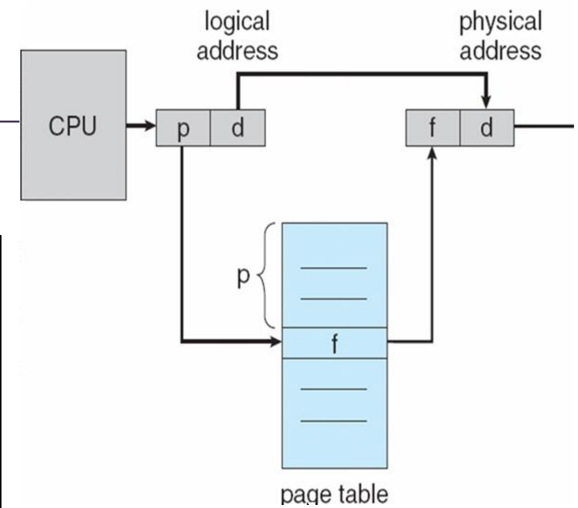
- A variant of hash page tables
- Each entry stores mapping information for a block of consecutive page tables



Inverted Page Tables (1)

Example





Inverted Page Tables (2)

Inverted page tables

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location
  - With information about the process that owns that page
- Have to manage PID
- Decreases memory needed to store each page table
- Increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one, or at most a few, page-table entries

Paging Page Tables

Addressing page tables

- Where are page tables stored? (and which address space?)
- (1) Physical memory
  - Easy to address, no translation required
  - But, allocated page tables consume memory for lifetime of VAS
- (2) Virtual memory (OS virtual address space)
  - Cold (unused) page table pages can be paged out to disk
  - But, addressing page tables requires translation
  - Do not page the outer page table (called wiring)
- Now we've paged the page tables, might as well page the entire OS address space, too
  - Need to wire special code and data (e.g., interrupt and exception handlers)

TLBs (1)

Let's make address translation efficient

Original page table scheme doubled the cost of memory lookups

- One lookup into the page table, another to fetch the data

Two-level page tables triple the cost!

- Two lookups into the page tables, a third to fetch the data
- This assumes the page table is in memory
  - If not, the overhead can be larger

How can we make this more efficient?

- Goal: make fetching from a virtual address as efficient as fetching from a physical address
- Cache the virtual-to-physical translation in hardware
- Translation Lookaside Buffer (TLB, hardware)
  - TLB managed by the Memory Management Unit (MMU, hardware)

TLBs (2)

Translation Lookaside Buffers

- Translate virtual page numbers into PTEs
- Can be done in a single machine cycle

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TLBs (3)

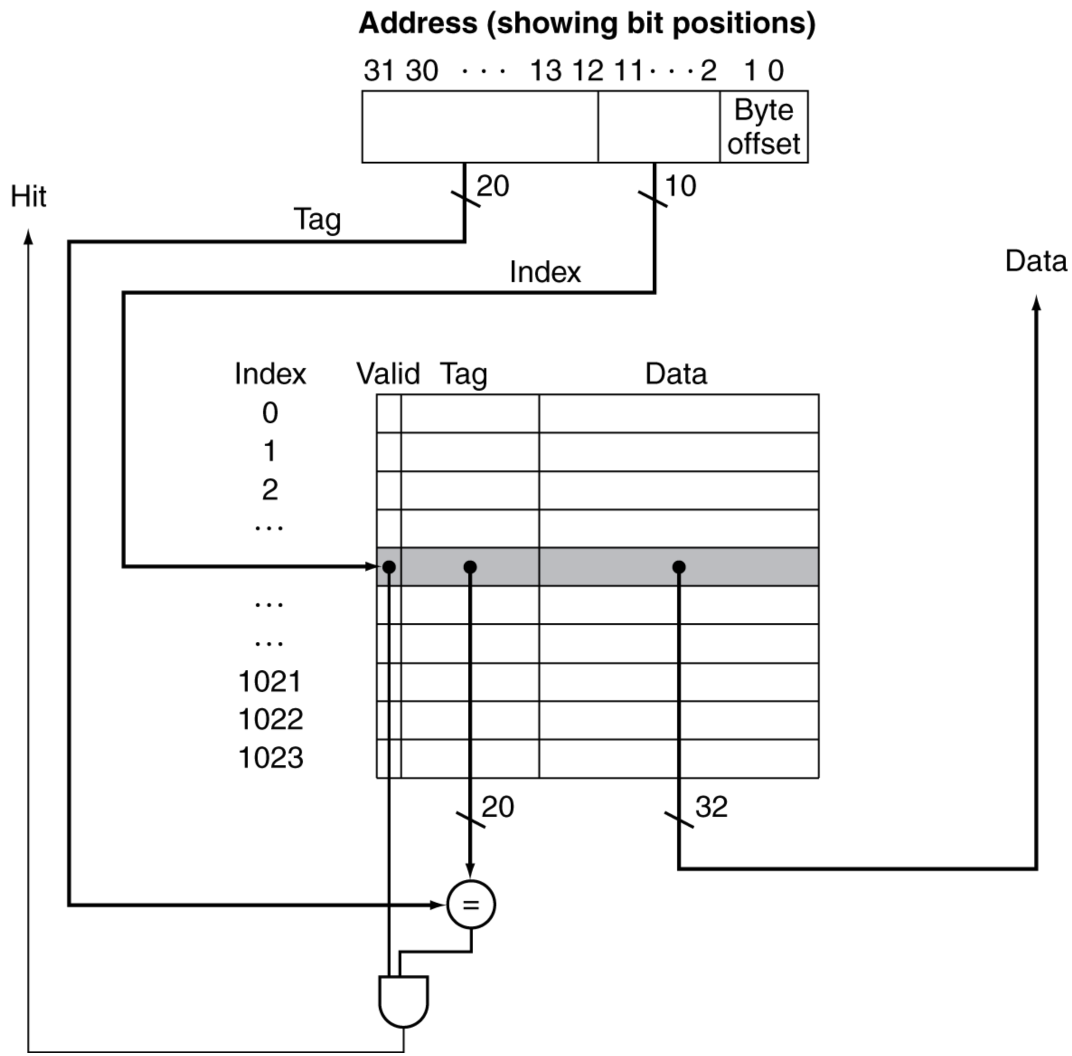
TLB is implemented in hardware

- Fully associative cache (all entries looked up in parallel)
- Cache tags are virtual page numbers
- Cache values are PTEs (entries from page tables)
- With PTE+offset, MMU can directly calculate the physical address

TLBs exploit locality

- Processes only use a handful of pages at a time
  - 16-48 entries in TLB is typical (64-192KB)
  - Can hold the "hot set" or "working set" of process
- Hit rates are therefore really important

Hardware Cache

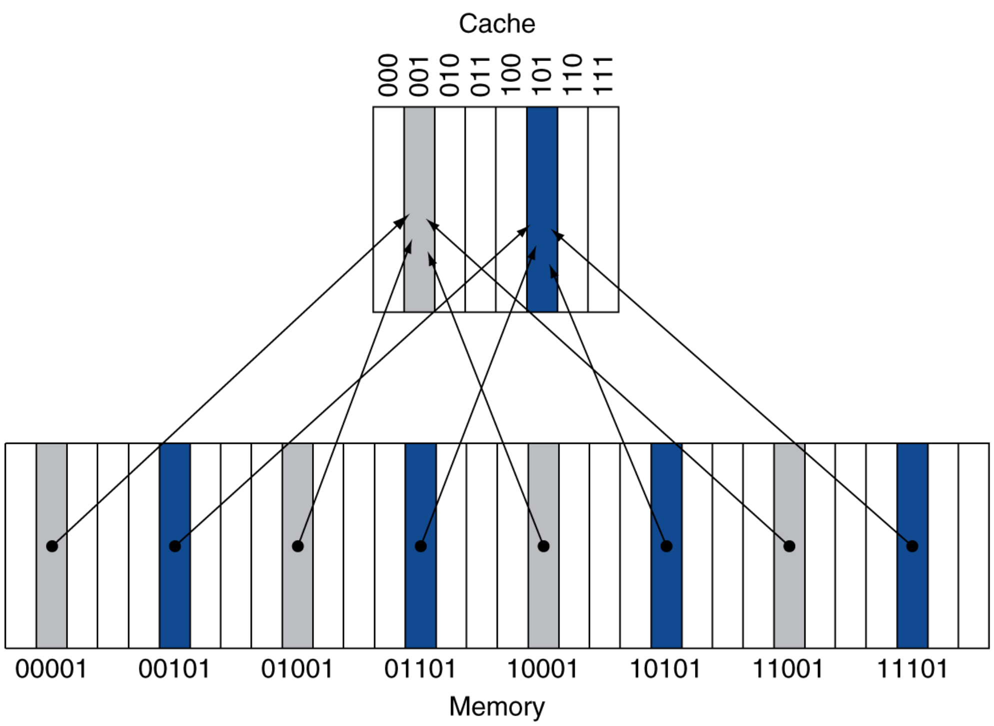


Direct Mapped Cache

Location determined by address

Direct mapped: only one choice

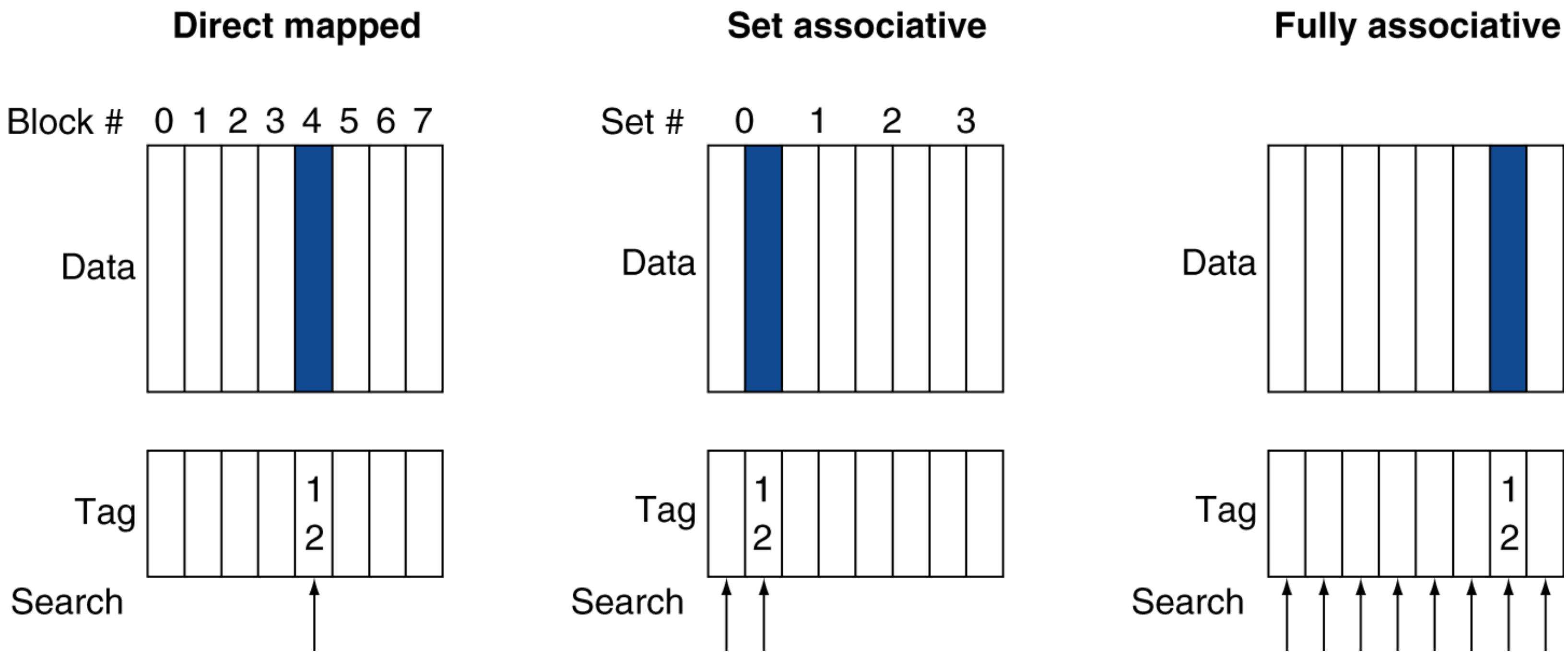
- (Block address) modulo (#Blocks in cache)



#Blocks is a power of 2

Use low-order address bits

Associative Cache Example



How Much Associativity

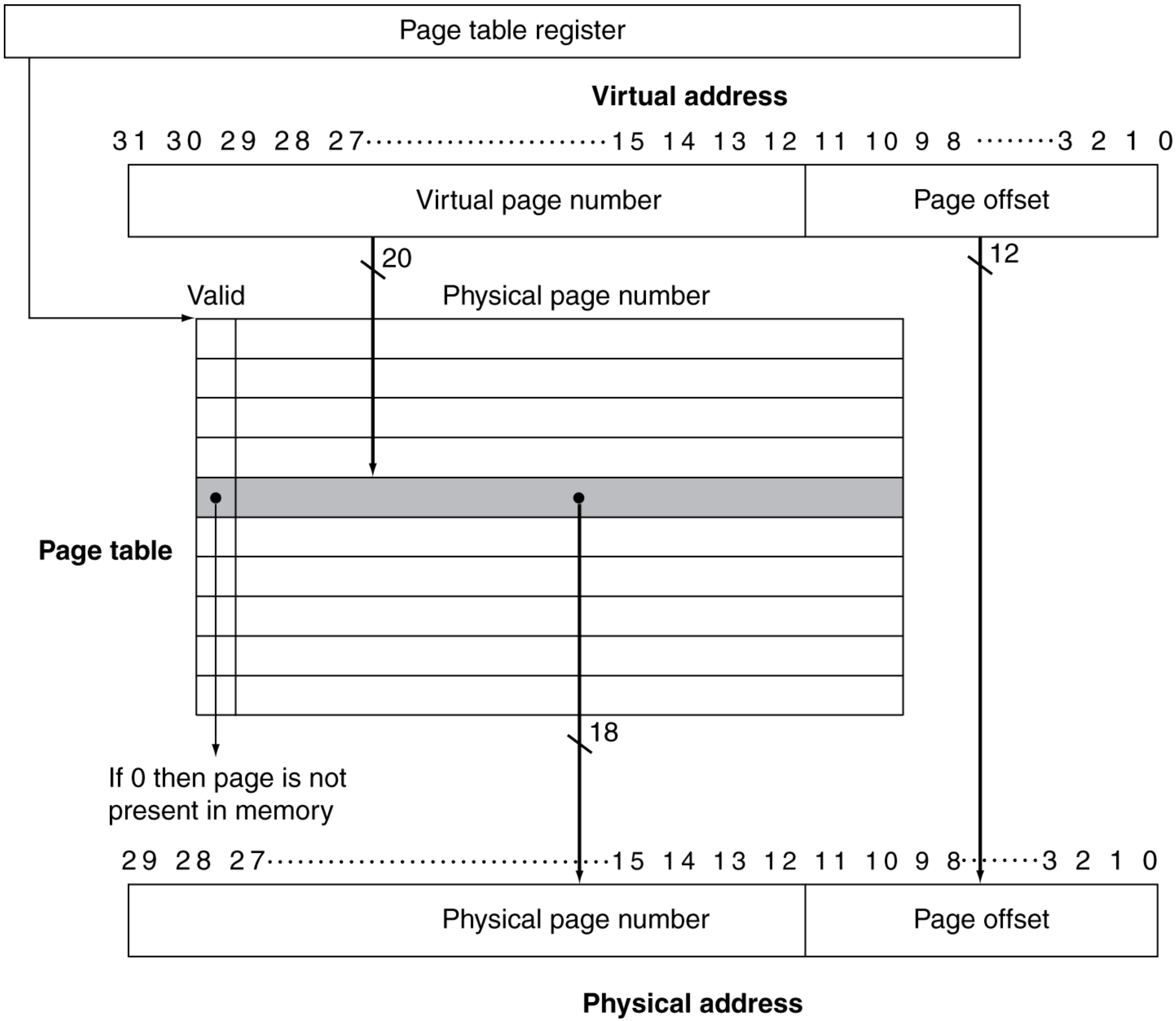
Increased associativity decreases miss rate

- But with diminishing response time

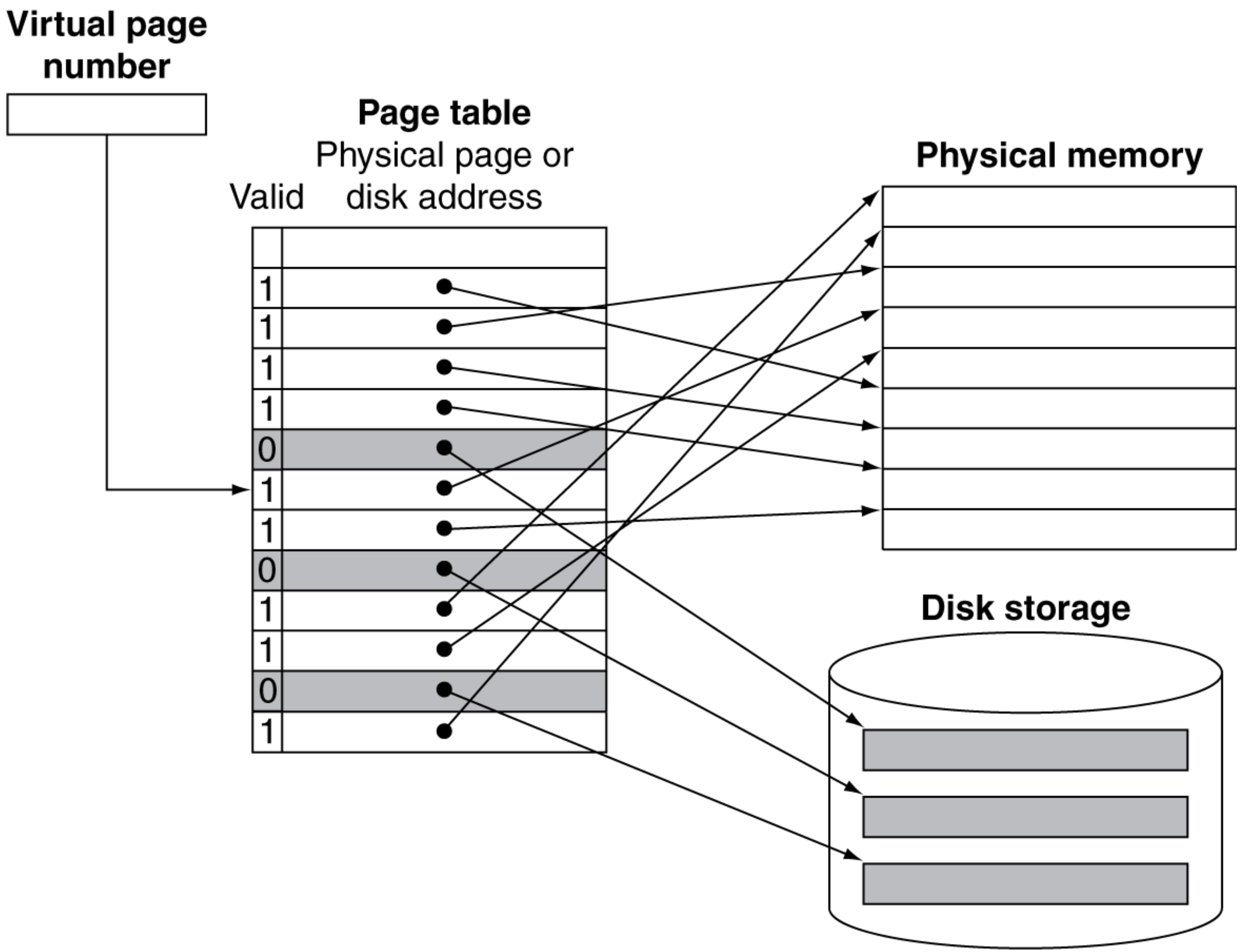
Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000

- 1-way: 10.3%
- 2-way: 8.6%
- 4-way: 8.3%
- 8-way: 8.1%

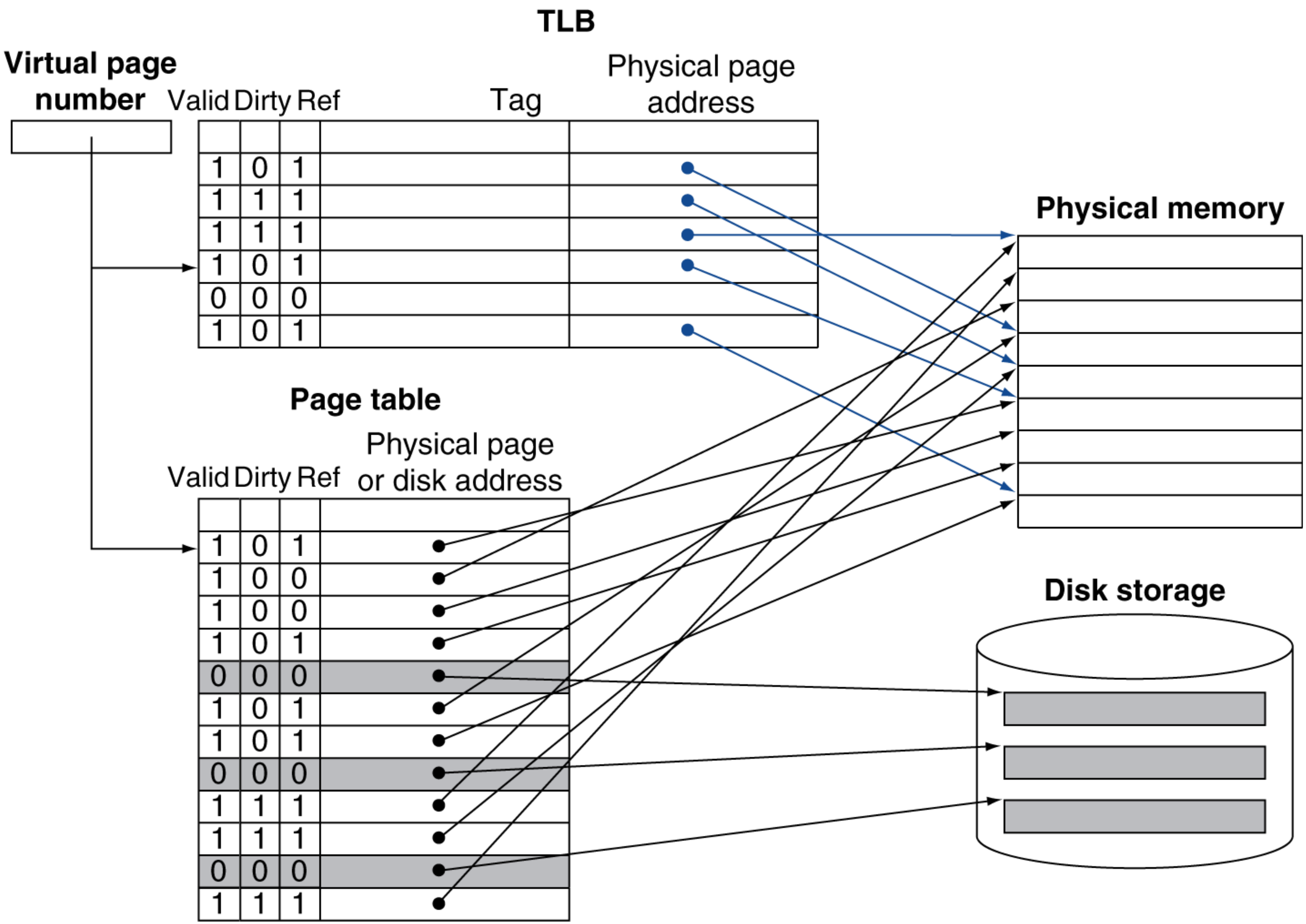
Translation Using a Page Table



Mapping Pages to Storage

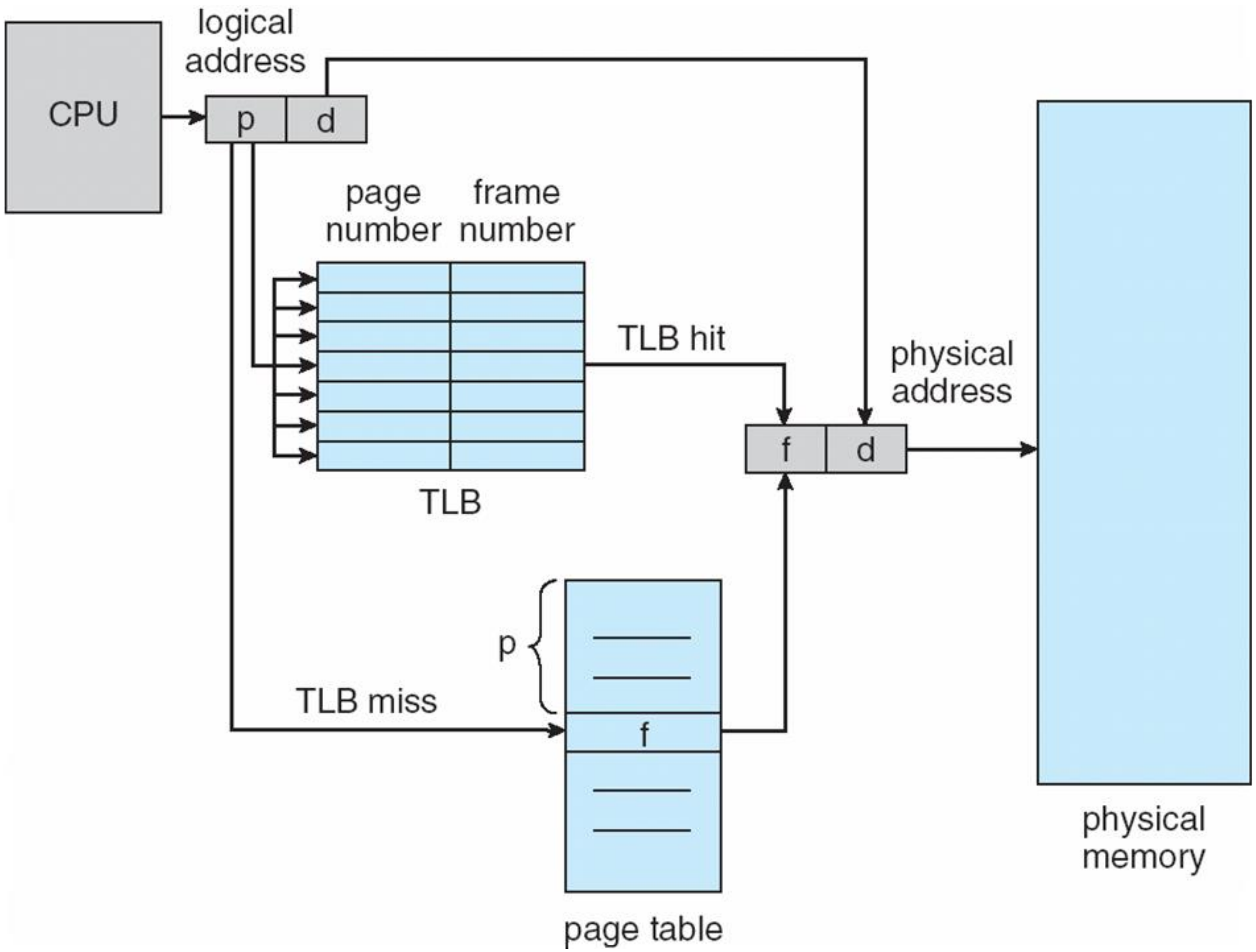


Fast Translation Using a TLB



TLBs (4)

Address translation with TLB



TLBs (5)

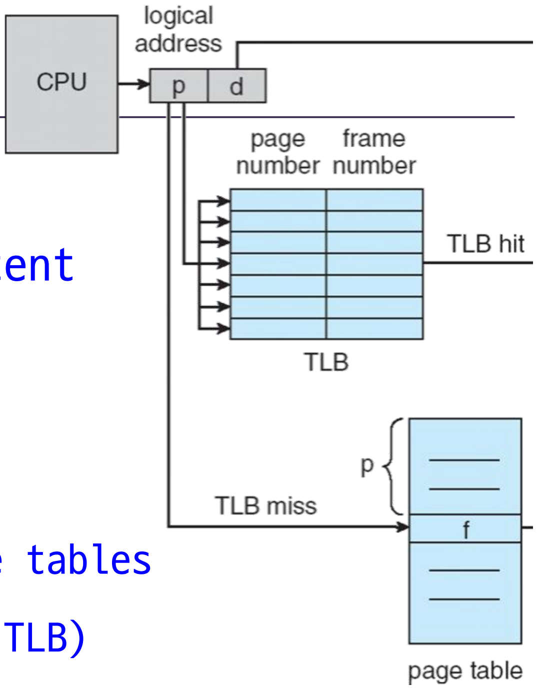
Address translations are mostly handled by the TLB

- More than 99% of translations
- But there are TLB misses occasionally (1%)

Handling TLB misses

- In case of a miss, who places translations into the TLB?
- Hardware (MMU): Intel x86
  - Knows where page tables are in memory
  - HW access them directly
  - Page tables have to be in hardware-defined format
- Software loaded TLB (OS)
  - TLB miss faults to OS, OS finds right PTE and loads TLB
    - Must be fast (but, 20-200 cycles typically)
    - CPU ISA has instructions for TLB manipulation
  - Page tables can be in any format convenient for OS (flexible)

TLBs (6)



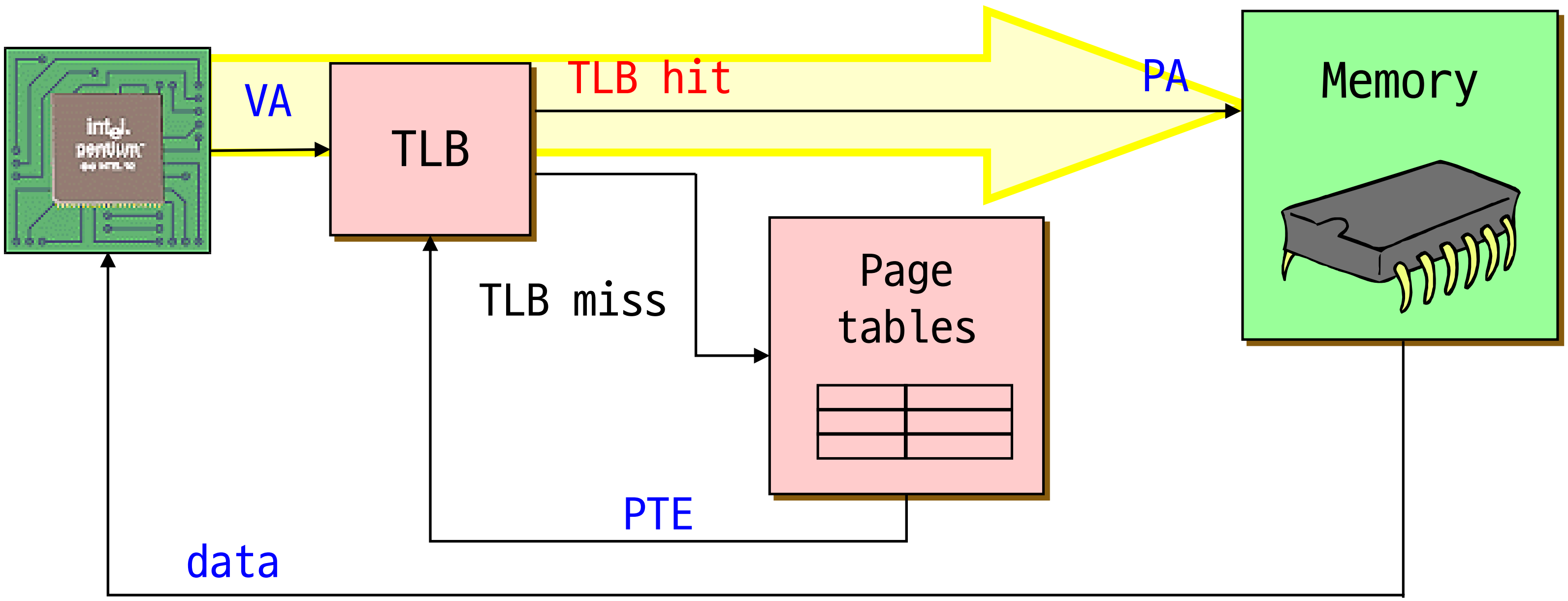
Managing TLBs

- OS ensures that TLB and page tables are consistent
  - Something is changed in page tables, the TLB entry should be invalidated
- Reload TLB on a process context switch
  - Remember, each process typically has its own page tables
  - Need to invalidate all the entries in TLB (flush TLB)
  - In IA-32, TLB is flushed automatically when the contents of CR3 (page directory base register) is changed
  - (cf.) Alternatively, we can store the PID as part of the TLB entry, but this is expensive
- When the TLB misses, and a new PTE is loaded, a cached PTE must be evicted
  - Choosing a victim PTE called the "TLB replacement policy"
  - Implemented in hardware, usually simple (e.g., LRU)

Memory Reference (1)

Situation

- Process is executing on the CPU, and it issues a read to a virtual address



Memory Reference (2)

The common case (TLB hits, more than 99%)

- The read/write goes to the TLB in the MMU
- TLB does a lookup using the page number of the address
- The page number matches, returning a PTE
- TLB validates that the PTE protection allows reads/writes
- PTE specifies which physical frame holds the page
- MMU combines the physical frame and offset into a physical address
- MMU then reads from that physical address, returns value to CPU

Memory Reference (3)

TLB misses: two implementation choices

- (1) MMU loads PTE from page table in memory
  - Hardware managed TLB, OS not involved in this step
    - OS has already set up the page tables so that the hardware can access it directly
- (2) Trap to the OS
  - Software managed TLB, OS intervenes at this point
    - OS does lookup in page tables, loads PTE into TLB
    - OS returns from exception, TLB continues

- After handling TLB misses, there is a valid PTE for the address in the TLB
- So the requested address is referred as a TLB hit case

Memory Reference (4)

TLB misses: recursive fault

- Page table lookup (by HW or OS) can cause a recursive fault if page table is paged out
  - Assuming page tables are in OS virtual address space
  - Page fault handler loads page table into physical memory
  - Load PTE into TLB
- When TLB has PTE, it restarts translation
  - Common case is that the PTE refers to a valid page in memory
  - Uncommon case is that TLB faults again on PTE
    - e.g., page is invalid

Memory Reference (5)

Page faults

- PF can be two cases
  - Read/Write/Execute – operation not permitted on page (protection fault)
  - Invalid – virtual page not allocated, or page not in physical memory
- TLB traps to the OS (software takes over)
  - Read/Write/Execute – OS usually will send fault back to the process, or might be playing tricks
    - e.g., copy on write, mapped files
  - Invalid (Not allocated) – OS sends fault to the process
  - Invalid (Not in physical memory) – OS allocates a frame, reads from disk, and maps PTE to physical frame (page fault handling)

Memory Reference Summary

1. TLB hit
   - Frame is in memory
   - Frame PF (generally not possible)
2. TLB miss – page table is in memory
   - Update TLB, restart
   - Frame is in memory / Frame PF
3. TLB miss – page table is paged out (PF)
   - PF handler for page table
   - Update TLB, restart
   - Frame is in memory / Frame PF