Modern Operating Systems

Chapter 3 – Paging

Zhang Yang

Autumn 2022

Content of the Lecture

- 3.3 Virtual Memory
  - Virtual Address
  - Paging
  - Address Translation Scheme
  - Page Table
  - Translation Lookaside Buffer
  - Multi-level Page Table
  - Inverted Page Table

Virtual Memory (1)

- How to Manage Bloatware?
  - Run programs that are too large to fit in memory.
  - Have systems that can support multiple programs running simultaneously.
  - Overlays?
  - Swapping?
- Need for Virtual Memory
  - Separation of user logical memory from physical memory.
  - Only *PART* of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Need to allow pages to be swapped in and out.

Virtual Memory (2)

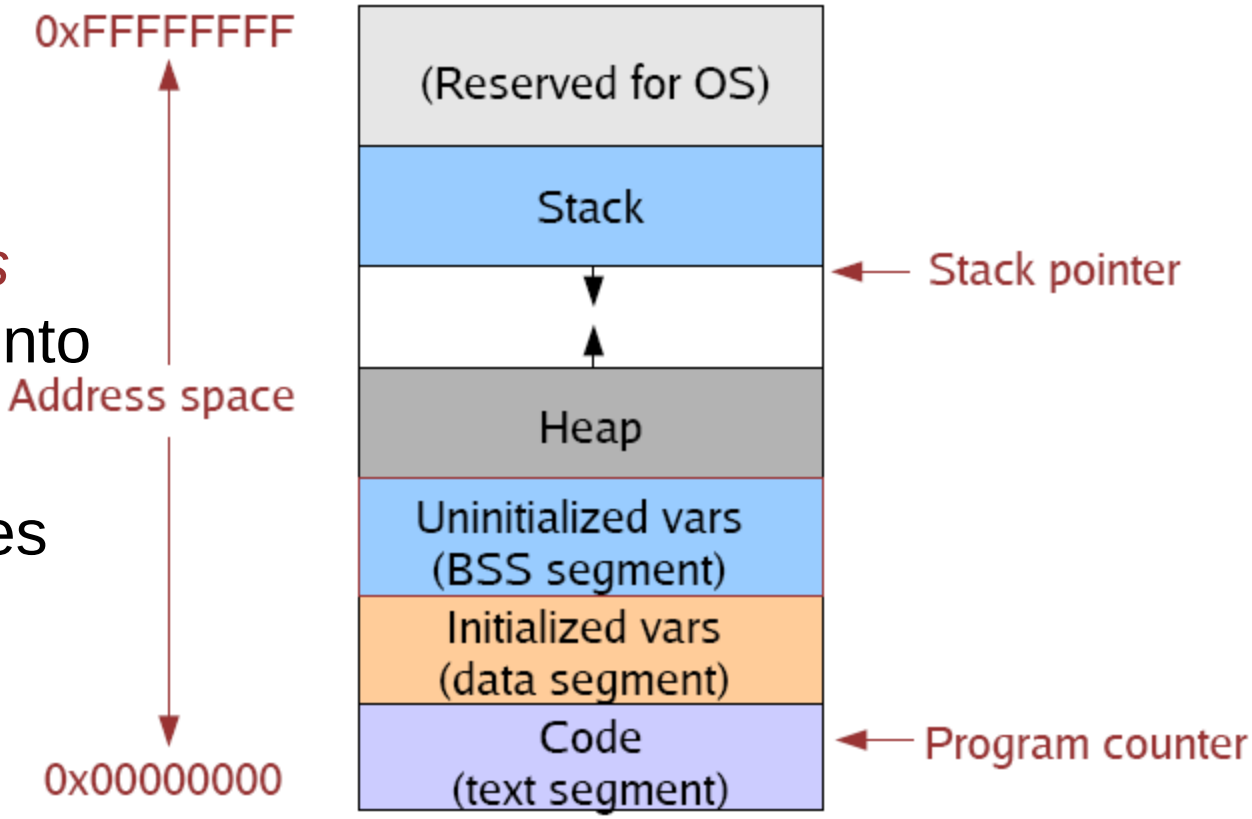
- Implementation of Virtual memory
  - Paging
    - Modern approach
    - Paging is presently most common.
  - Segmentation
    - Basic early approach
    - Typically very simple
    - Probably better suited to behavior of process, although it can be harder to use and harder to implement.
  - Combined paging and segmentation

Virtual Memory (3)

- Support needed for Virtual Memory
  - Hardware Support
    - Memory Management Unit (MMU)
    - Translation Lookaside Buffer (TLB)
  - OS Support
    - Virtual memory system to control the MMU and TLB.

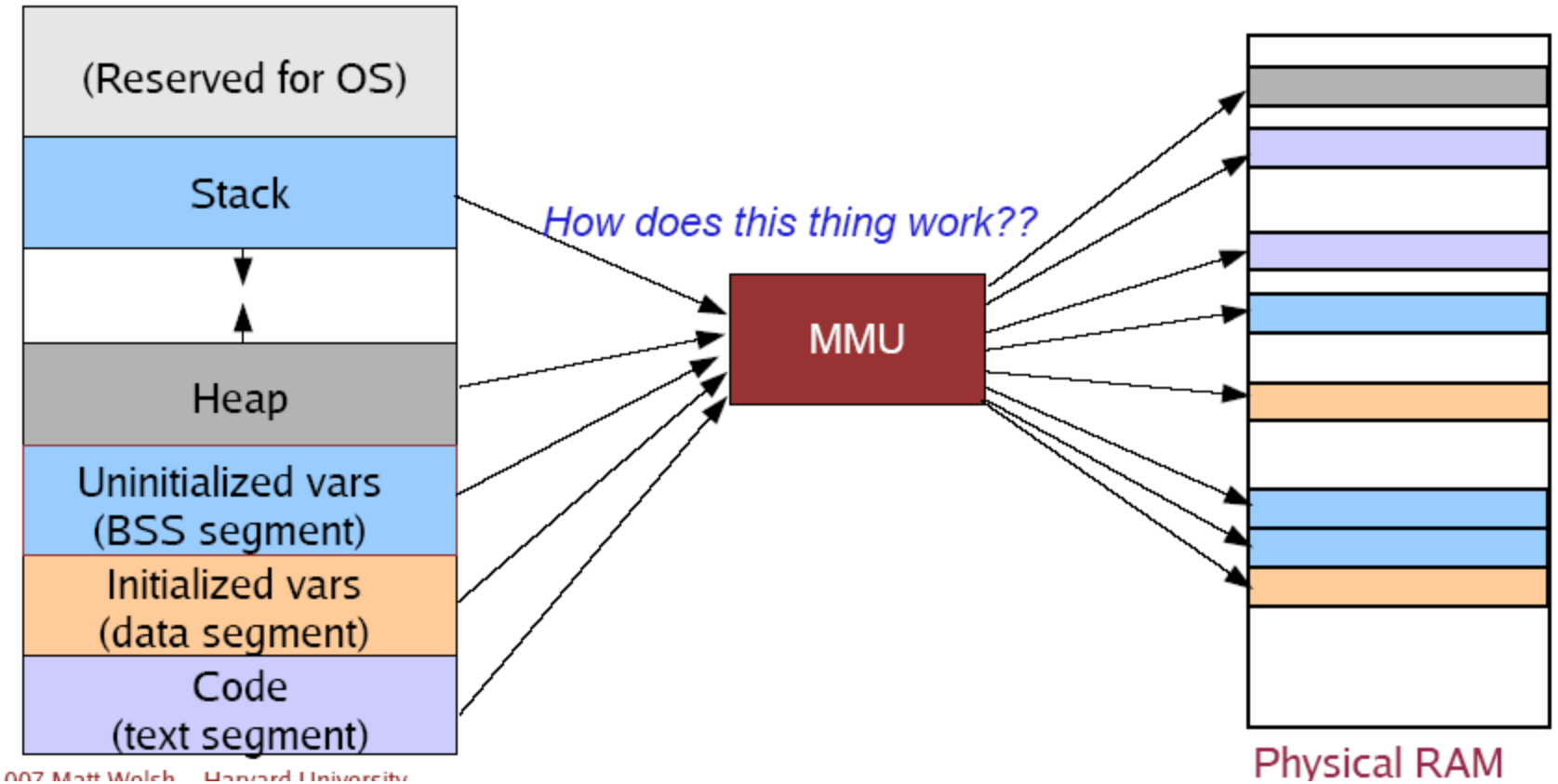
Virtual Address (1)

- A *virtual address* is a memory address that a process uses to access its own address space.
  - The virtual address is *not the same* as the physical RAM address in which it is stored.
  - When a process accesses a virtual address, the MMU hardware *translates* the virtual address into a physical address
  - The OS determines the *mapping* from virtual address to physical address.



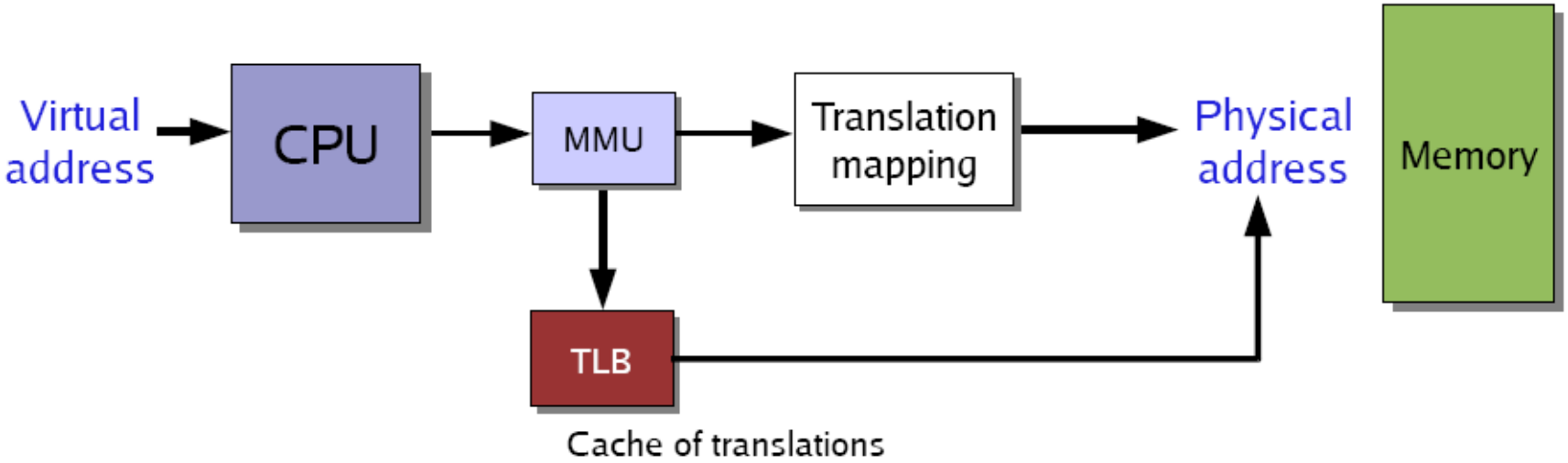
Virtual Address (2)

- Virtual Addresses allow isolation
- Virtual Addresses allow relocation



MMU and TLB

- MMU
  - Hardware that translates a virtual address to a physical address.
  - Each memory reference is passed through the MMU.
- TLB
  - Cache for MMU virtual-to-physical address translations
  - Just an optimization, but an important one!

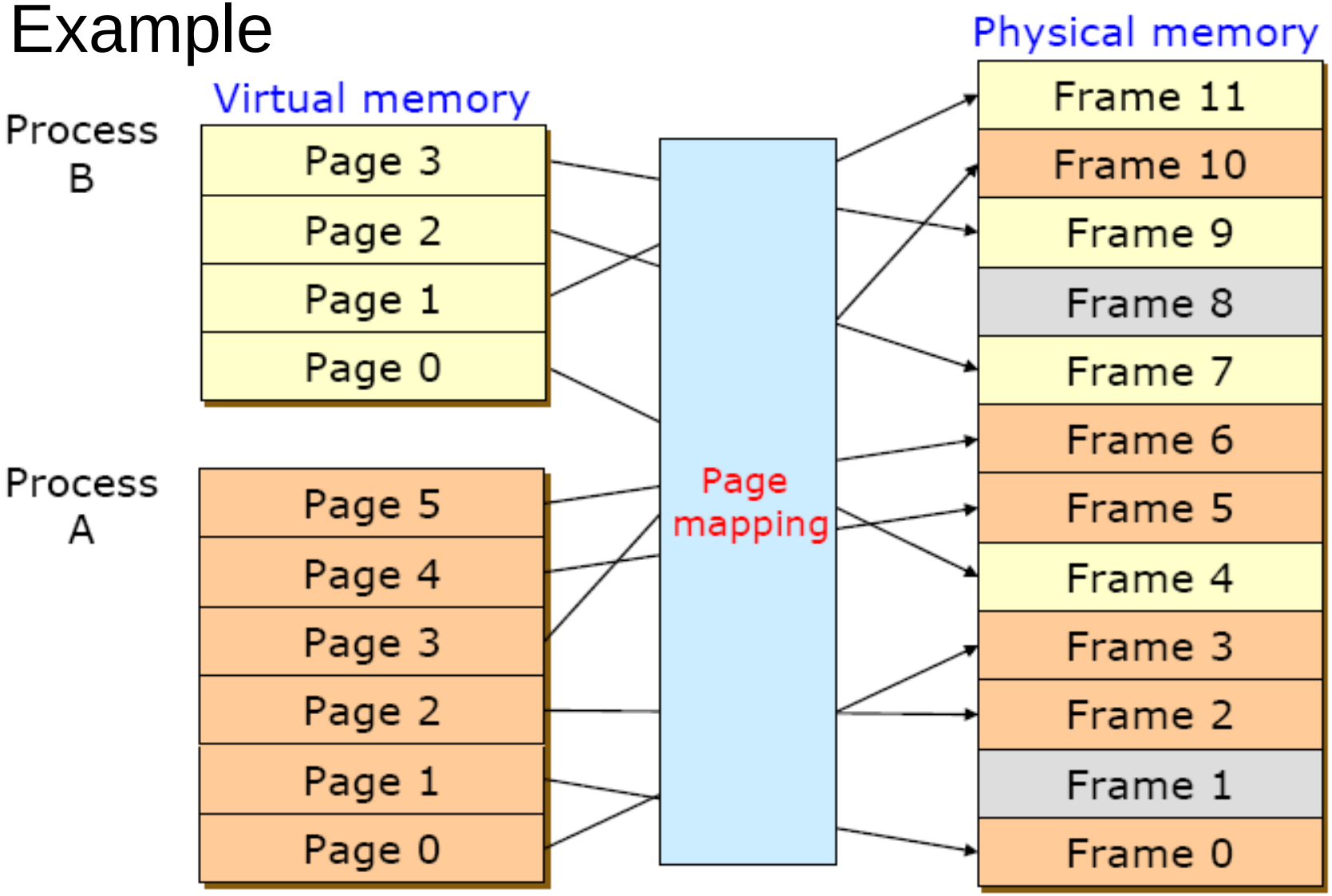


Paging (1)

- Physical Memory
  - Divided into page frames (fixed-sized blocks)
  - Size is power of 2, e.g. 512 bytes –64KB
- Logical Memory
  - Divided into pages
  - The same size as page frames.

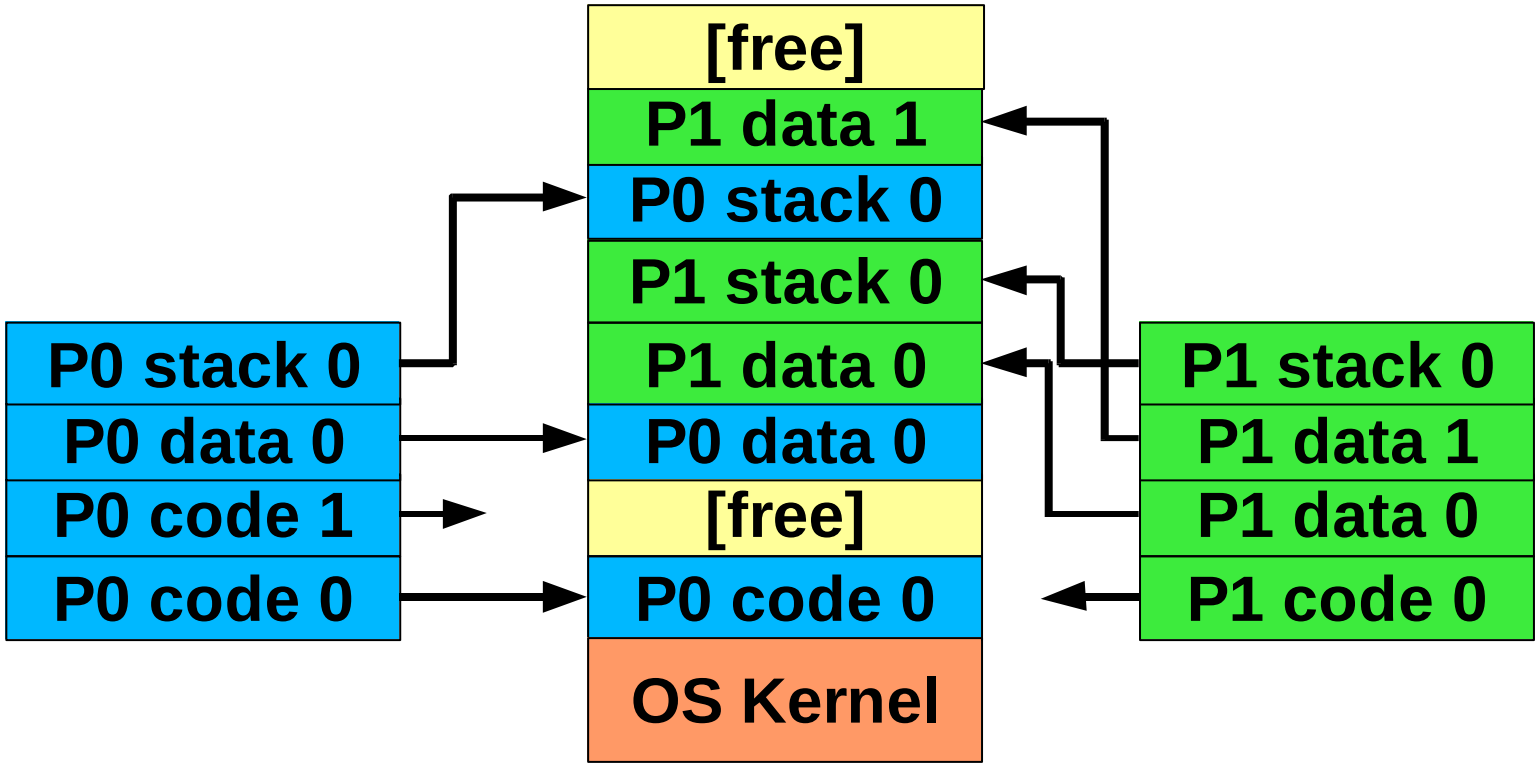
Paging (2)

- Example



Paging (3)

- Partial Residence Example



Paging (4)

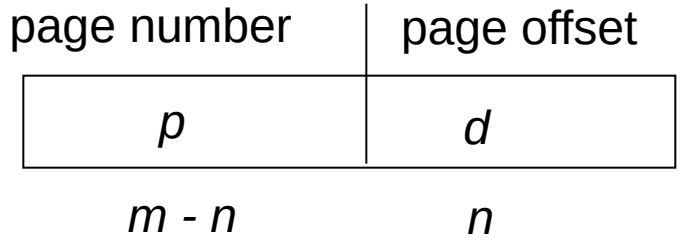
- Operating System Responsibilities
  - Maintain the page table.
    - A Page table is an array that translates from pages to frames.
  - Allocate sufficient pages from free frames to execute a program.
    - To run a program of size n pages, need to find n free page frames and load program.
  - OS keeps track of all free frames.
    - Set up a page table to translate logical to physical addresses.

Paging (5)

- Advantages
  - Permits the physical address space of a process to be noncontiguous and be allocated as needed.
  - Easy to allocate physical memory.
  - Easy to “page out”/swap chunks of memory.
- Issue
  - Internal Fragmentation

Address Translation Scheme (1)

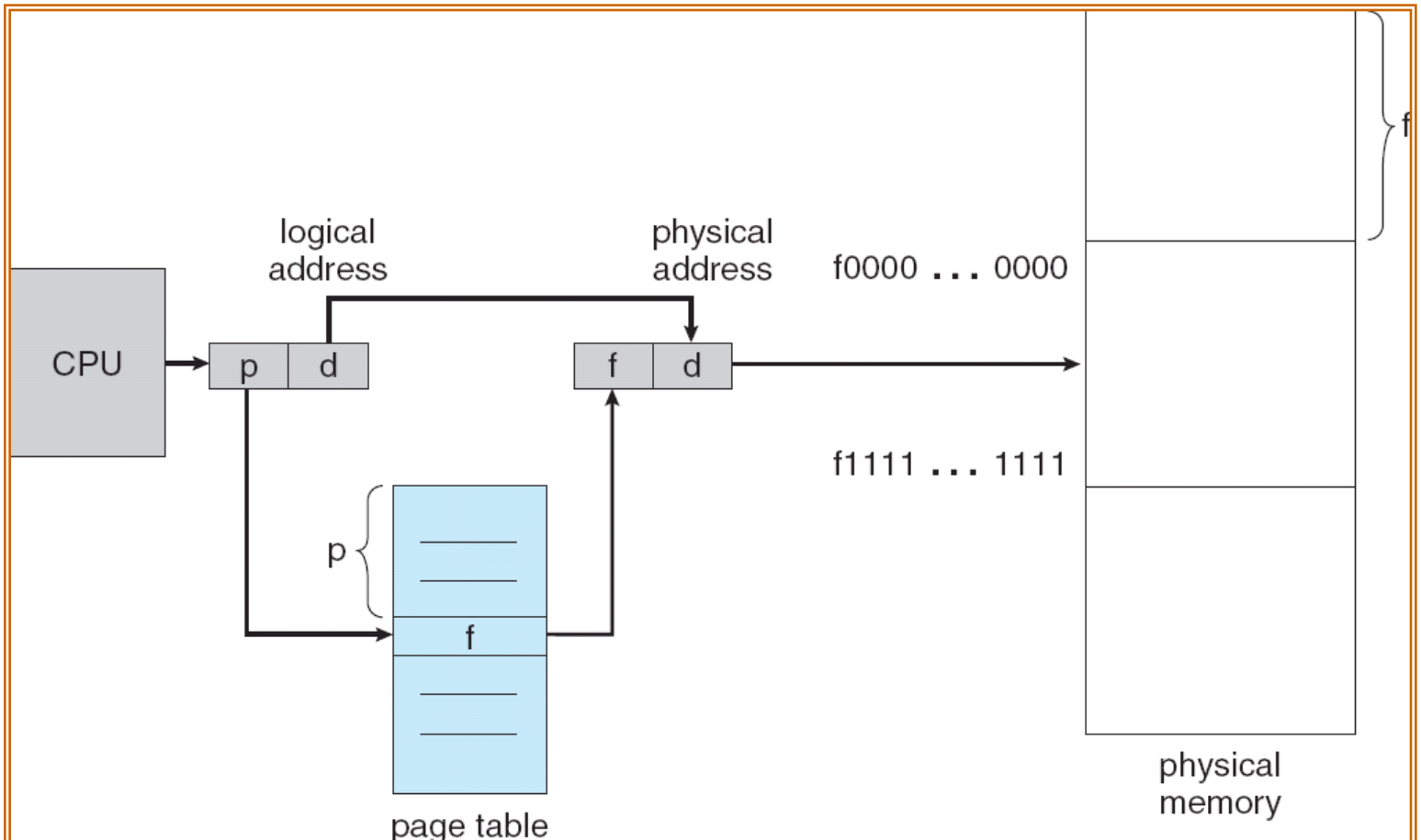
- Address generated by CPU is divided into
  - For given logical address space 2^m and page size 2^n



  - Page number (p)
    - Used as an **index** into the page table
    - Page table contains **base address** of each page in physical memory
  - Page offset (d)
    - Combined with base address to define the physical memory address sent to the memory unit.

Address Translation Scheme (2)

- Physical Address
  - Frame number (f)
  - Page offset (d)



Address Translation Scheme (3)

- Example
  - Internal operation of MMU with 16 4KB pages

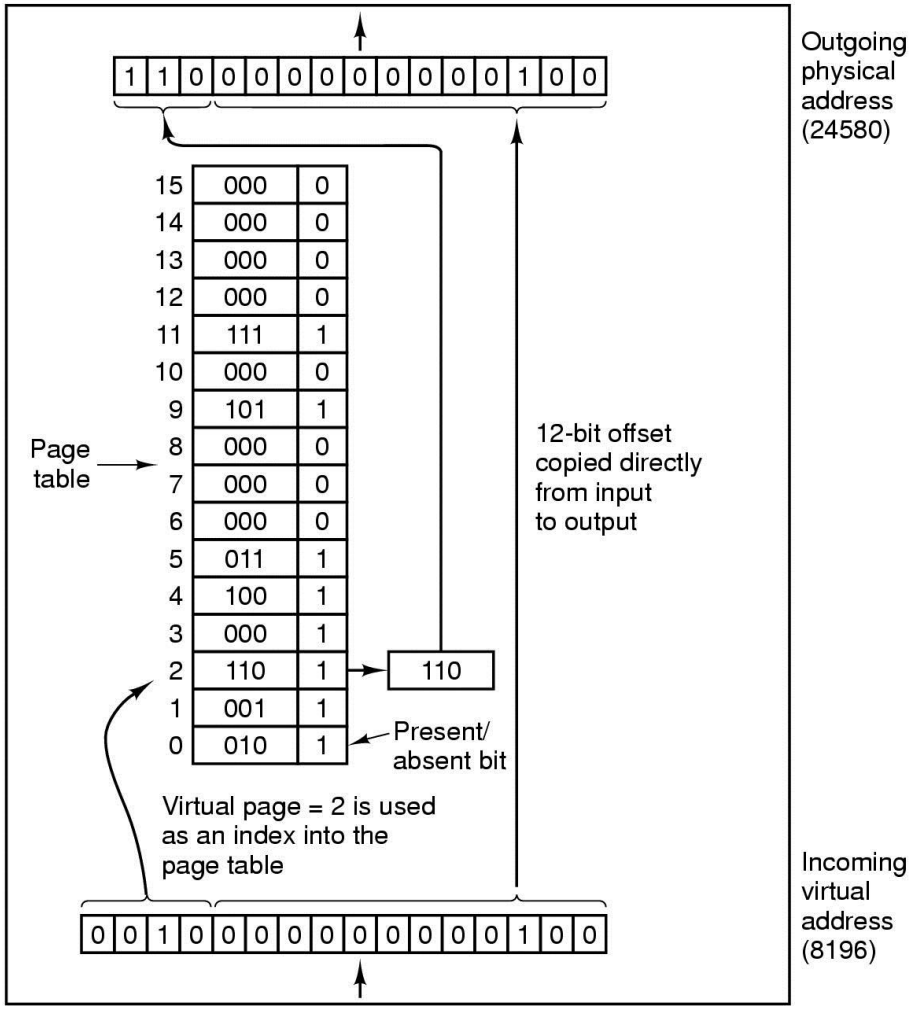
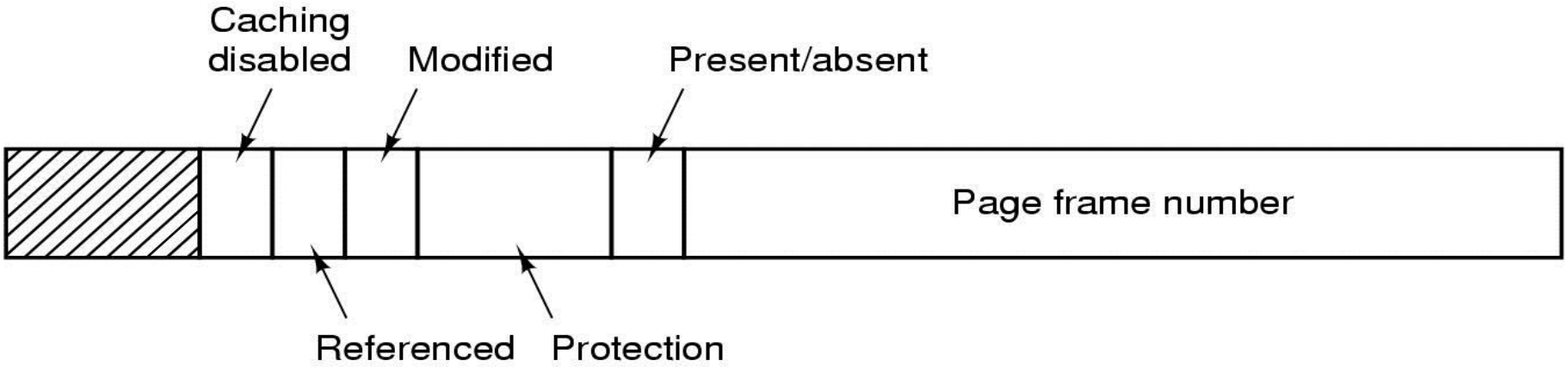


Figure 3-10

Page Tables

- Managed by OS.
- Map VPN (Virtual Page Number) to PFN
  - VPN is the index into the table that determines PFN
- One page table entry (PTE) per page in virtual address space, i.e. one PTE per VPN.
- Most operating systems allocate a page table for each process.

Structure of Page Table Entry



- Page frame number: Map the frame number
- Present/absent bit: 1/0 indicates valid/invalid entry
- Protection bit: What kinds of access are permitted.
- Modified (dirty bit): Set when modified and writing to the disk occur
- Referenced: Set when page is referenced (help decide which page to evict)
- Caching disabled - Cache is used to keep data that logically belongs on the disk in memory to improve performance.

Page Tables: Design Issues (1)

- Fast Access Issue
  - Often need to make 2 or more page table references per instruction, so fast access is necessary!
- Page Table for Large Memories Issue
  - Average waste due to internal fragmentation is ½ page per process, so want small page size
  - But, small page size → many pages →

  large page table → page tables (for each process) will take up a lot of memory space

Page Tables: Design Issues (2)

- Solution to Fast Access Issue
  - TLB : translation look-aside buffer
- Solution to Page Table for Large Memories
  - Multi-level Page Tables
  - Inverted Page Tables

Translation Lookaside Buffer (1)

- Each virtual address reference requires 2 physical memory references.
  - 1:To page table to get mapping
  - 2:To page frame itself
- Eliminate #1 by caching recent mappings in translation lookaside buffer (TLB)
  - Take advantage of fact that most programs reference small number of pages often.
  - TLB is part of MMU.
  - Each TLB entry is a page table entry (has same elements as PTE).
  - TLB is a special hardware (called associative memory) – searches all PTE's in TLB at once in parallel.
  - Approximately 64-256 PTE's in TLB.

Translation Lookaside Buffers (2)

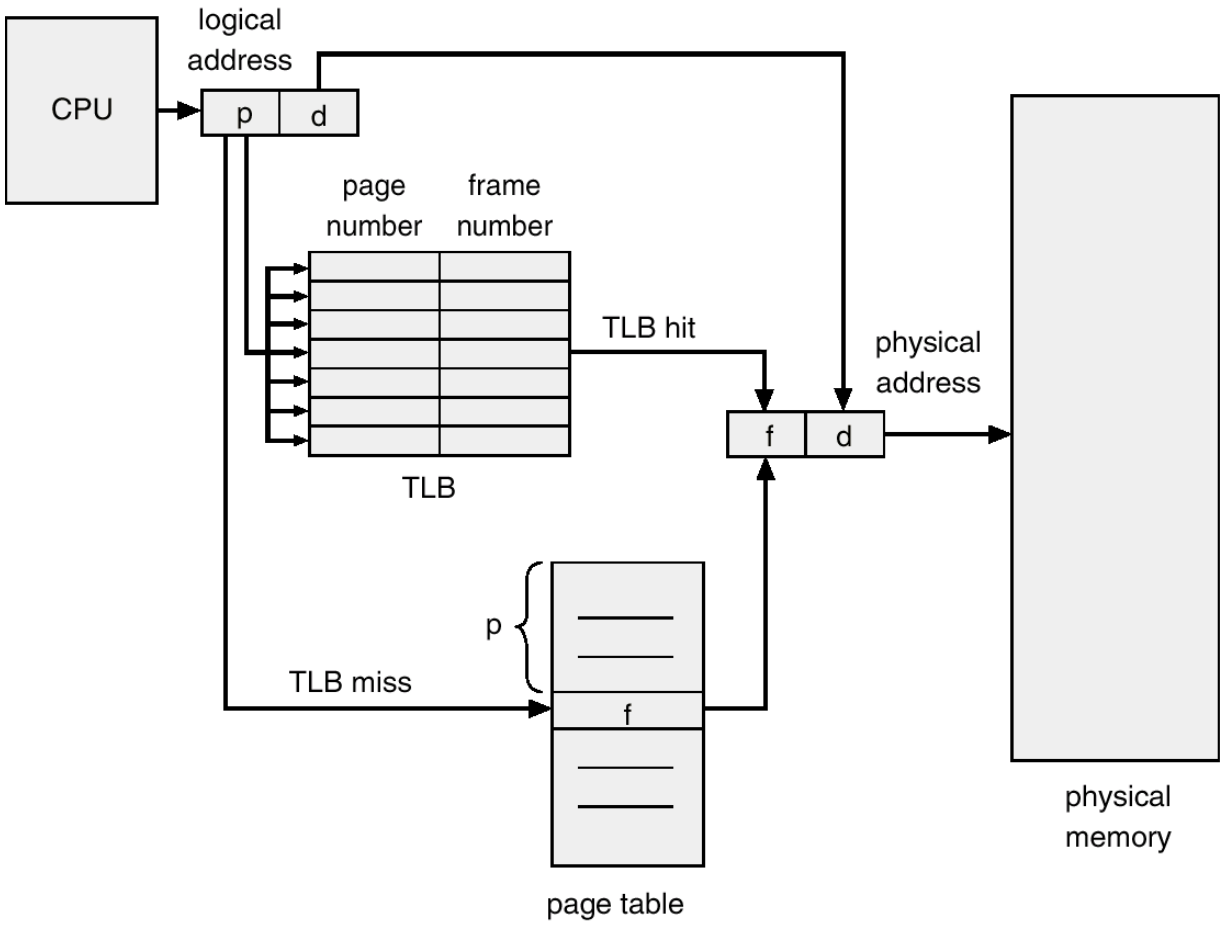
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 3-12. A TLB to speed up paging.

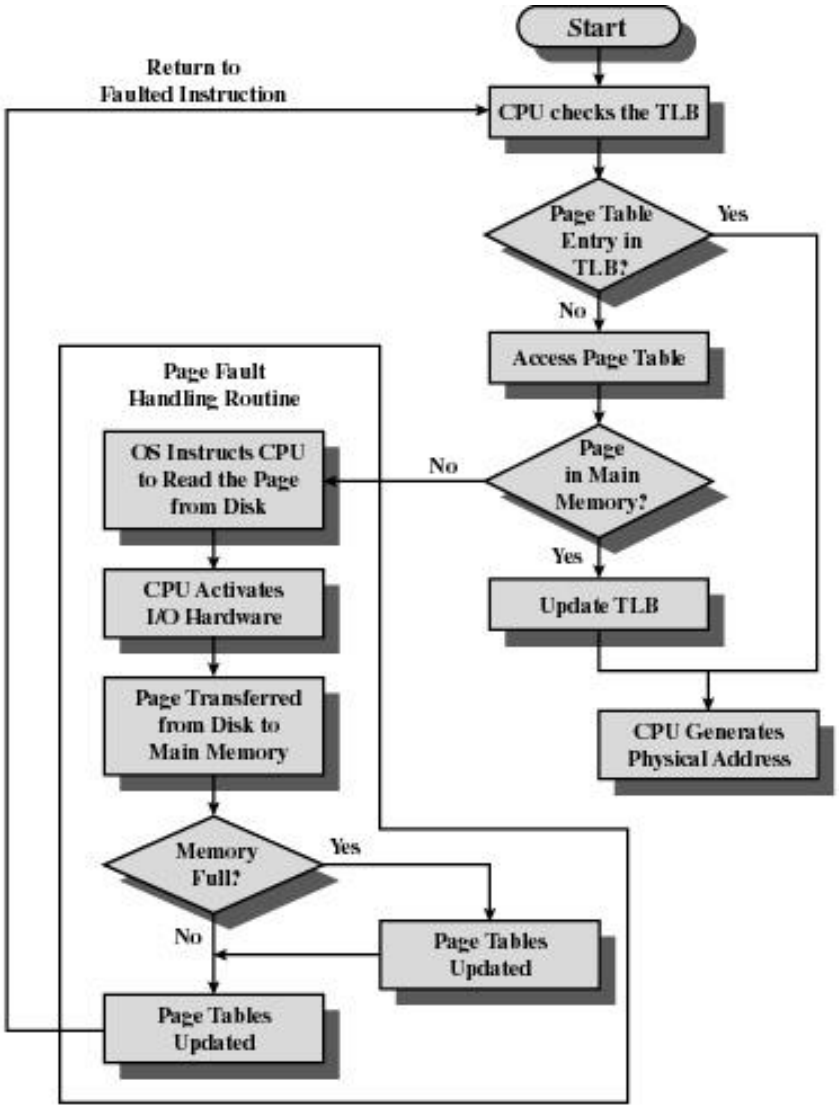
Translation Lookaside Buffer (3)

- If a virtual address is presented to MMU, the hardware checks TLB by comparing all entries simultaneously (in parallel).
- If match is valid, the page is taken from TLB without going through page table.
- If match is not valid
  - MMU detects miss and does an ordinary page table lookup.
  - It then evicts one page out of TLB and replaces it with the new entry, so that next time that page is found in TLB.

Translation Lookaside Buffer (4)



Translation Lookaside Buffer (5)



Hardware TLB Management

- On a TLB miss
  - Hardware loads the PTE into the TLB.
    - Need to write back if there is no free entry.
  - Generate a fault if the page containing the PTE is invalid.
  - OS performs fault handling.
- On a TLB hit, hardware checks the valid bit.
  - If valid, pointer to page frame in memory.
  - If invalid, the hardware generates a page fault.
    - Perform page fault handling.
    - Restart the faulting instruction.

Software TLB Management

- RISC machines (SPARC, MIPS, Alpha) do page management in software. If virtual address is not in the TLB, TLB fault occurs and is passed to OS.
- On a miss in TLB, OS
  - Write back if there is no free entry.
  - Check if the page containing the PTE is in memory.
  - If no, perform page fault handling.
  - Load the PTE into the TLB.
  - Restart the faulting instruction.
- Soft Miss
  - A soft miss occurs when the page referenced is not in the TLB, but is in memory.
- Hard Miss
  - A hard miss occurs when the page itself is not in memory (and of course, also not in the TLB).

Hardware vs. Software TLB Management

- Hardware Approach
  - Efficient
  - Inflexible
  - Need more space for page table.
- Software Approach
  - Flexible
  - Software can do mappings by hashing.
    - PP# $\rightarrow$ (Pid, VP#)
    - (Pid, VP#) $\rightarrow$ PP#
  - Can deal with large virtual address space.

Recall: Page Tables Design Issues

- Solution to Fast Access Issue
  - TLB : translation look-aside buffer
- Solution to Page Table for Large Memories
  - Multi-level Page Tables
  - Inverted Page Tables

Multi-level Page Tables

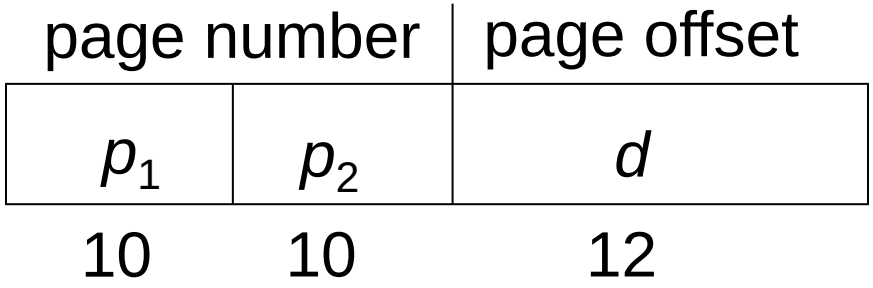
- Suppose
  - 4KB (2^{12}) page size,64-bit address space,8-byte PTE
  - Would need a 32,000 TB page table!
    - $(2^{64} / 2^{12}) * 2^3 = 2^{55}$ bytes
- Since the page table can be very large, one solution is to page the page table.
- Tree of Page Tables
  - Lowest-level tables hold PTEs (page table entries).
  - Upper-level tables hold pointers to lower-level tables.
  - Different parts of VPN used to index different levels.

Two-Level Paging (1)

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - A page number consisting of 20 bits.
  - A page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - A 10-bit page number.
  - A 10-bit page offset.

Two-Level Paging (2)

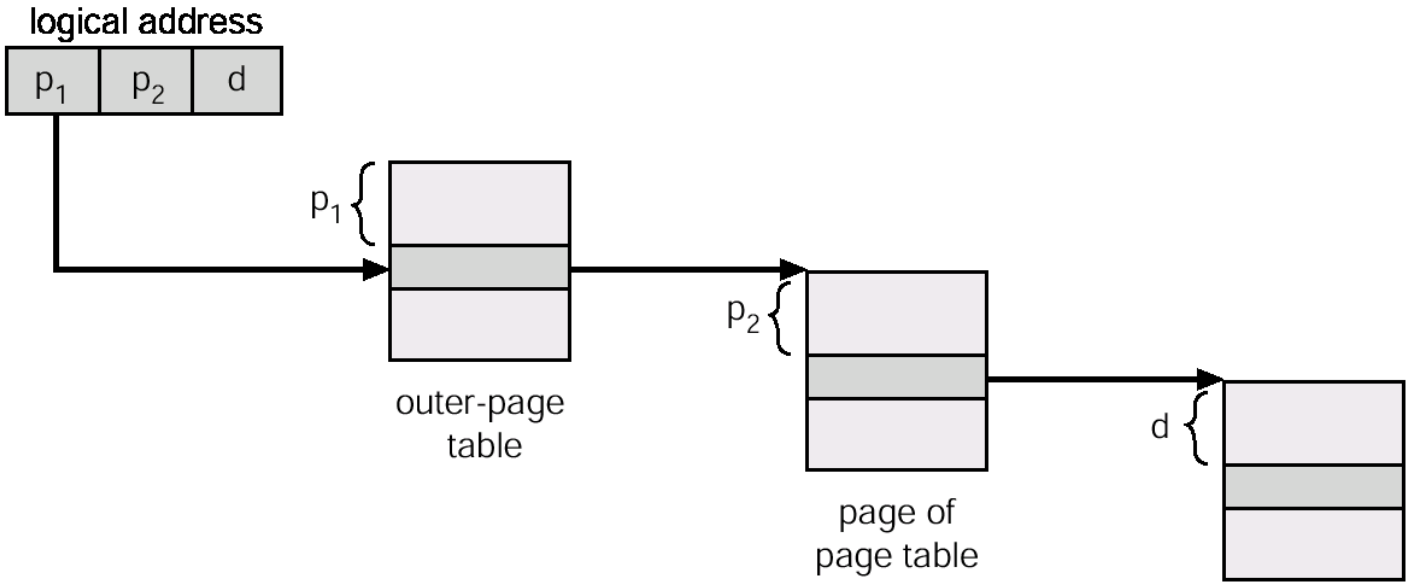
- Thus, a logical address is as follows:



- Where p1 is an index into the outer page table, and p2 is the displacement within the page of the outer page table.

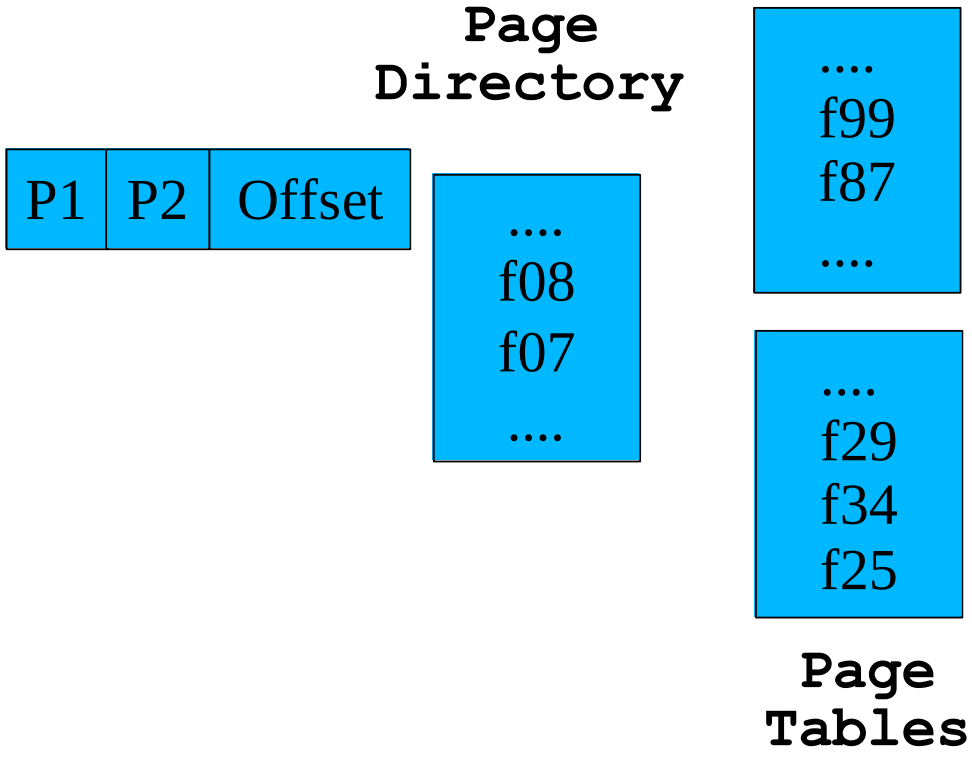
Two-Level Paging (3)

- Address Translation Scheme



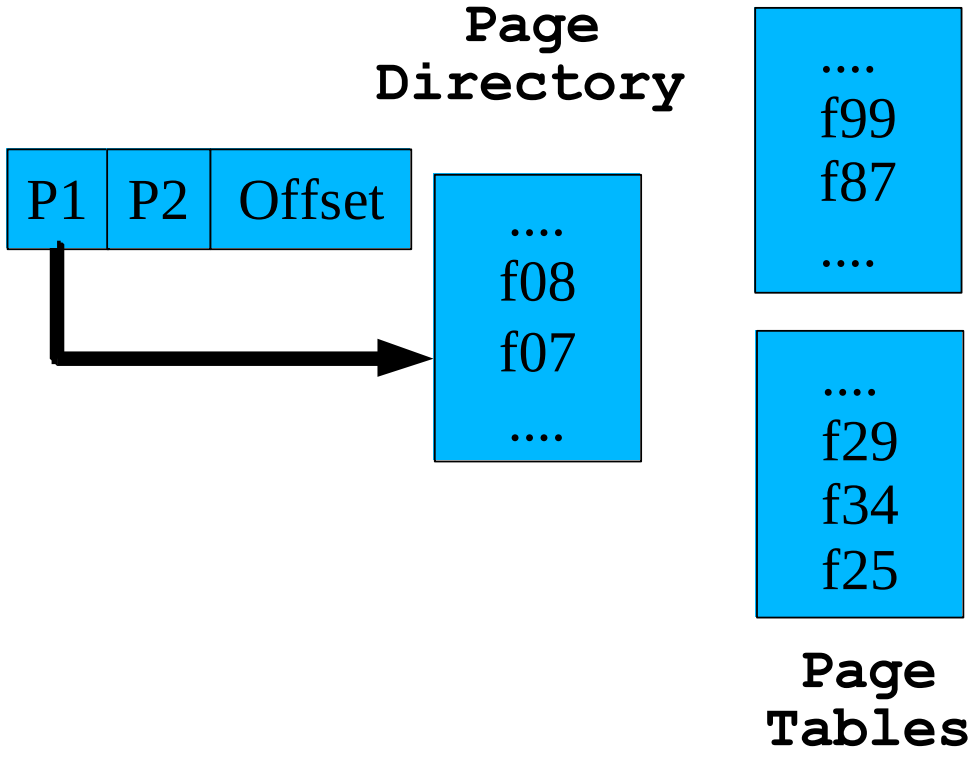
Two-Level Paging (4)

- Example 1



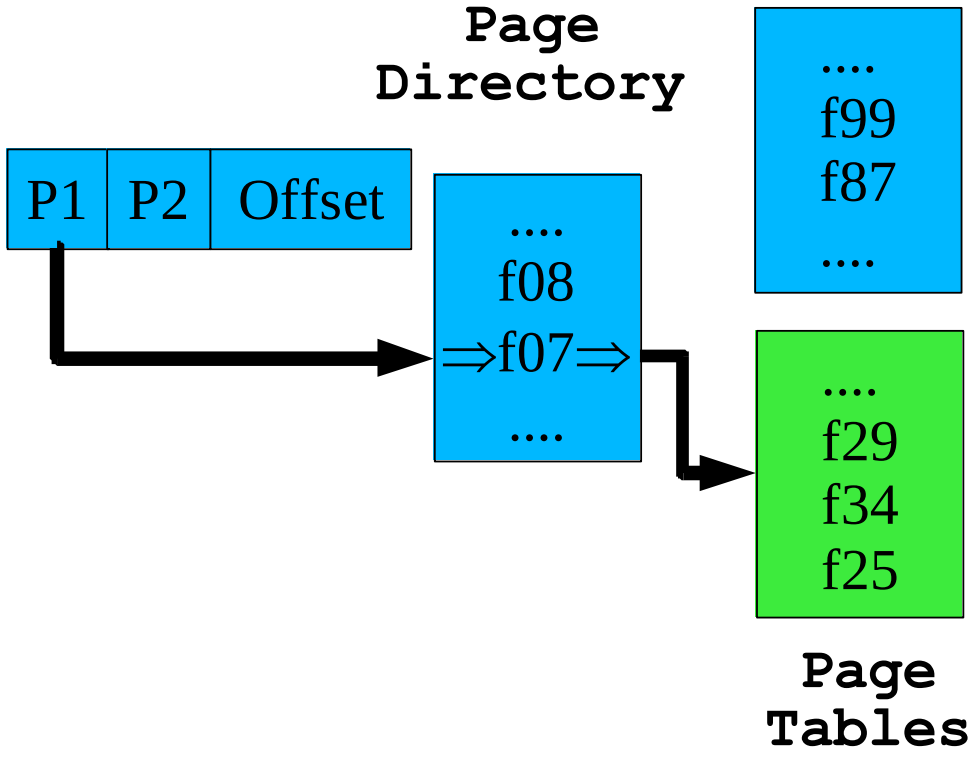
Two-Level Paging (5)

- Example 1 (ctd.)



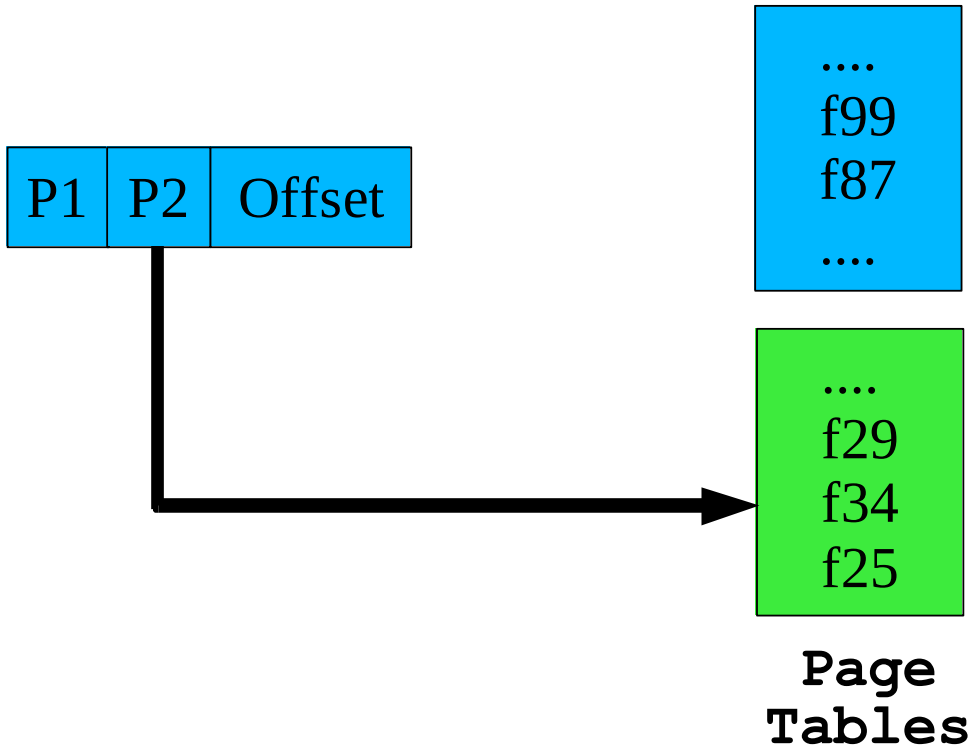
Two-Level Paging (6)

- Example 1 (ctd.)



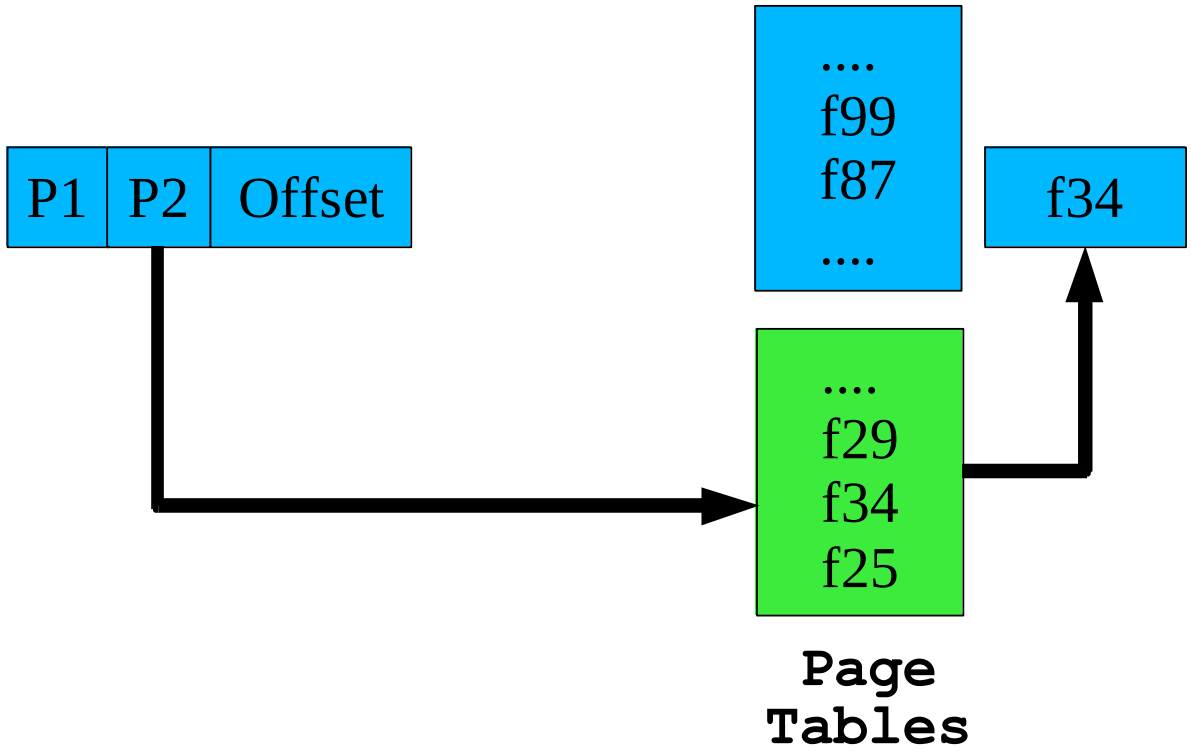
Two-Level Paging (7)

- Example 1 (ctd.)



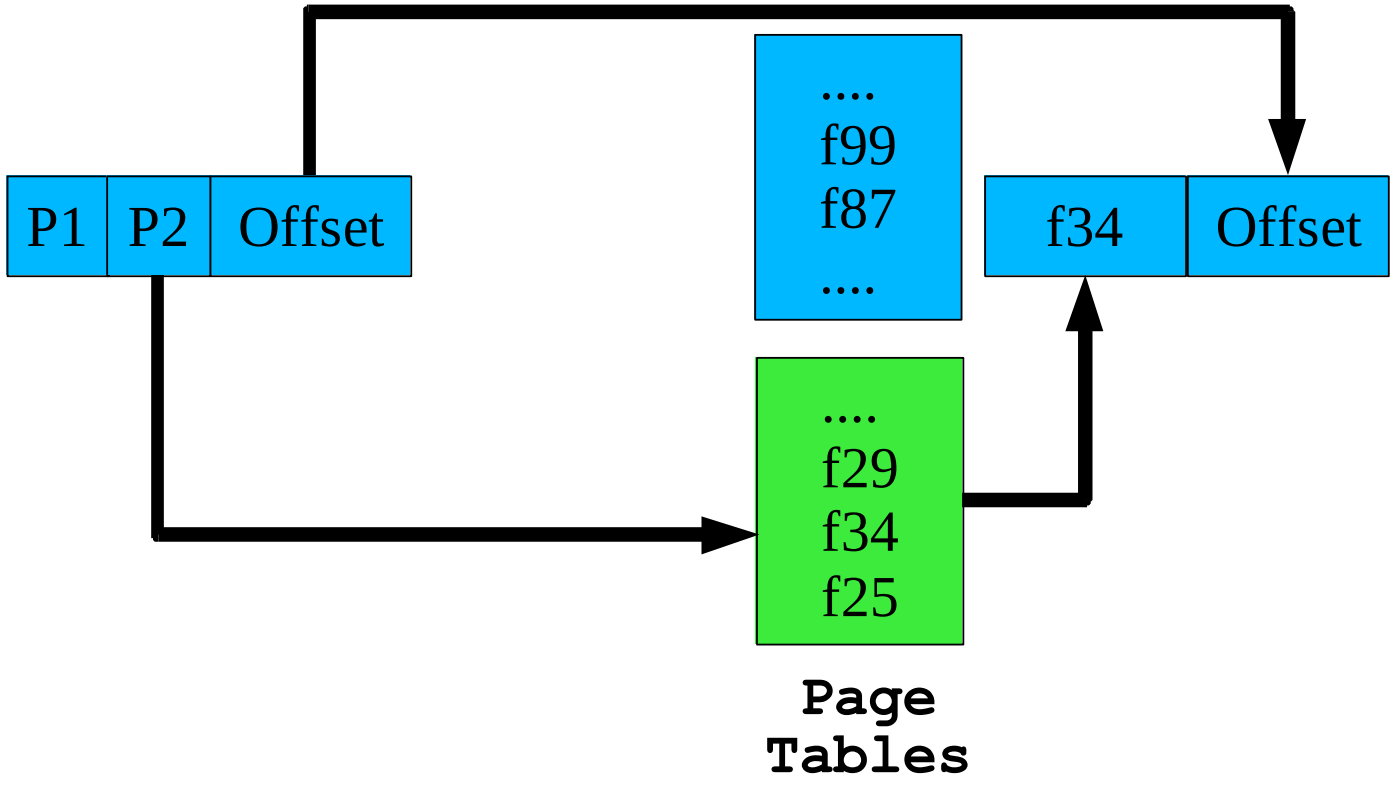
Two-Level Paging (8)

- Example 1 (ctd.)



Two-Level Paging Example (9)

- Example 1 (ctd.)



Two-Level Paging (10)

- Example 2
  - Textbook P199

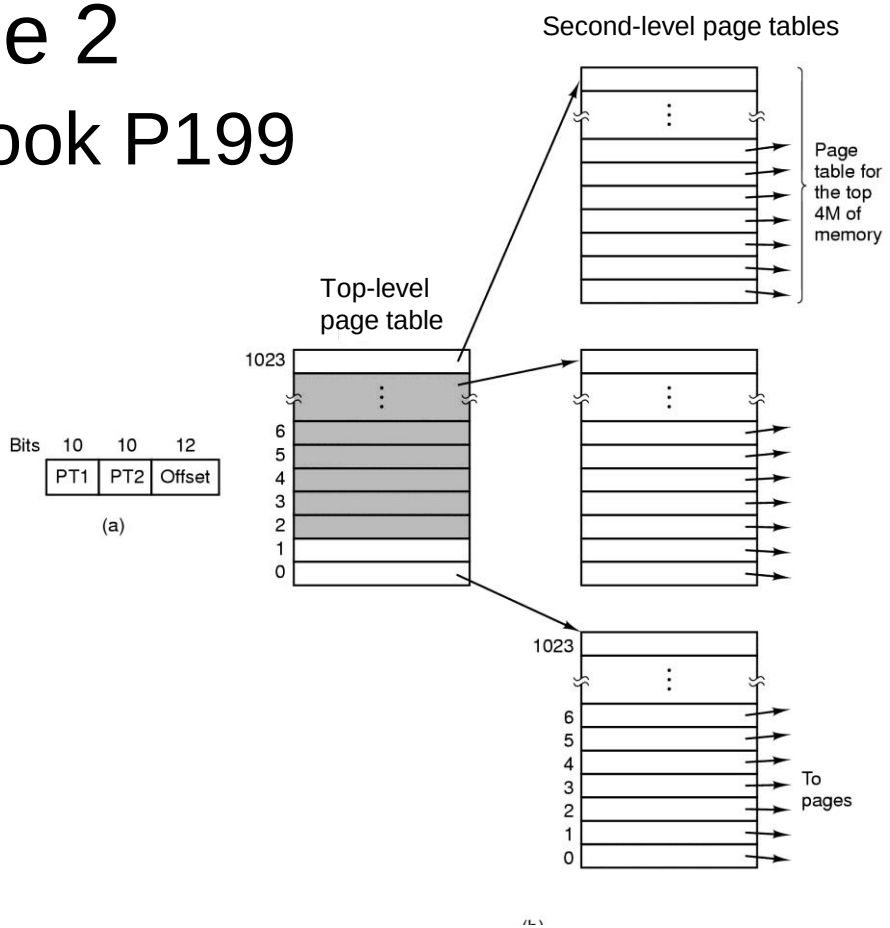


Figure 3-13 (a) 32 bit address with 2 page table fields

(b) Two-level page tables

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Generalizing

- Early architectures used 1-level page tables.
- VAX, x86 used 2-level page tables.
- SPARC uses 3-level page tables.
- Alpha 68030 uses 4-level page tables.

Why Inverted Page Table? (1)

- Question
  - Consider a system in which the virtual address space is 64 bits, the page size is 4KB, and the amount of physical memory is 512MB. Assume that each table entry would require about 4 bytes. How much space would a simple single-level page table take?

- Solution
  - Such a table contains one entry per virtual page, or $2^{(64-12)}=2^{52}$ entries. so the total page table size is $4*2^{52}=2^{54}$bytes=16petabytes. And this is for each process! (1PB=(1024^2)GB)

Why Inverted Page Table? (2)

- Discussion of Solution
  - Multi-level Page Table
    - Of course, a process is unlikely to use all 64 bits of address space, so how about using multilevel page tables?
    - How many levels would be required to ensure that each page table require only a single page?
    - Assuming an entry takes a constant 4 bytes of space, each page table can store 1024 entries, or 10 bits of address space. Thus 5 levels are required. But this results in 6 memory accesses for each address translation!

Why Inverted Page Table? (3)

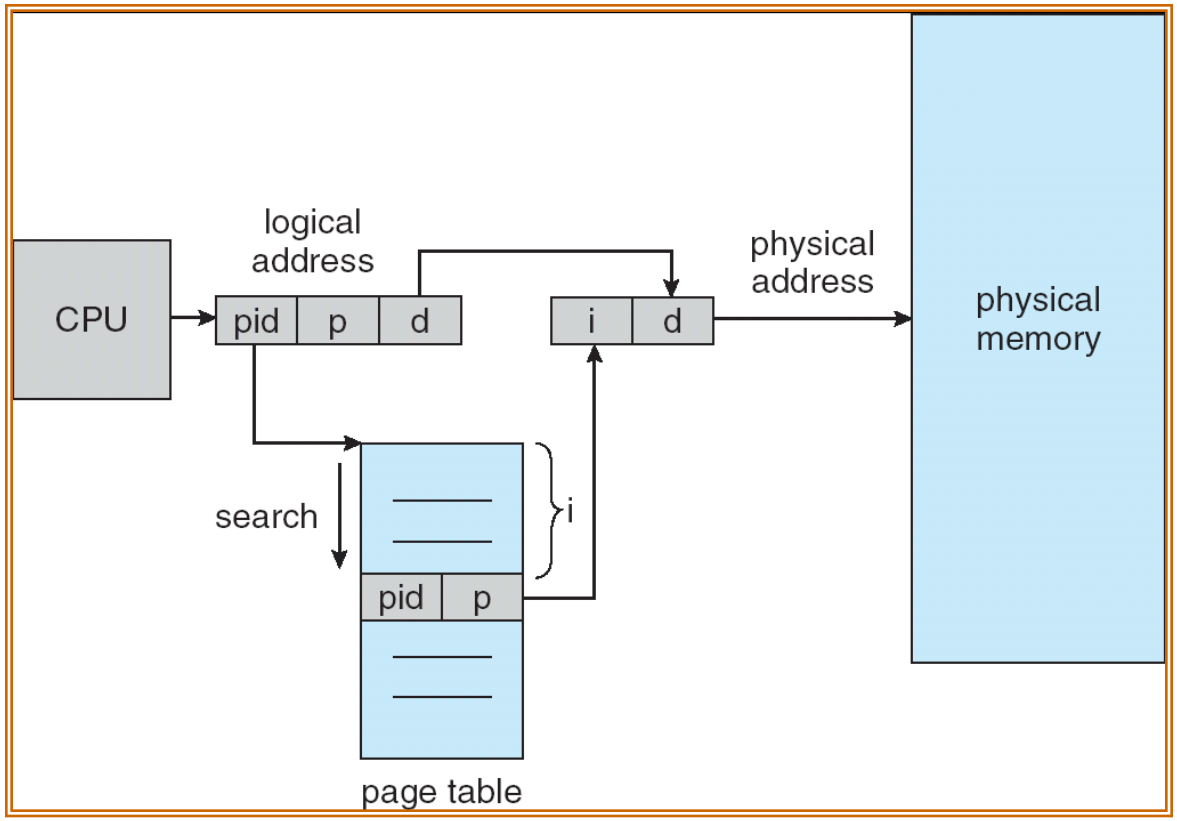
- Discussion of Solution (ctd.)
  - Inverted Page Table
    - But notice that there are only 512MB of memory in the system, or $2^{(29-12)} = 2^{17} = 128K$ physical pages.
    - If we can somehow manage to store only a single page table entry per physical page, the page table size decreases considerably, to 2MB assuming each entry takes 16 bytes.

Linear Inverted Page Table (1)

- One global page table
  - 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. (process, virtual page)
  - The physical page number is not stored, since the index in the table corresponds to it.

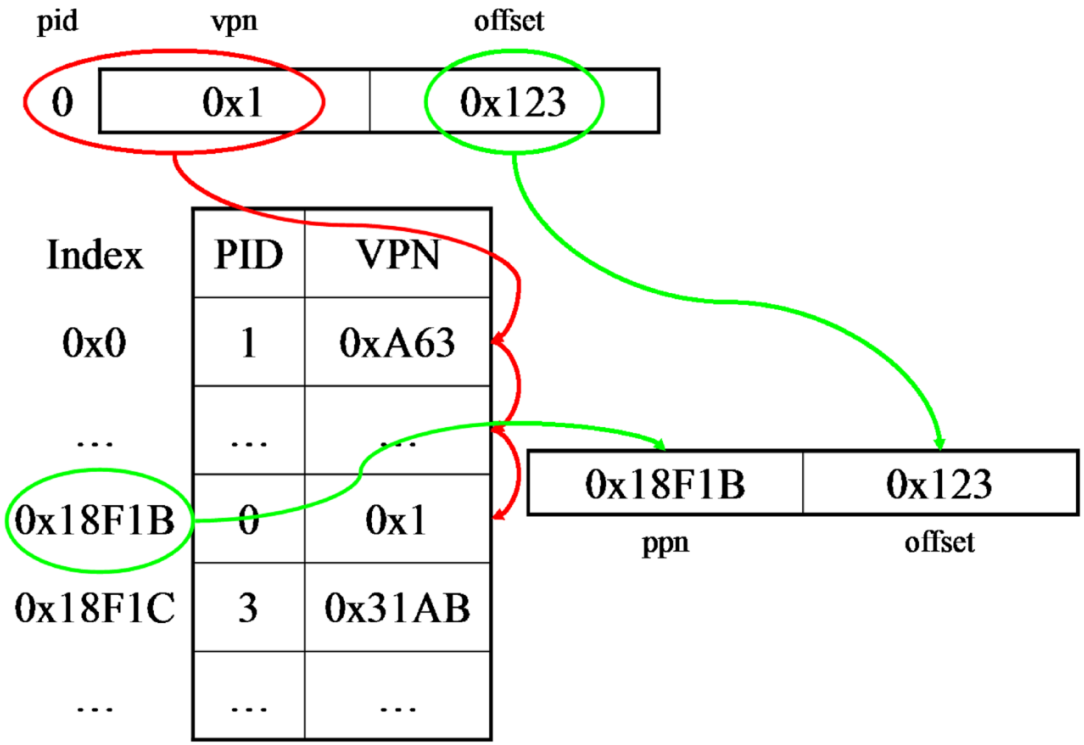
Linear Inverted Page Table (2)

- Implementation
  - TLB access eliminates search most of the time.
- Address Translation Scheme



Linear Inverted Page Table (3)

- Example
  - Translation procedure using a linear inverted page table. Info bits exist in each entry, though they are not shown.



Linear Inverted Page Tables (4)

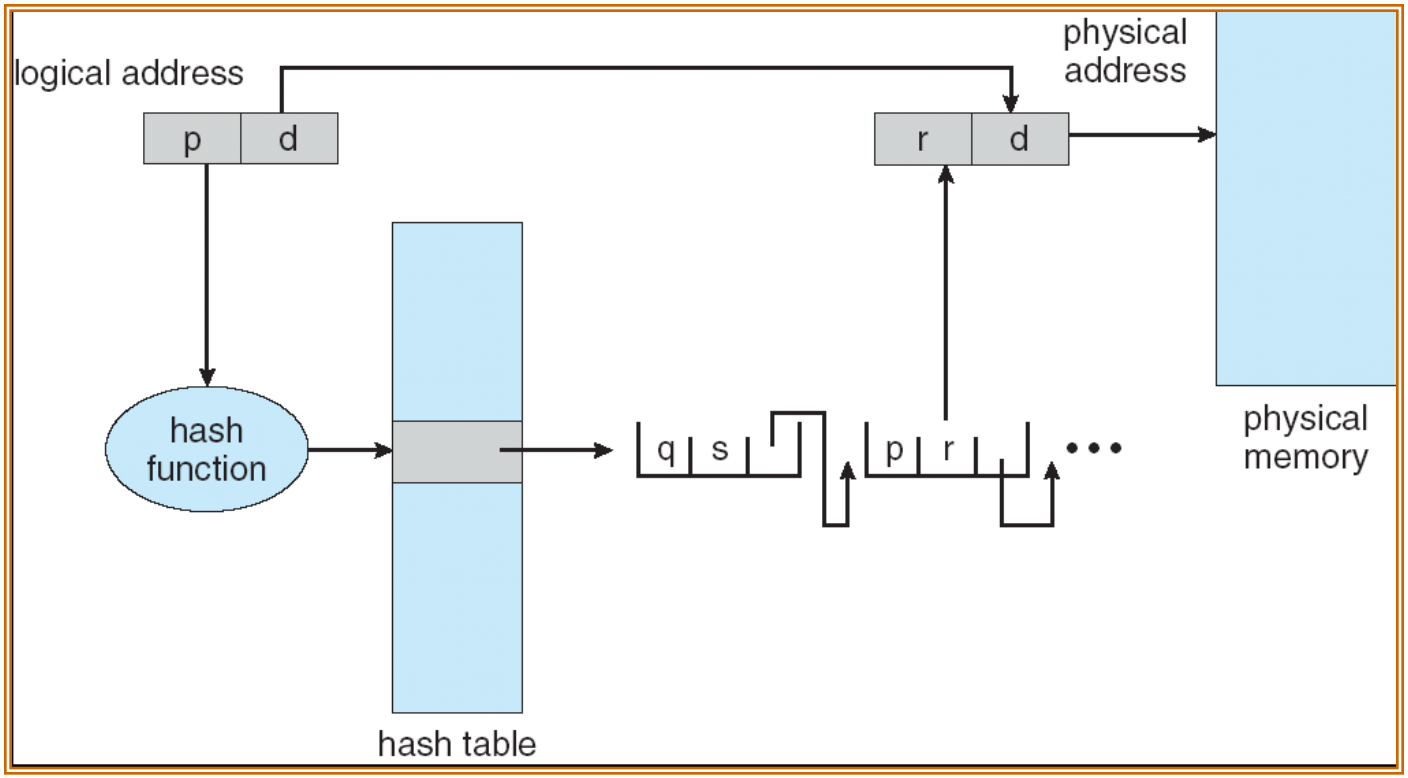
- Pros
  - Small page table for large address space.
- Cons
  - Lookup is difficult. Increases time needed to search the table when a page reference occurs. Slower memory access because of searching.
    - E.g. In the previous example, the worst scenario, will search for 2^{17} entries of the inverted page table.

Hashed Inverted Page Tables (1)

- Common in address spaces > 32 bits.
- Linear inverted page tables require too many memory accesses.
- Keep another level before actual inverted page table (hash table).
  - The input of hash table is PID and VPN, and the output is the index of inverted page table.
  - When hashing with hash table, there may be conflicts, which can be solved by using chain address method.
  - Add the next field in the inverted page table items to form a linked list (the index of the header is in the hash table)

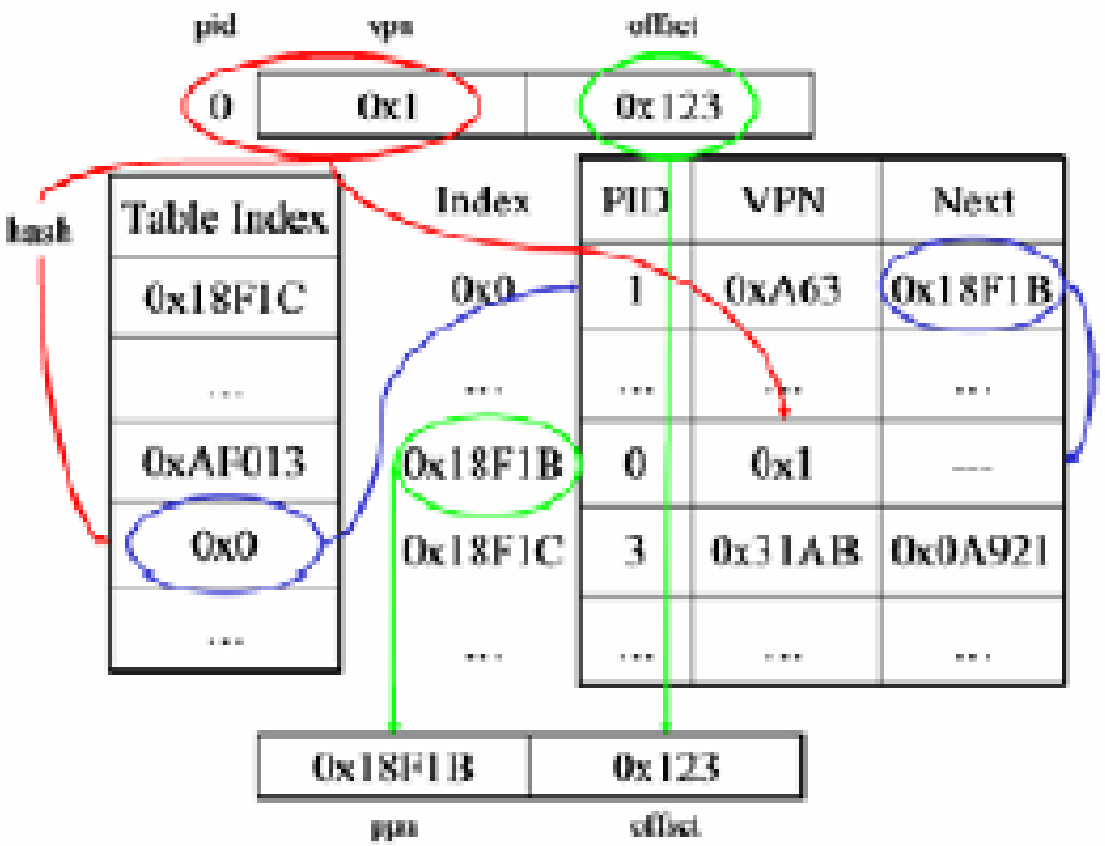
Hashed Inverted Page Tables (2)

- Lookup in hash table for page table entry.
  - Compare process ID and virtual page number
    - If match, then found.
    - If not match, check the next pointer for another page table entry and check again.

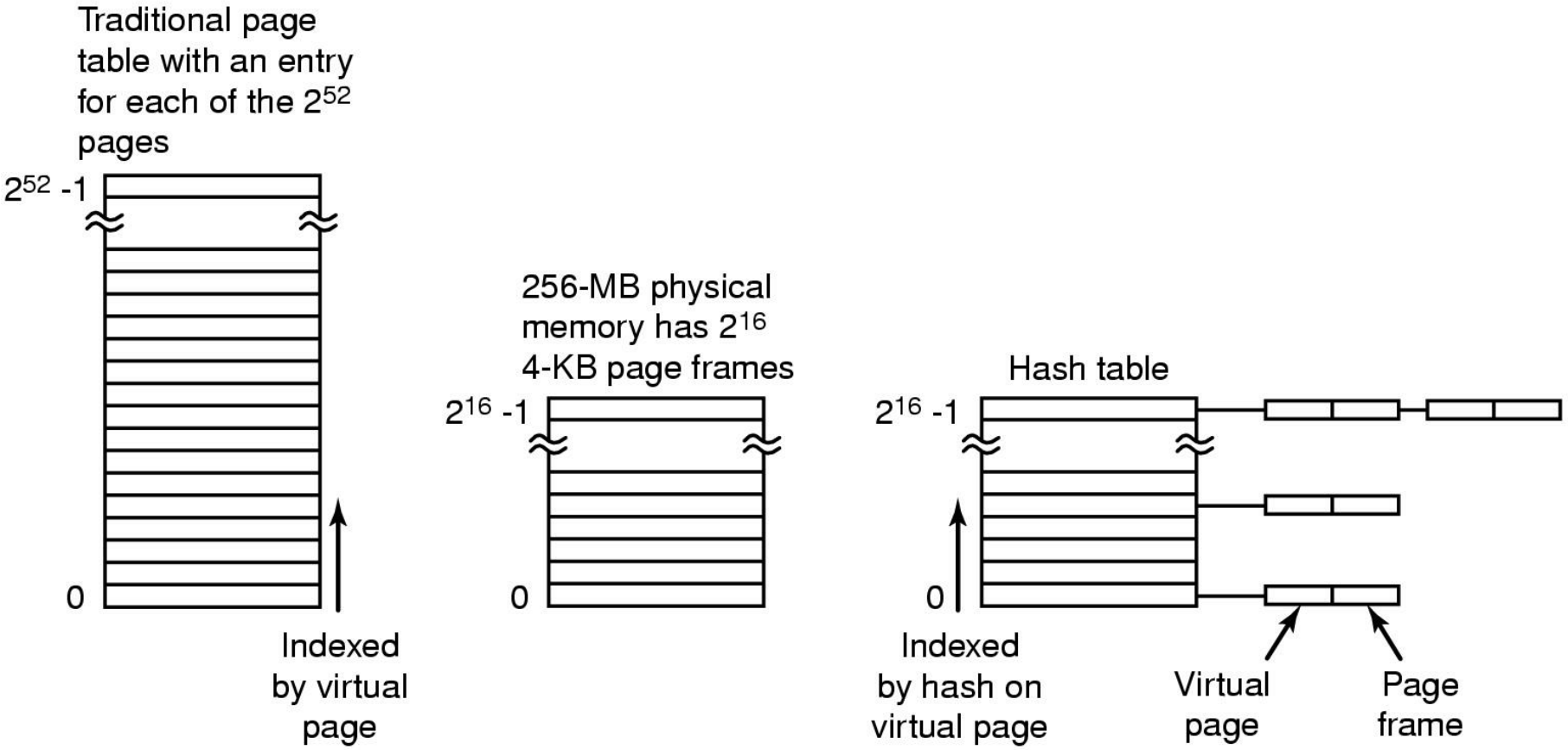


Hashed Inverted Page Tables (3)

- Example



Hashed Inverted Page Tables (4)



Comparison of a traditional page table with an inverted page table

Hashed Inverted Page Tables (5)

- Inverted page tables are currently used on some IBM and Hewlett-Packard workstations and will become more common as 64-bit machines become widespread.
- Pros
  - With a good hashing scheme and a hashmap proportional to the size of physical memory, O(1) time. Very efficient!
- Cons
  - Overhead of managing hash chains, etc

Summary

- Paging
- Address Translation Scheme
- Page Table
- TLB
- Multilevel Page Table
- Inverted Page Table

Homework

- P254: 6,7,17