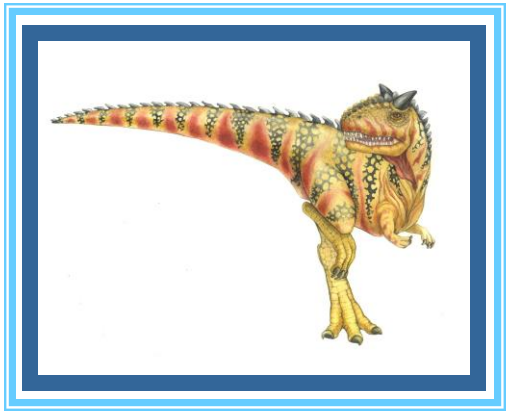
Memory Management (contd.)



- **Slides mostly borrowed from Silberschatz & Galvin**
  - With occasional modifications from us

The story so far

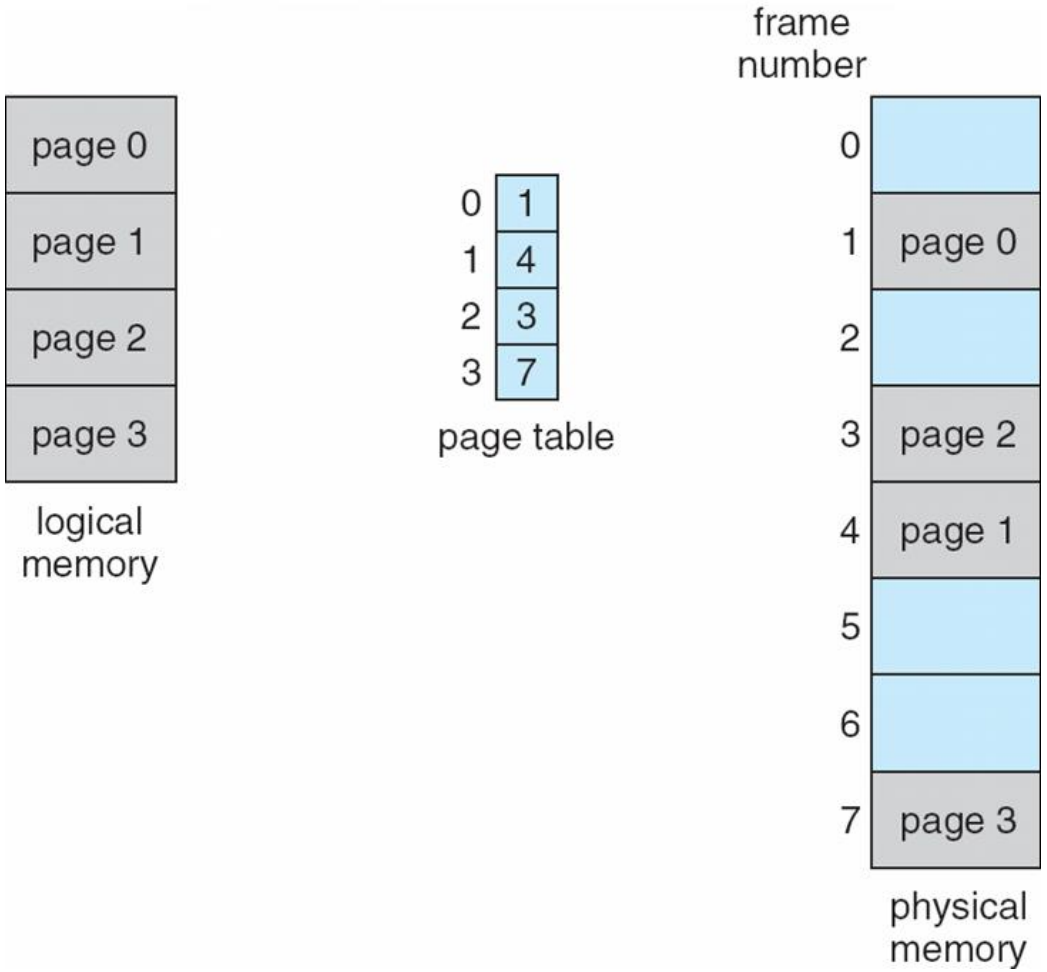
- **Basic motivation and concepts of memory management**
  - Address binding: process of mapping address space in the code to address space in main memory
  - Concept of logical/Virtual and physical address space
  - MMU: H/W that maps virtual to physical address
  - Swapping (context switch, double buffering)

The story so far

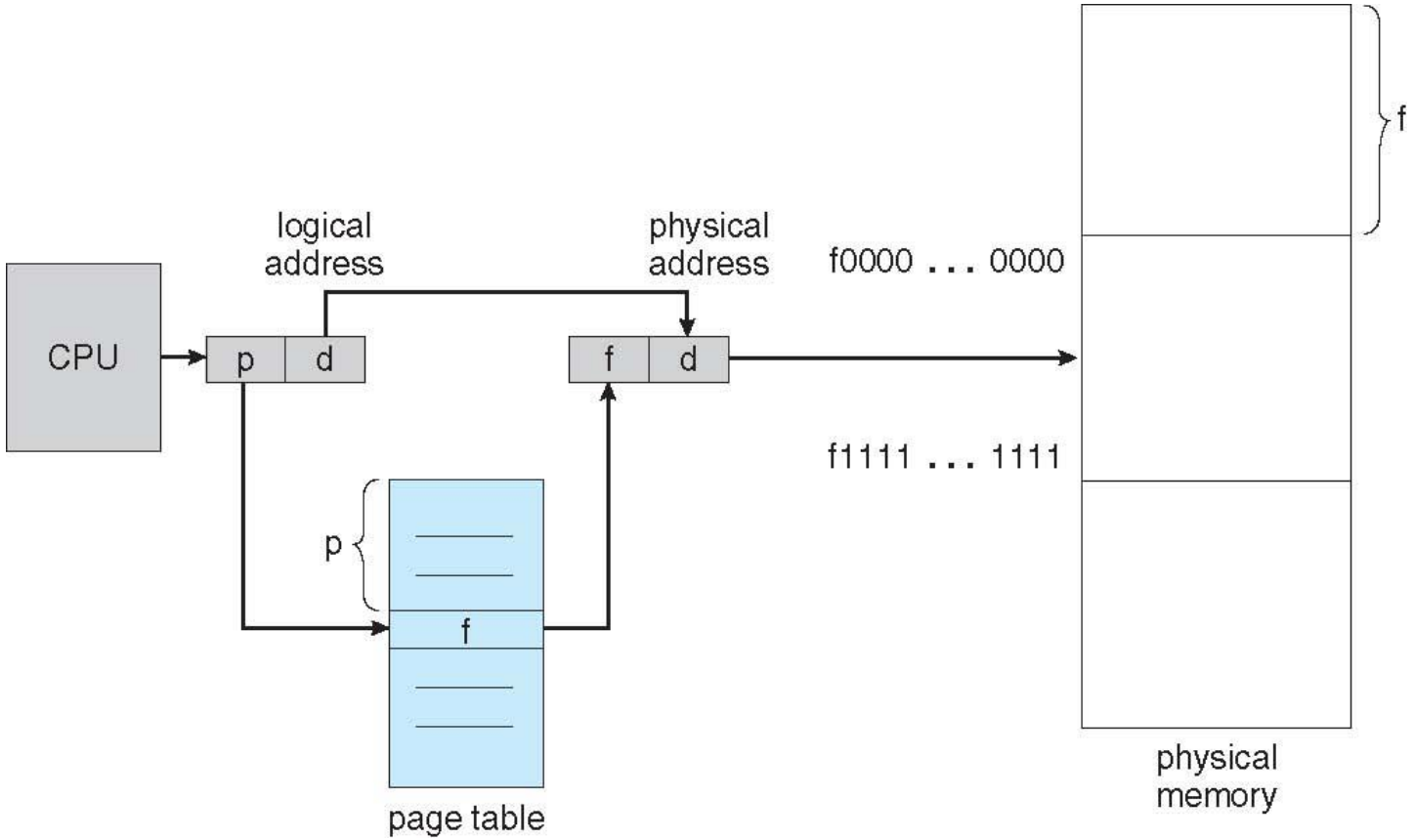
- **Basic motivation and concepts of memory management**
  - Address binding: process of mapping address space in the code to address space in main memory
  - Concept of logical/Virtual and physical address space
  - MMU: H/W that maps virtual to physical address
  - Swapping (context switch, double buffering)

- **Allocating memory to processes**
  - Contiguous, multiple-partition
  - Problems: dynamic storage allocation, internal/external fragmentation
  - Segmentation: logical parts of a program, can be distributed throughout physical memory
  - Paging: Pages, frames, page table

Paging Model - Recap



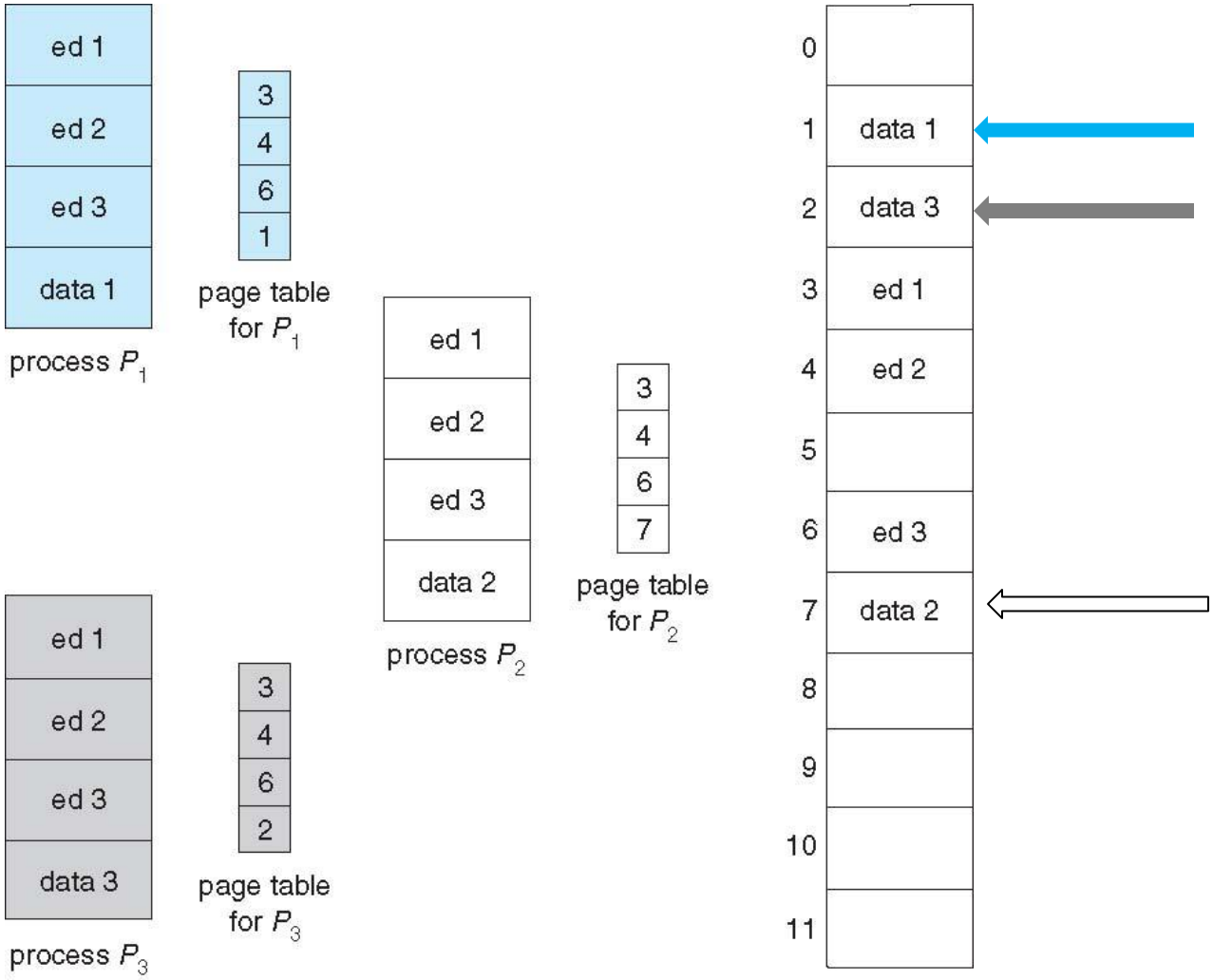
Paging Hardware - Recap



Advantage of paging: Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among multiple processes (i.e., text editors, compilers, window systems)
  - Also useful for inter-process communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process might also keep its own copy of private code / data
  - The pages for the private code / data can appear anywhere in the logical address space

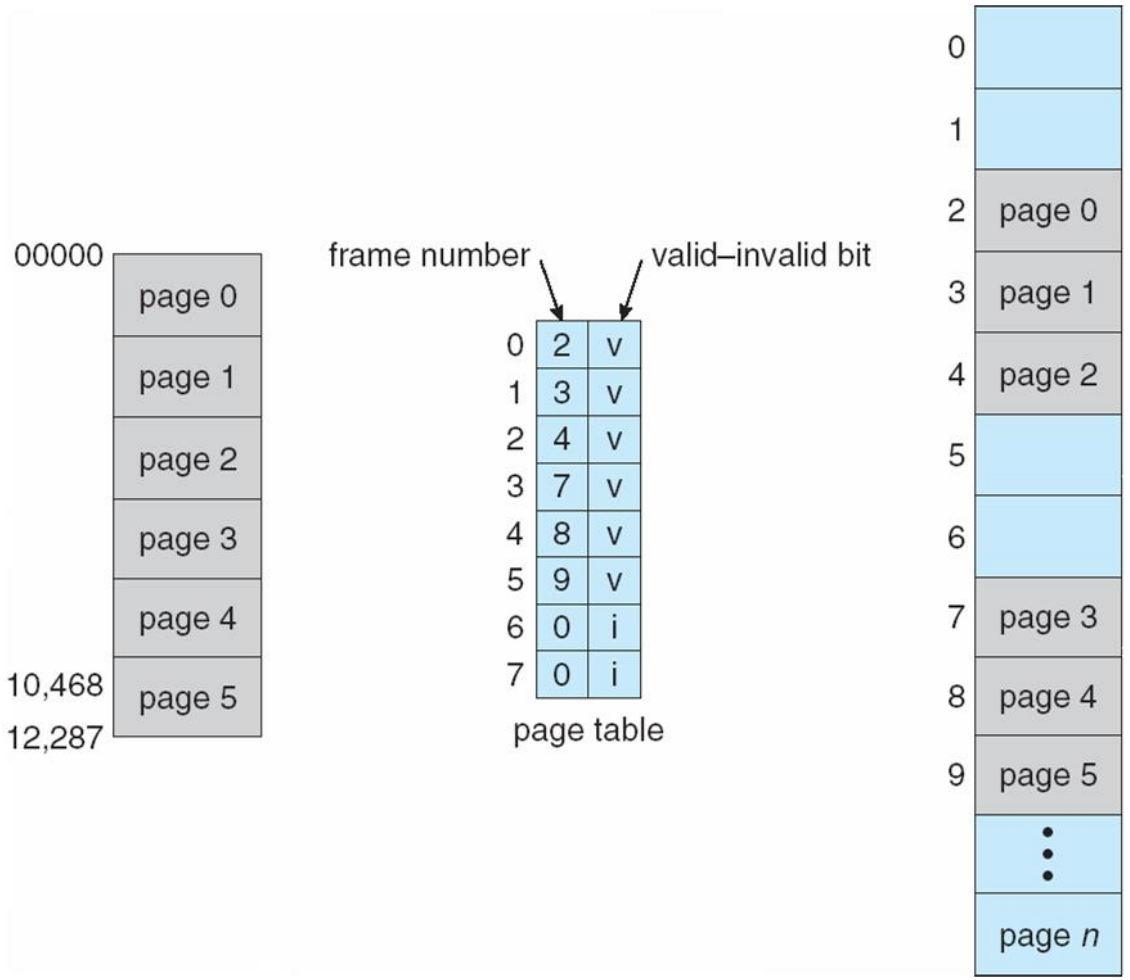
Shared Pages Example



Memory Protection in Paging

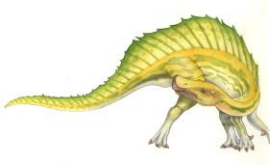
- Memory protection implemented by associating protection bits with each page (these bits usually kept in the page table)
  - Indicates if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal (valid) page
  - "invalid" indicates that the page is not in the process' logical address space
  - Can use a **page-table length register** (**PTLR**) to indicate size of the page table, i.e., which pages are valid
- Any violations result in a trap to the kernel

Valid (v) or Invalid (i) Bit In A Page Table



Memory Management: Topics

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table (and variants of the page table)



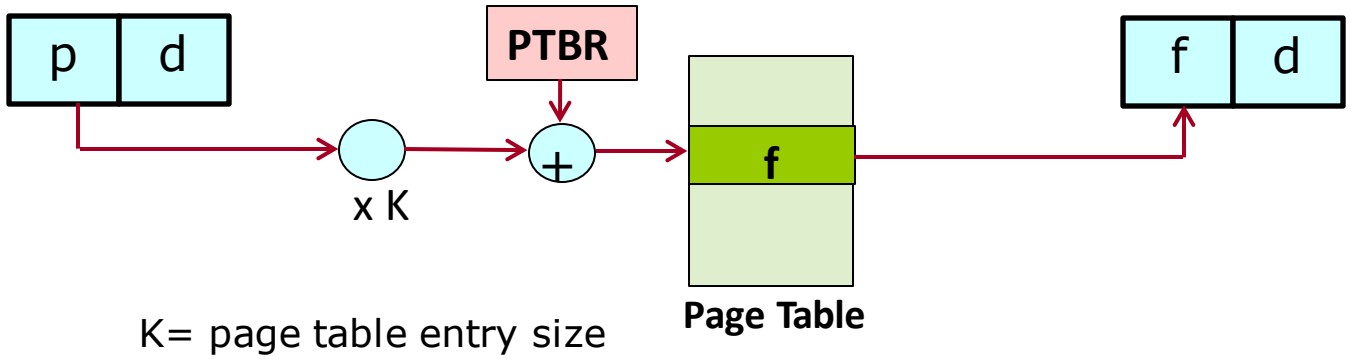
Implementation of Page Table

- Page Table has to be accessed for every memory access by a process
  - Implementation should allow fast access
- Where to store the Page Table?
- Best case – a set of dedicated registers
  - Only feasible if the page table is small
  - Not practical in most modern systems
- Practical case – Page Table stored in the memory

Implementation of Page Table

- In practice, page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table

- In this scheme every data/instruction access requires two memory accesses
  - One for the page table entry, and one for the actual data/instruction
  - Too expensive



Implementation of Page Table (Cont.)

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

- TLBs typically small (between 32 and 1,024 entries)
  - Contains a few of the (most frequently/recently accessed) page table entries
  - For a logical address generated by CPU, its page number is searched in TLB; If page number is found (TLB hit), its frame number is immediately available to access the data directly in memory

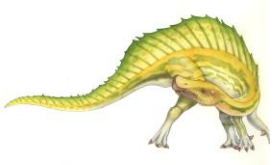
- On a TLB miss, a memory reference to the page table needed
  - Value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

Associative Memory

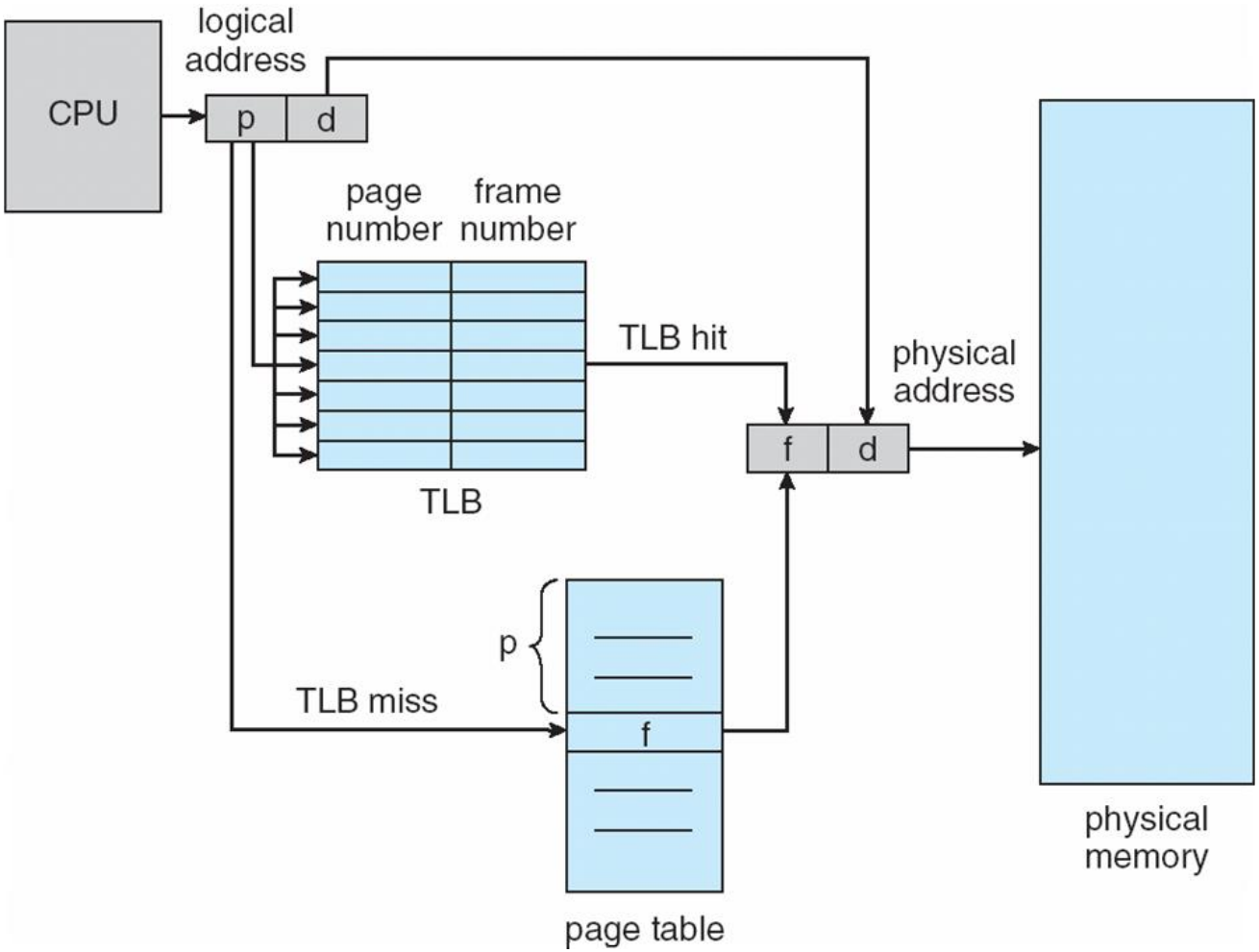
- TLB is associative memory – allows parallel search (item to be searched is compared with all keys simultaneously)

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get Frame# out
  - Otherwise get Frame# from page table in memory



Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α (% of times that a page number is found in TLB)

- **Effective Access Time** (**EAT**)

$$EAT = (1 + \varepsilon)\,\alpha + (\mathbf{2 + \varepsilon})(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$

Even larger for multi-level page tables

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers: 2^{32} byte memory locations
  - Page size of 4 KB (4x 2^{10} byte = 2^{12} byte)
  - Page table would have ~1 million entries (2^{32} / 2^{12})
  - If each page table entry is 4 bytes
    - 4 MB (2^{20}x4 byte) of physical address space / memory for page table alone – equivalent to 1024 pages of 4KB each
    - That amount of memory used to cost a lot
    - **Don't want to allocate that contiguously in main memory**

Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers: 2^{32} byte memory locations
  - Page size of 4 KB (4x 2^{10} byte = 2^{12} byte)
  - Page table would have ~1 million entries (2^{32} / 2^{12})
  - If each page table entry is 4 bytes
    - 4 MB (2^{20}x4 byte) of physical address space / memory for page table alone – equivalent to 1024 pages of 4KB each
    - That amount of memory used to cost a lot
    - **Don't want to allocate that contiguously in main memory**

We moved to paging because we wanted to avoid allocating large blocks of contiguous memory

But we are back to the same problem – as logical address space increases, page tables need large memory blocks

Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers: 2^{32} byte memory locations
  - Page size of 4 KB (4x 2^{10} byte = 2^{12} byte)
  - Page table would have ~1 million entries (2^{32} / 2^{12})
  - If each page table entry is 4 bytes
    - 4 MB (2^{20}x4 byte) of physical address space / memory for page table alone – equivalent to 1024 pages of 4KB each
    - That amount of memory used to cost a lot
    - **Don't want to allocate that contiguously in main memory**
- **Solutions**
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

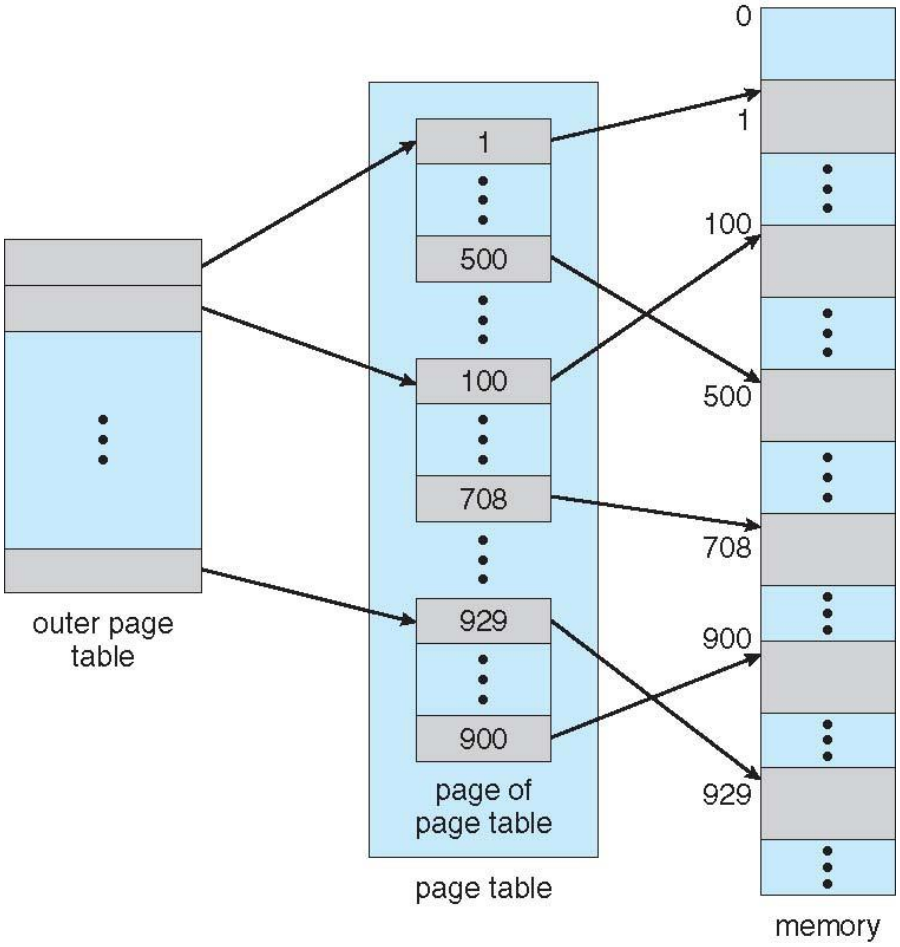
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then **page the page table**

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then **page the page table (POPT: pages of page table)**

Two-Level Page-Table Scheme



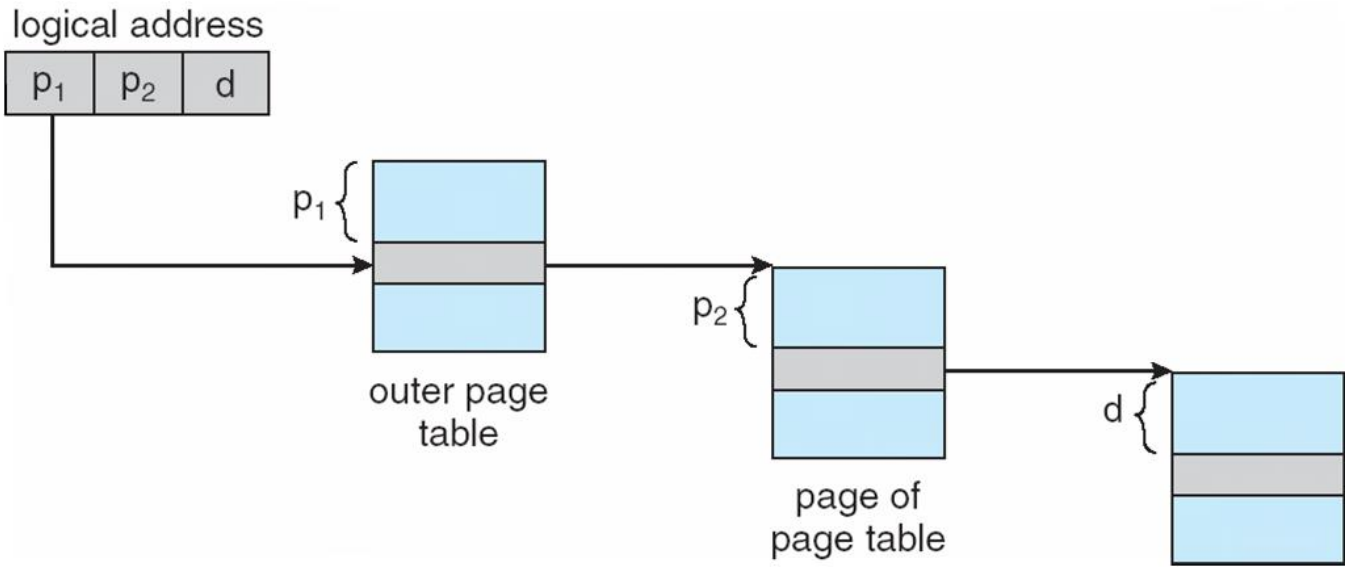
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

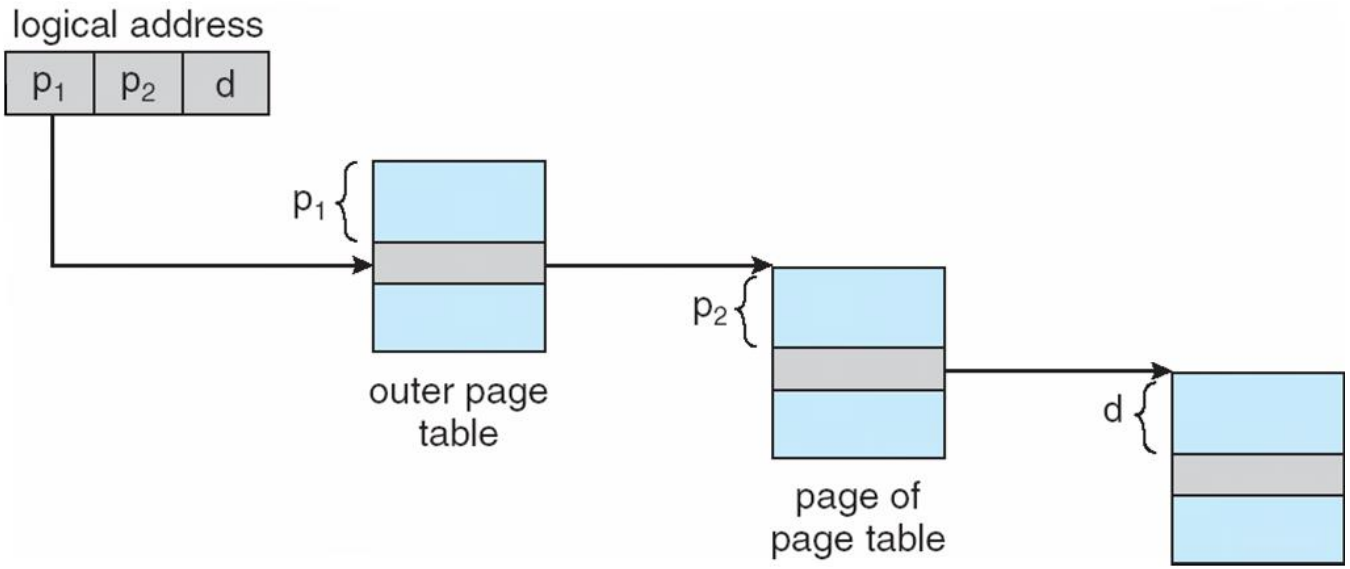
page number		page offset
p_1	p_2	d
12	10	10

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



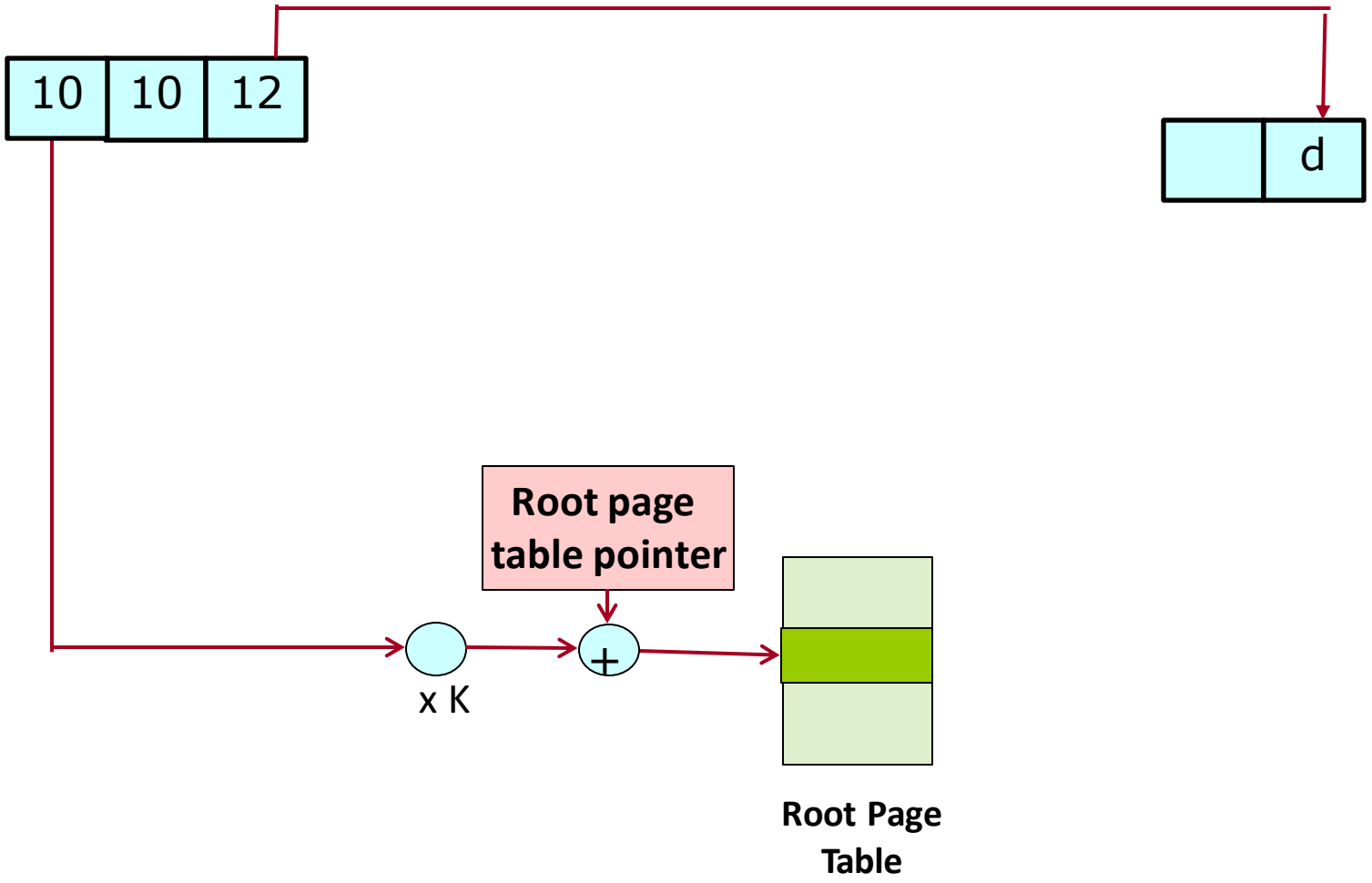
Address-Translation Scheme



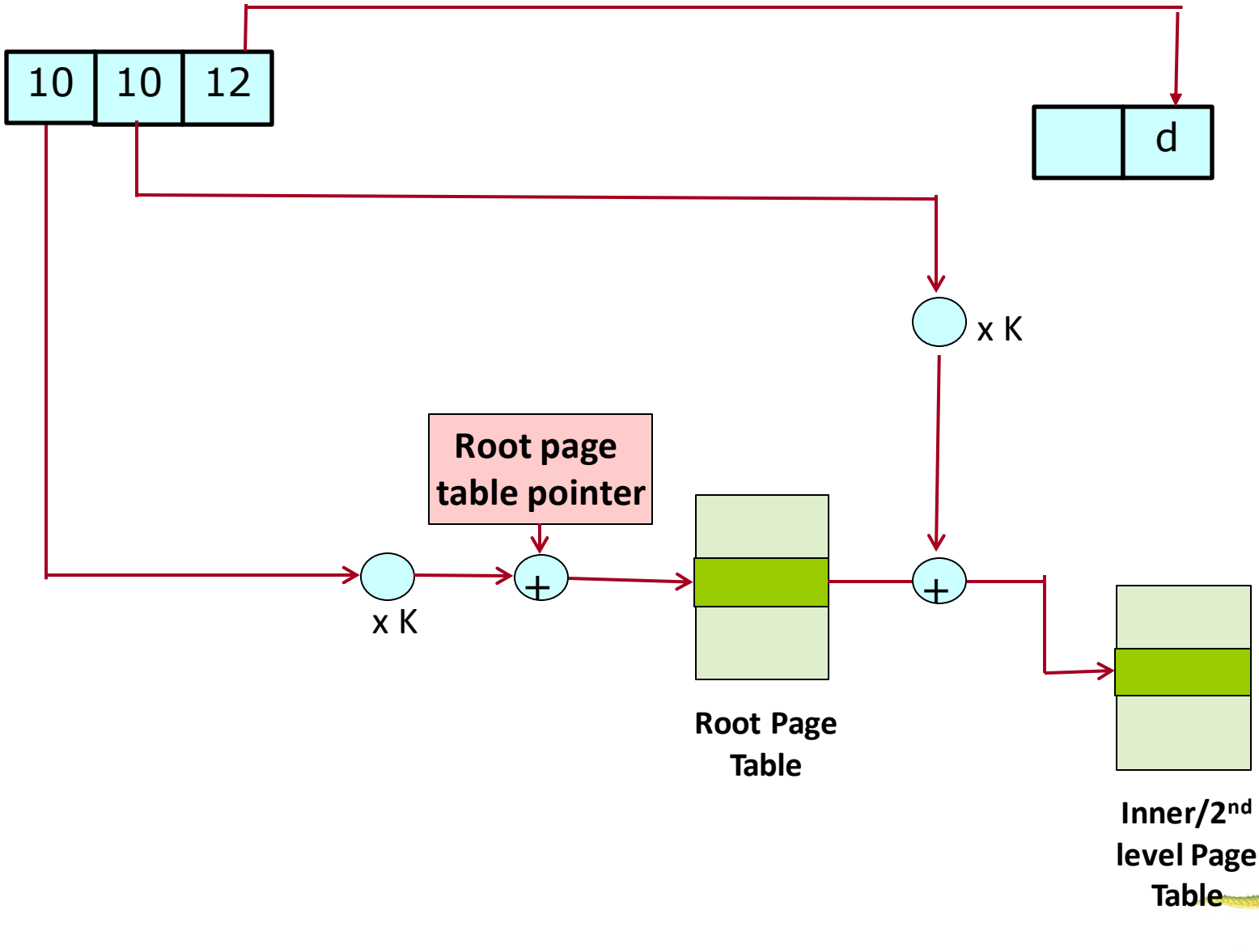
Cost: Three memory Accesses (previously two)

Implementation of 2-level paging

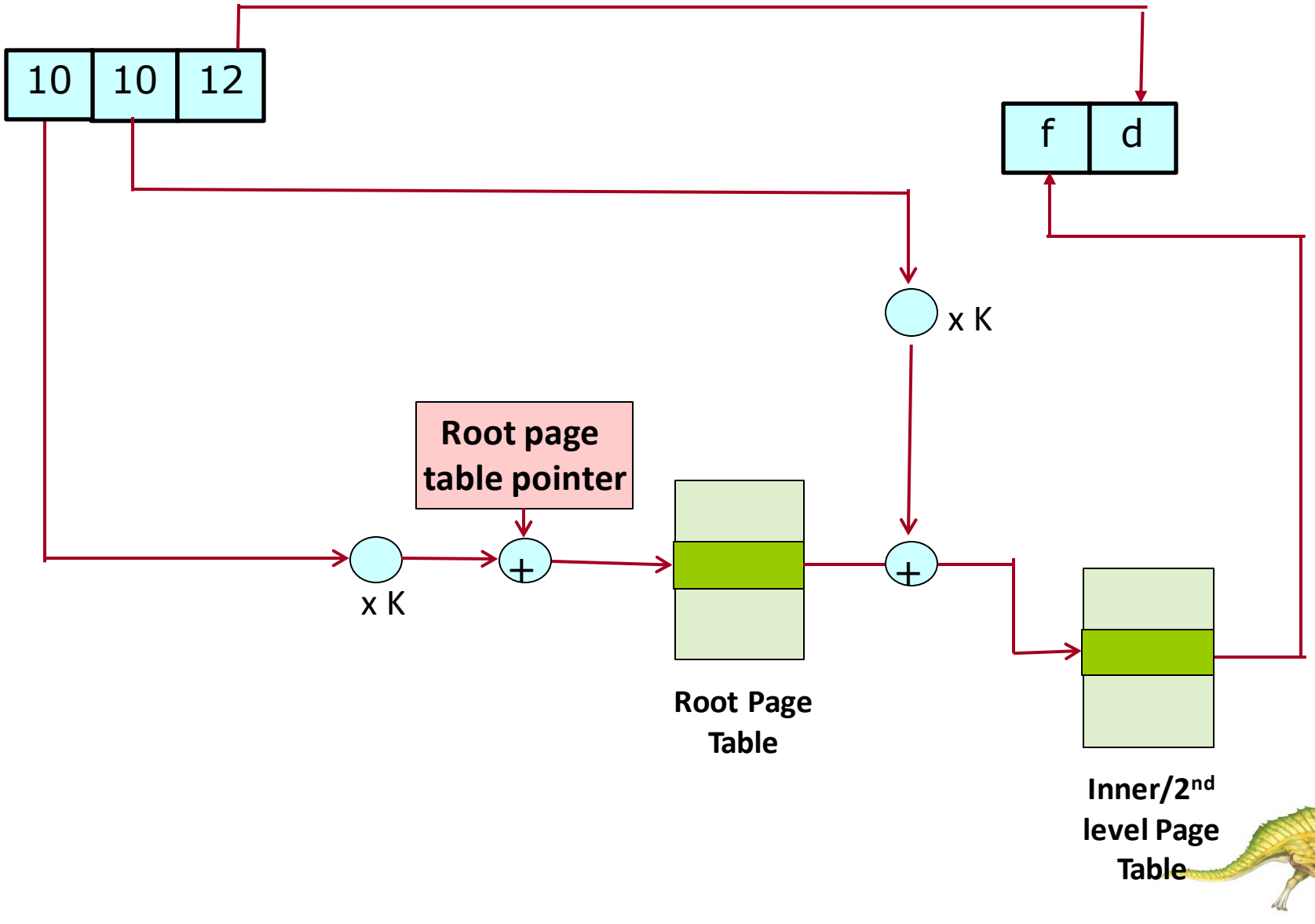
Implementation of 2-level paging



Implementation of 2-level paging

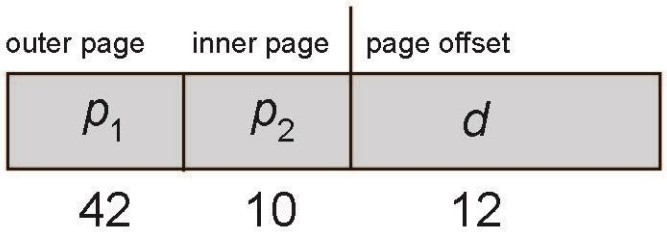


Implementation of 2-level paging



64-bit Logical Address Space

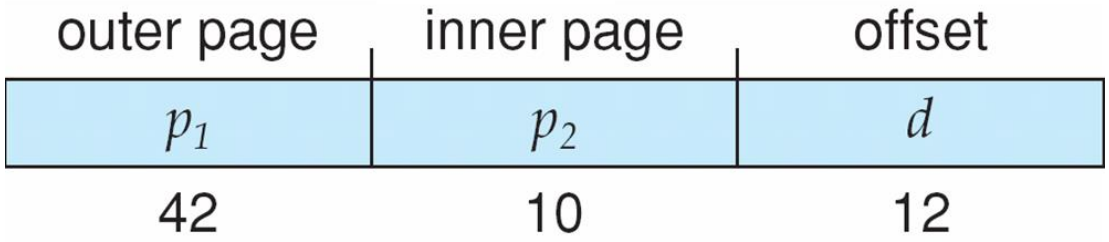
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If 2-level scheme, inner/2^{nd} level page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes = **2^{14} GB**

Won't work!

Three-level Paging Scheme



- One solution is to add a 2nd outer page table

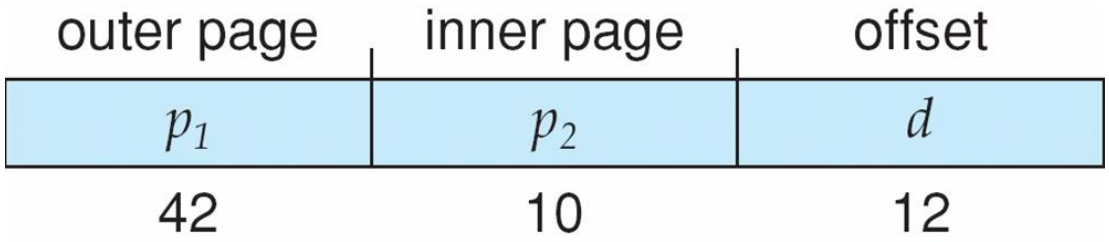
Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

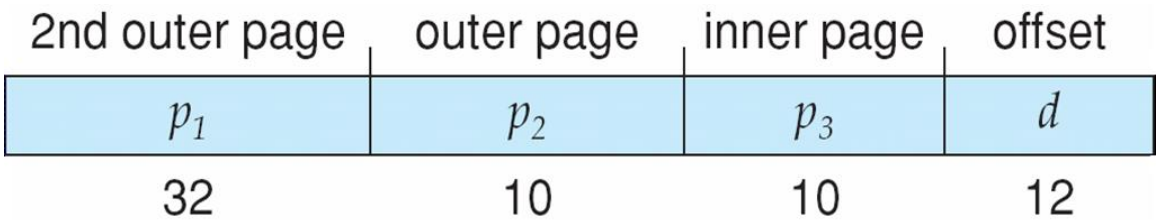
- One solution is to add a 2nd outer page table

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

Three-level Paging Scheme

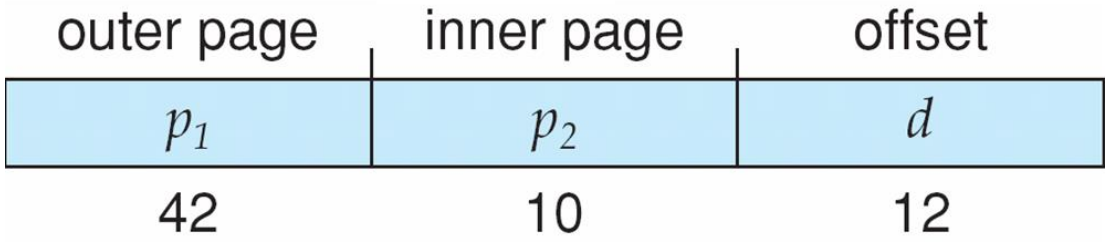


- One solution is to add a 2nd outer page table

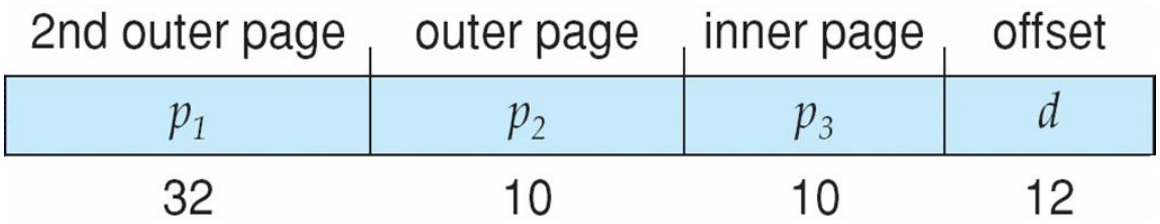


- But in this example the 2nd outer page table is still 2^{34} bytes in size
  - Even 16 GB is too huge to store contiguously
  - Also 4 memory access to get to one physical memory location

Three-level Paging Scheme



- One solution is to add a 2nd outer page table



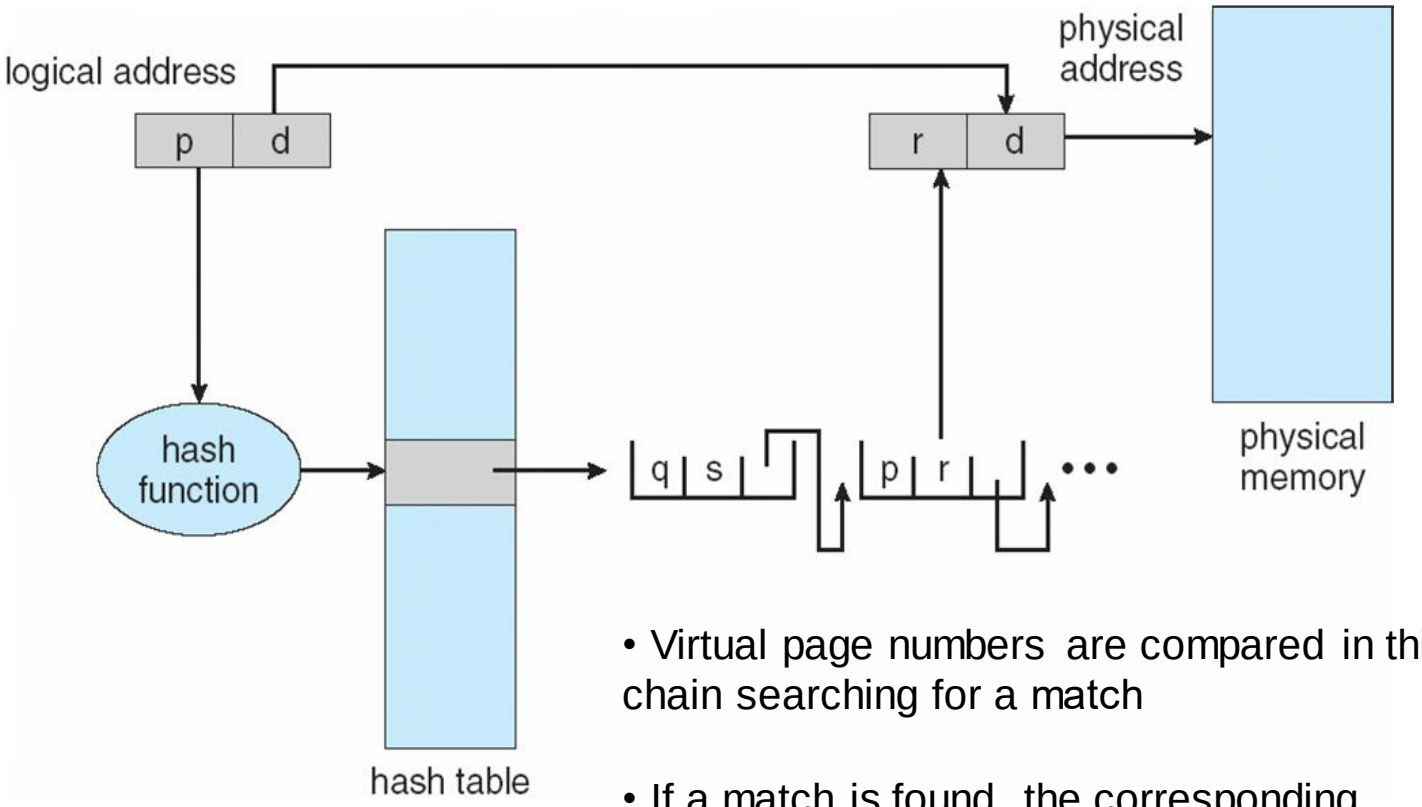
- But in this example the 2nd outer page table is still 2^{34} bytes in size
  - Even 16 GB is too huge to store contiguously
  - Also 4 memory access to get to one physical memory location

Multi-level paging schemes do not scale well
Need alternate mechanisms

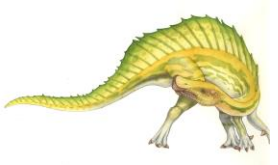
Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains
  - (1) the virtual page number
  - (2) the value of the mapped physical page frame
  - (3) a pointer to the next element

Hashed Page Table

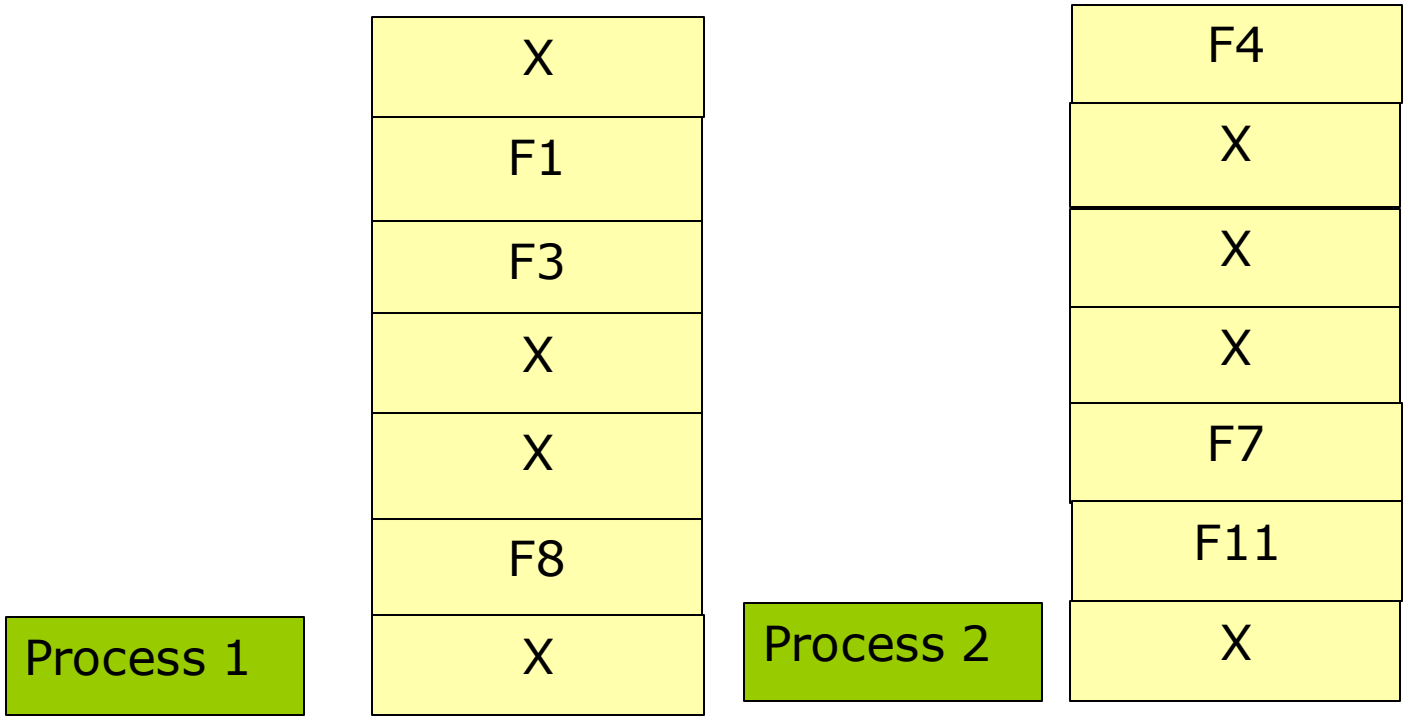


- Virtual page numbers are compared in this chain searching for a match
- If a match is found, the corresponding physical frame is extracted



Inverted Page Table: Background

- Till now what we have discussed:
  - Each process has a page table that keeps track of all logical pages of this process

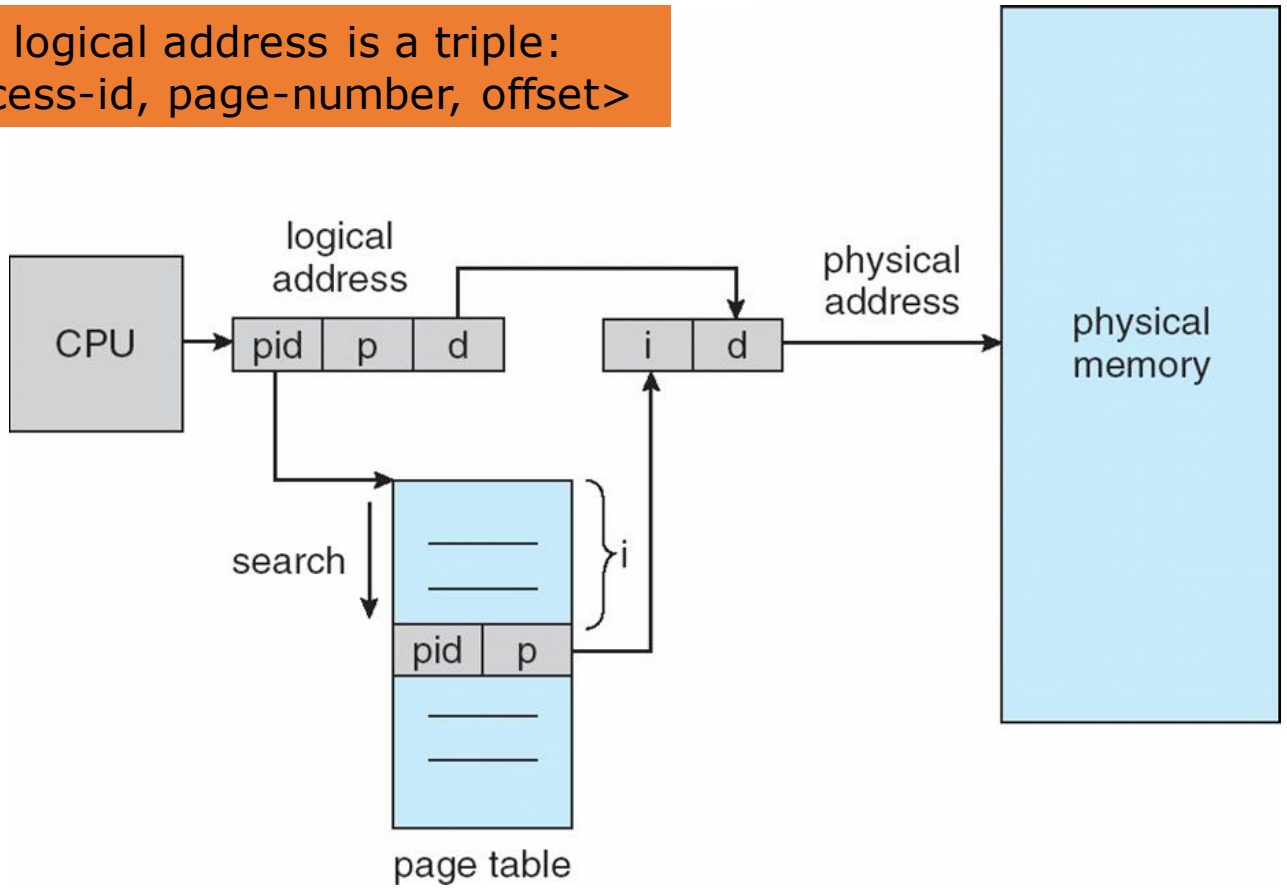


Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages (frames)
- Inverted page table: One entry for each real page (frame) of memory
  - Entry = the virtual address of the page stored in that real memory location + information of the process that owns the page

Inverted Page Table Architecture

A logical address is a triple:
<process-id, page-number, offset>



Inverted Page Table

- Decreases memory needed to store each page table, but increases time needed to search the table for a page reference
  - Searching uses the CPU-generated virtual address, but the inverted page table is sorted by the physical address
- Can make searching physical address relatively fast:
  - Use hash table (as described earlier)
  - TLB can accelerate access further (even before consulting hash table)

Summary

- Advantages of Paging – shared memory, avoid external fragmentation
- Requirements for shared paging – valid / invalid bit

- Where to store the Page Table? In the memory
  - Cost – multiple memory accesses for each data access
  - Optimization for faster access – use TLB (associative cache)

- Structure of the Page Table in memory
  - Page Table can be too large for contiguous memory allocation
  - Hierarchical multi-level Page Table
    - Store pages of page table (POPT)
    - Does not scale well as logical address space becomes larger
  - Hashed Page Table
  - Inverted Page Table