CISC 7310X
C06b Main Memory: Paging

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

- These slides are a revision of the slides provided by the authors of the textbook

Outline

- Paging
- Structure of the Page Table

- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

Paging

- A memory allocation scheme where physical address space of a process can be noncontiguous
  - Process is allocated physical memory whenever there is available physical memory
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks

Frames and Pages

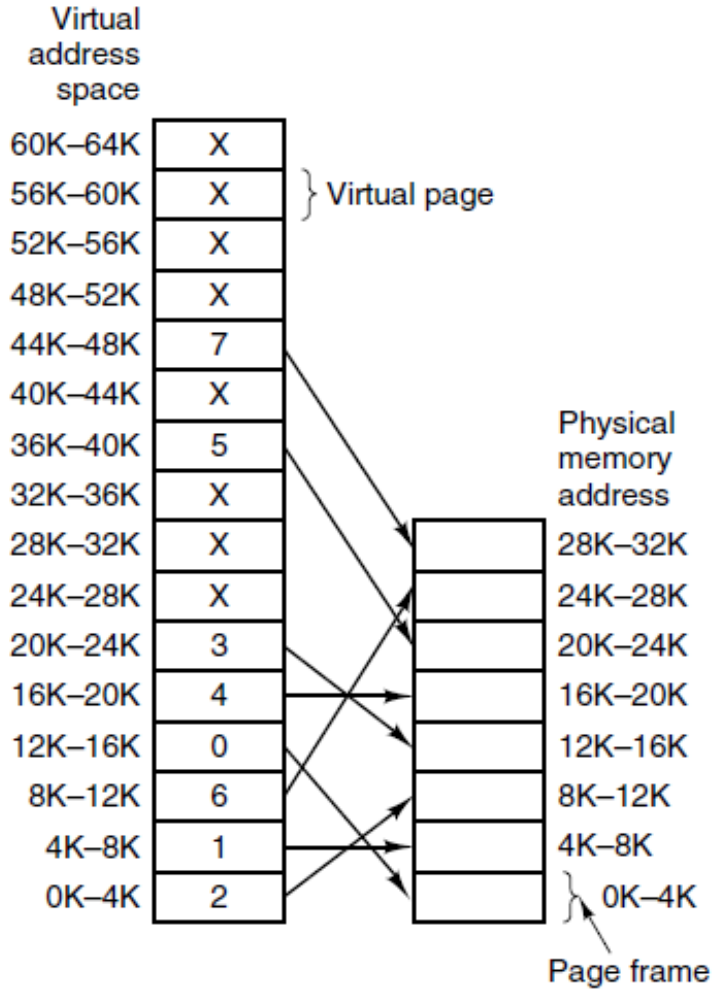
- Divide *physical memory* into fixed-sized blocks called *frames*
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide *logical memory* into blocks of same size called *pages*
- Backing store (where binary executable is store) likewise split into pages

Paging: Basic Scheme

- OS keeps track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
  - Map N pages to N frames
- Set up a page table to translate logical to physical addresses
- Still have Internal fragmentation (some memory may be unused in a frame)

Example: Basic Paging Scheme

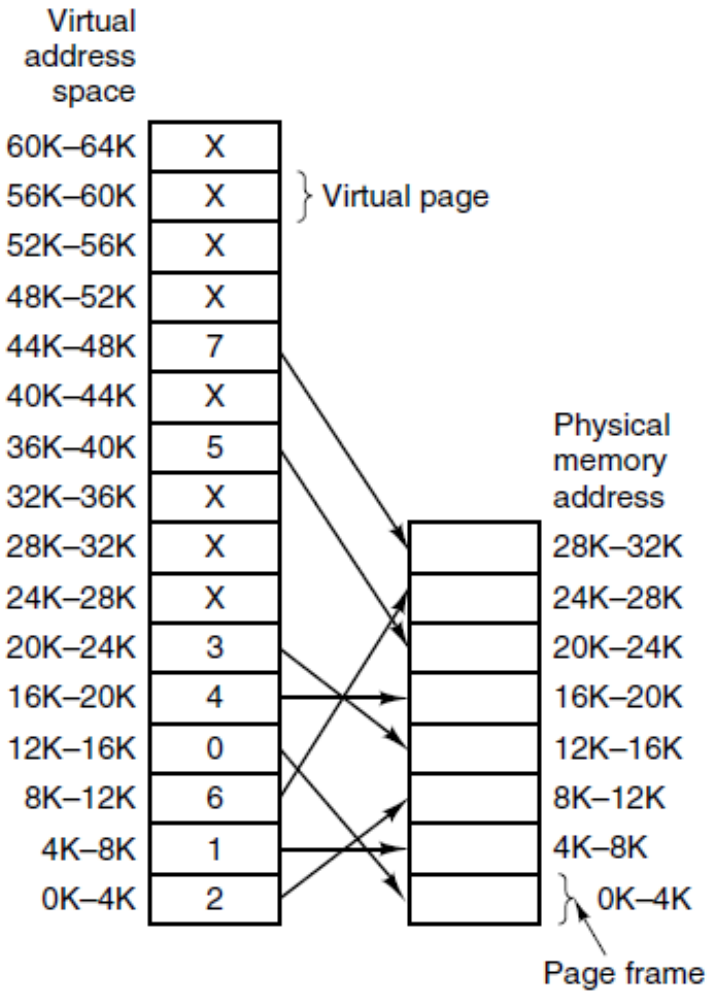
- Virtual/logical address (process)
  - 16-bit address
  - Address space
    - 0 ~ (2^{16} – 1 = 64K - 1)
    - Divided into pages, each 4KB
- 32 KB physical memory
  - Page frames: pages in the physical memory
- 64 KB virtual space: 16 x 4 = 64, so 16 virtual pages
- 32 KB physical memory: 8 x 4 = 32, so 8 page frames
- Transfer between memory and disk is always in whole pages



- [Figure 3-9 in Tanenbaum & Bos, 2014]

Example: Memory Address

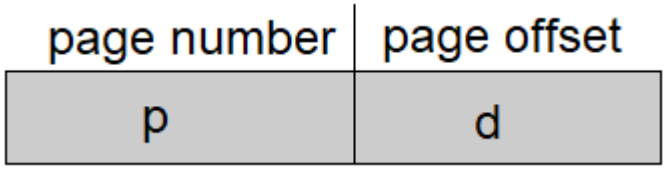
- MMU maintains a map per process
  - Page size: 4K
- What if
  - MOV REG, (8203)
- 8203 is a virtual address, passed to MMU (8K = 8192)
  - determines that 8203 is in page 2 in virtual address space
  - determines that the page is mapped to page frame 6 in physical memory
  - Maps the virtual address to physical address
    - 8203 / 4K = 2 (table lookup → 6)
    - 8203 % 4K + 6 * 4K = 24587



- [Figure 3-9 in Tanenbaum & Bos, 2014]

Address Translation Scheme

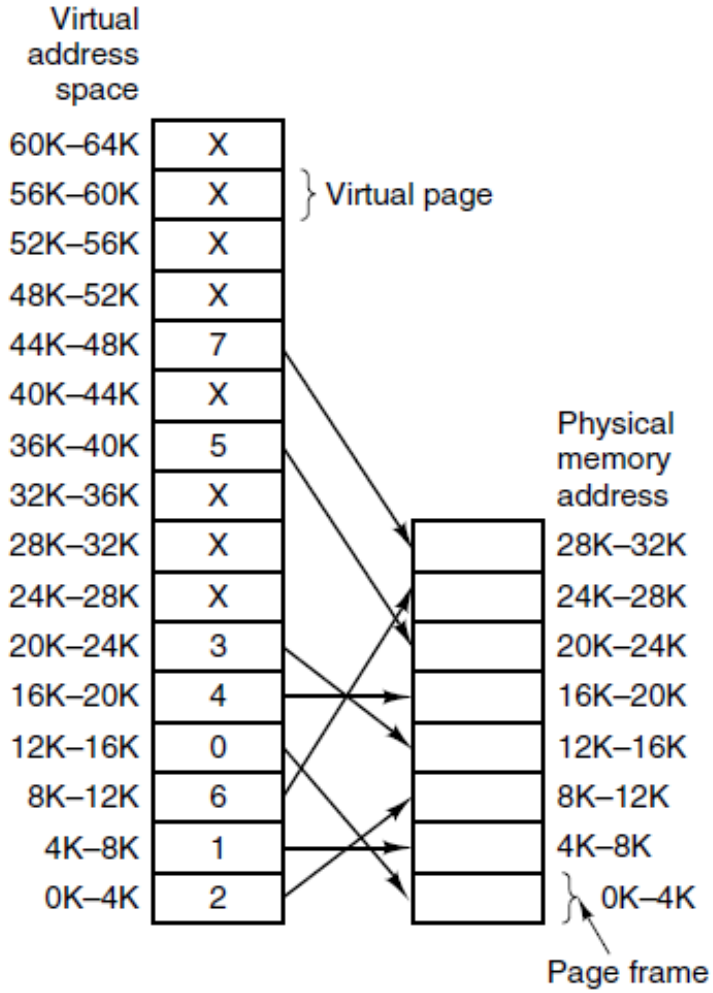
- Address generated by CPU is divided into:
  - Page number (p) – used as an index into a page table which contains base address of each page in physical memory
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit



  - For given logical address space 2^m and page size 2^n

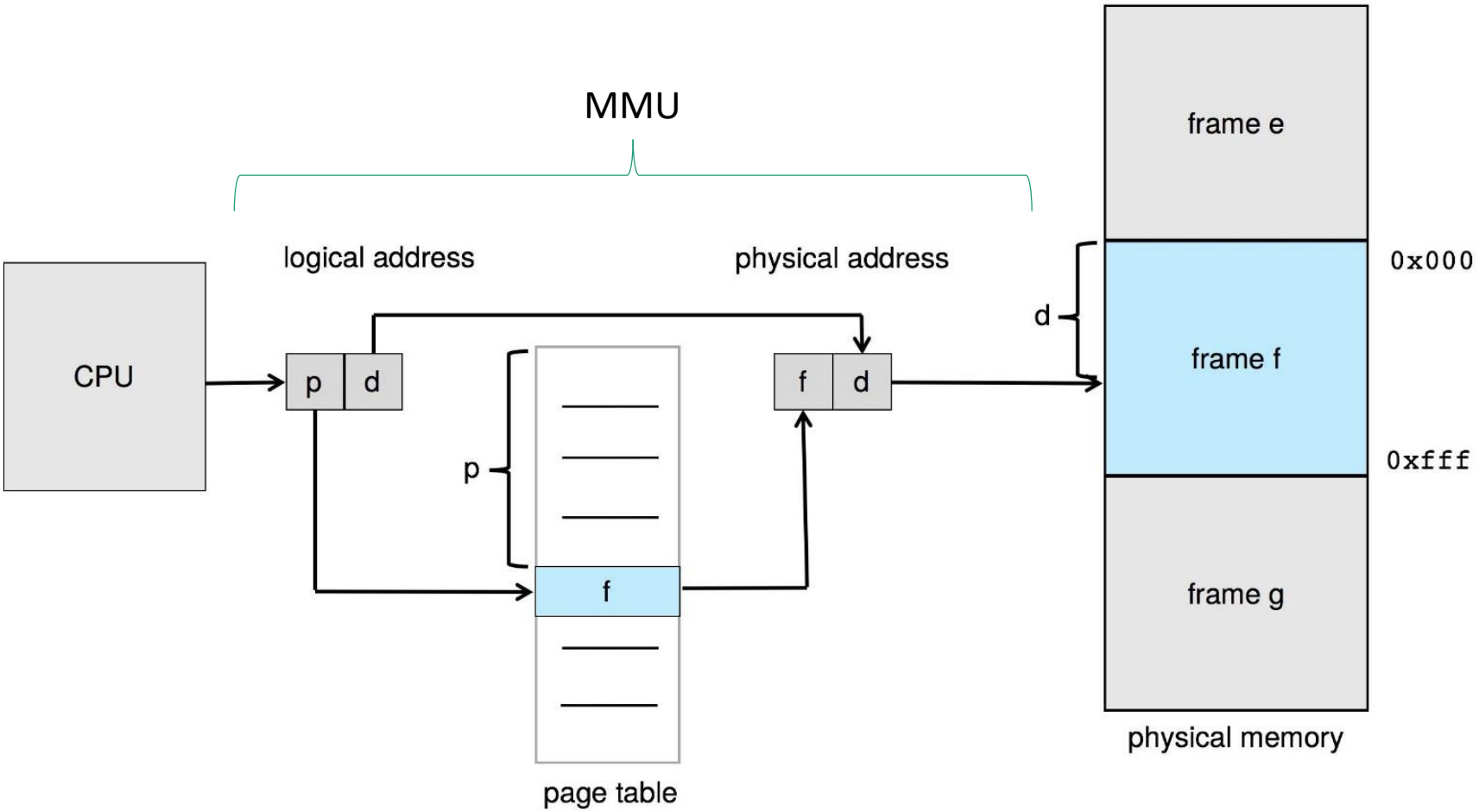
Example: p, d, m, and n?

- MMU maintains a map per process
  - Page size: 4K
- What if
  - MOV REG, (8203)
- 8203 is a virtual address, passed to MMU (8K = 8192)
  - determines that 8203 is in page 2 in virtual address space
  - determines that the page is mapped to page frame 6 in physical memory
  - Maps the virtual address to physical address
    - 8203 / 4K = 2 (table lookup → 6)
    - 8203 % 4K + 6 * 4K = 24587



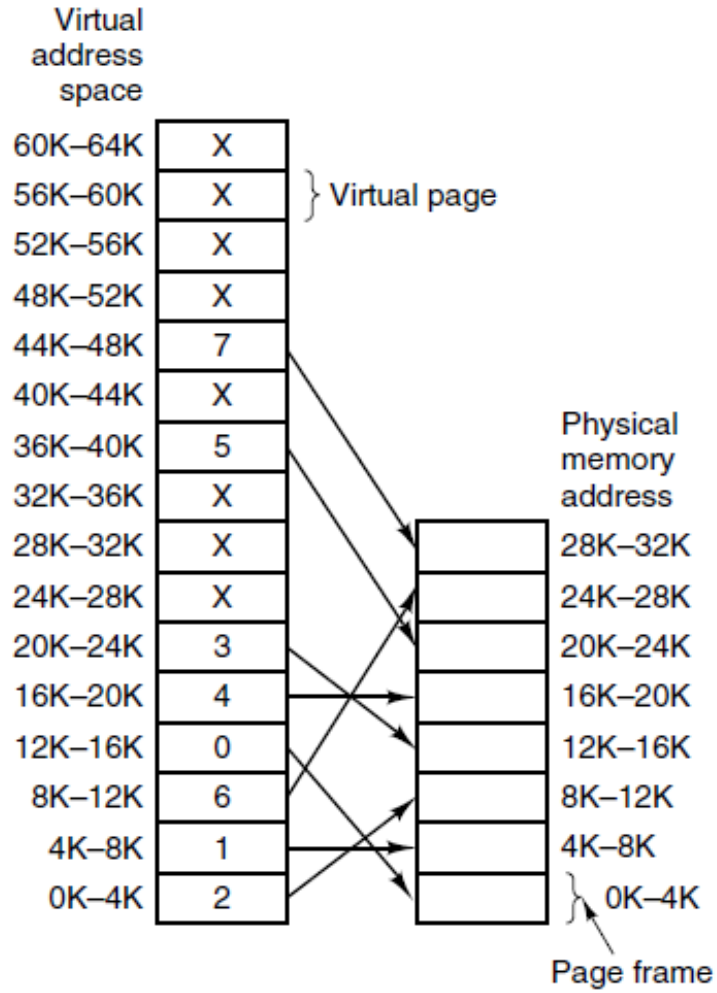
- [Figure 3-9 in Tanenbaum & Bos, 2014]

Paging Hardware



Example: Constructing Page Table

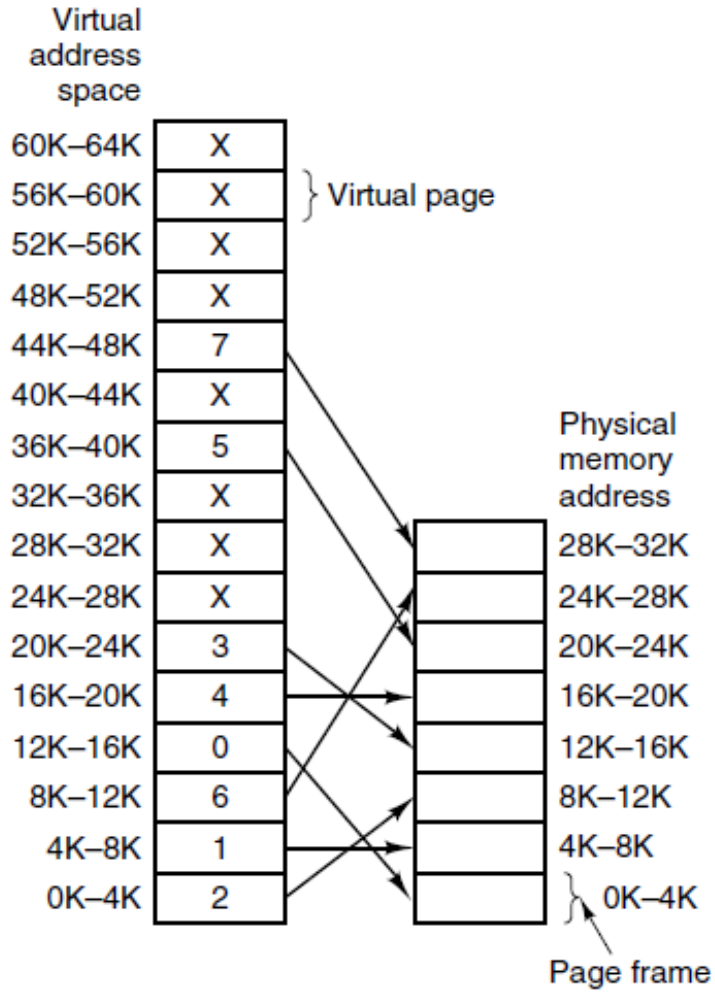
Page number (p)	Frame Number (f)



- [Figure 3-9 in Tanenbaum & Bos, 2014]

Example: Constructing Page Table

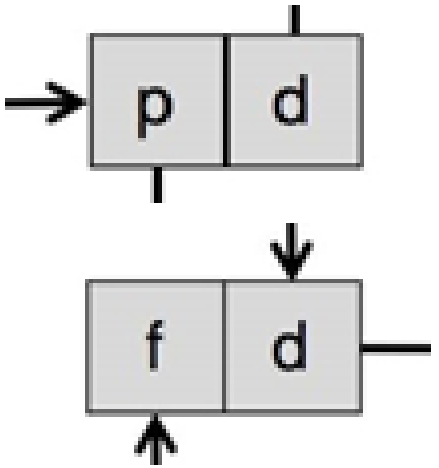
Page number (p)	Frame Number (f)
3 $(0011)_2$	0 $(000)_2$
1 $(0001)_2$	1 $(001)_2$
0 $(0000)_2$	2 $(010)_2$
5 $(0101)_2$	3 $(011)_2$
4 $(0100)_2$	4 $(100)_2$
9 $(1001)_2$	5 $(101)_2$
2 $(0010)_2$	6 $(110)_2$
11 $(1011)_2$	7 $(111)_2$



- [Figure 3-9 in Tanenbaum & Bos, 2014]

Paging Hardware: Example

- Page & frame sizes: 4K, so d is 12 bits
- Logical address space: 64K
  - 64K / 4K = 16, so p is 4 bits
  - p d: 4 + 12 = 16 bits
- Physical address space: 32K
  - 32K / 4K = 8, so f is 3 bits
  - f d: 3 + 12 = 15 bits
- Then, consider MOV REG, (8203)
  - 8203_{10} = 0010 0000 0000 1011

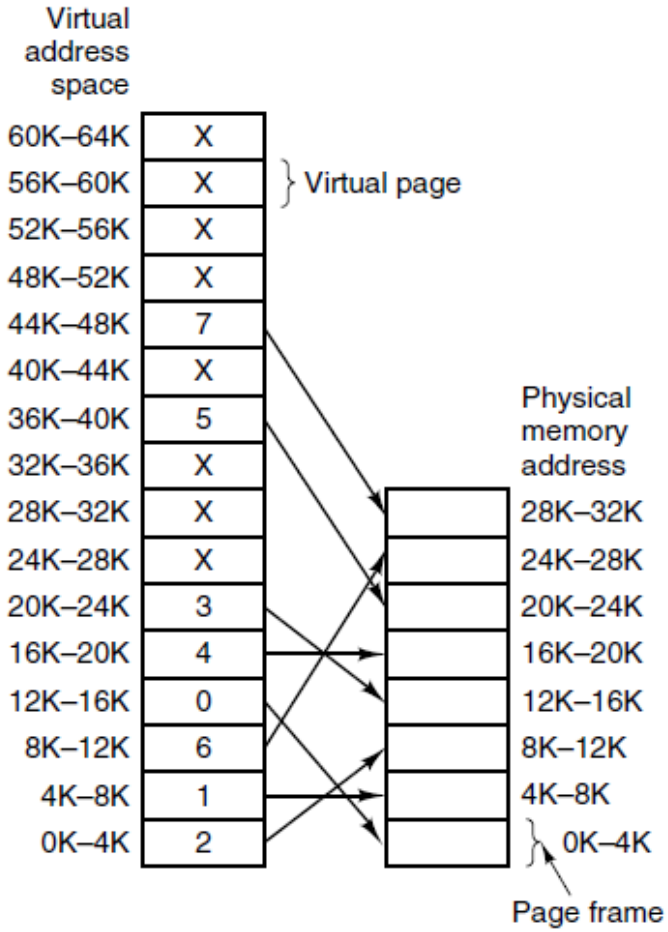


Paging Hardware: Example: Using p to Look up f

Page number (p)	Frame Number (f)
3 $(0011)_2$	0 $(000)_2$
1 $(0001)_2$	1 $(001)_2$
0 $(0000)_2$	2 $(010)_2$
5 $(0101)_2$	3 $(011)_2$
4 $(0100)_2$	4 $(100)_2$
9 $(1001)_2$	5 $(101)_2$
2 $(0010)_2$	6 $(110)_2$
11 $(1011)_2$	7 $(111)_2$

8203_{10} = 0010 0000 0000 1011

?



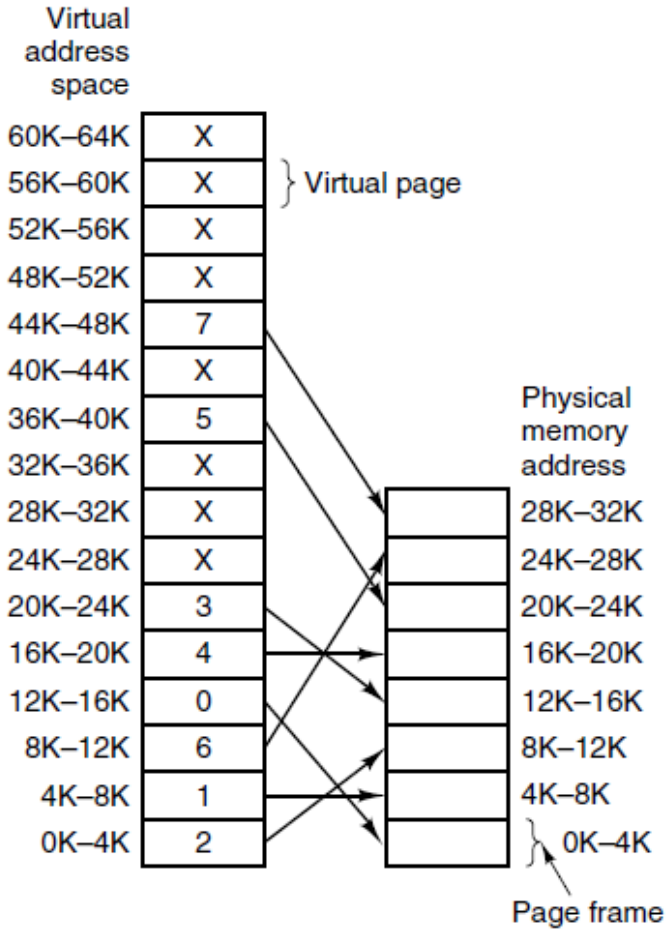
- [Figure 3-9 in Tanenbaum & Bos, 2014]

Paging Hardware: Example: Using p to Look up f

Page number (p)	Frame Number (f)
3 $(0011)_2$	0 $(000)_2$
1 $(0001)_2$	1 $(001)_2$
0 $(0000)_2$	2 $(010)_2$
5 $(0101)_2$	3 $(011)_2$
4 $(0100)_2$	4 $(100)_2$
9 $(1001)_2$	5 $(101)_2$
2 $(0010)_2$	6 $(110)_2$
11 $(1011)_2$	7 $(111)_2$

8203_{10} = 0010 0000 0000 1011

110



- [Figure 3-9 in Tanenbaum & Bos, 2014]

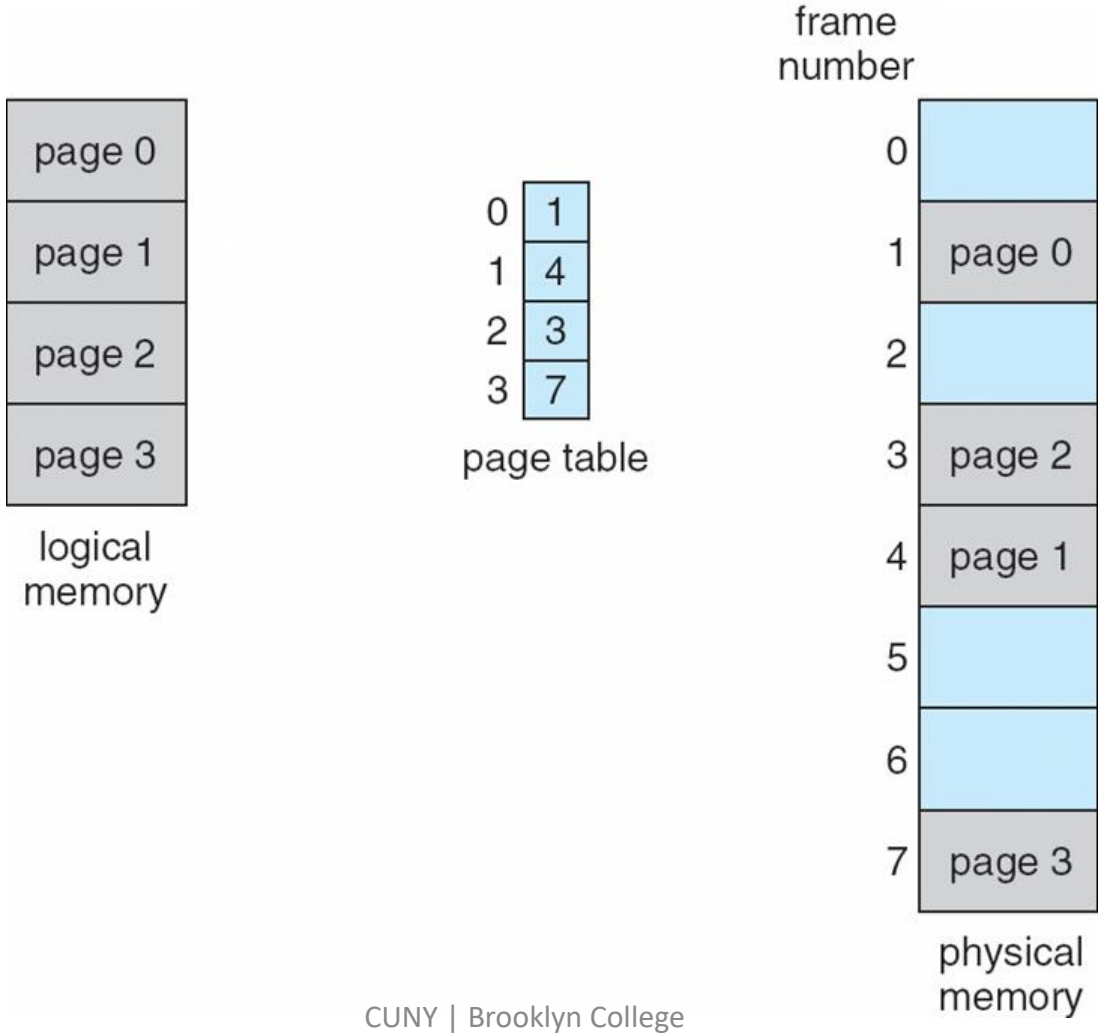
Paging Hardware: Example: f d = ?

8203_{10} = 0010 0000 0000 1011

110

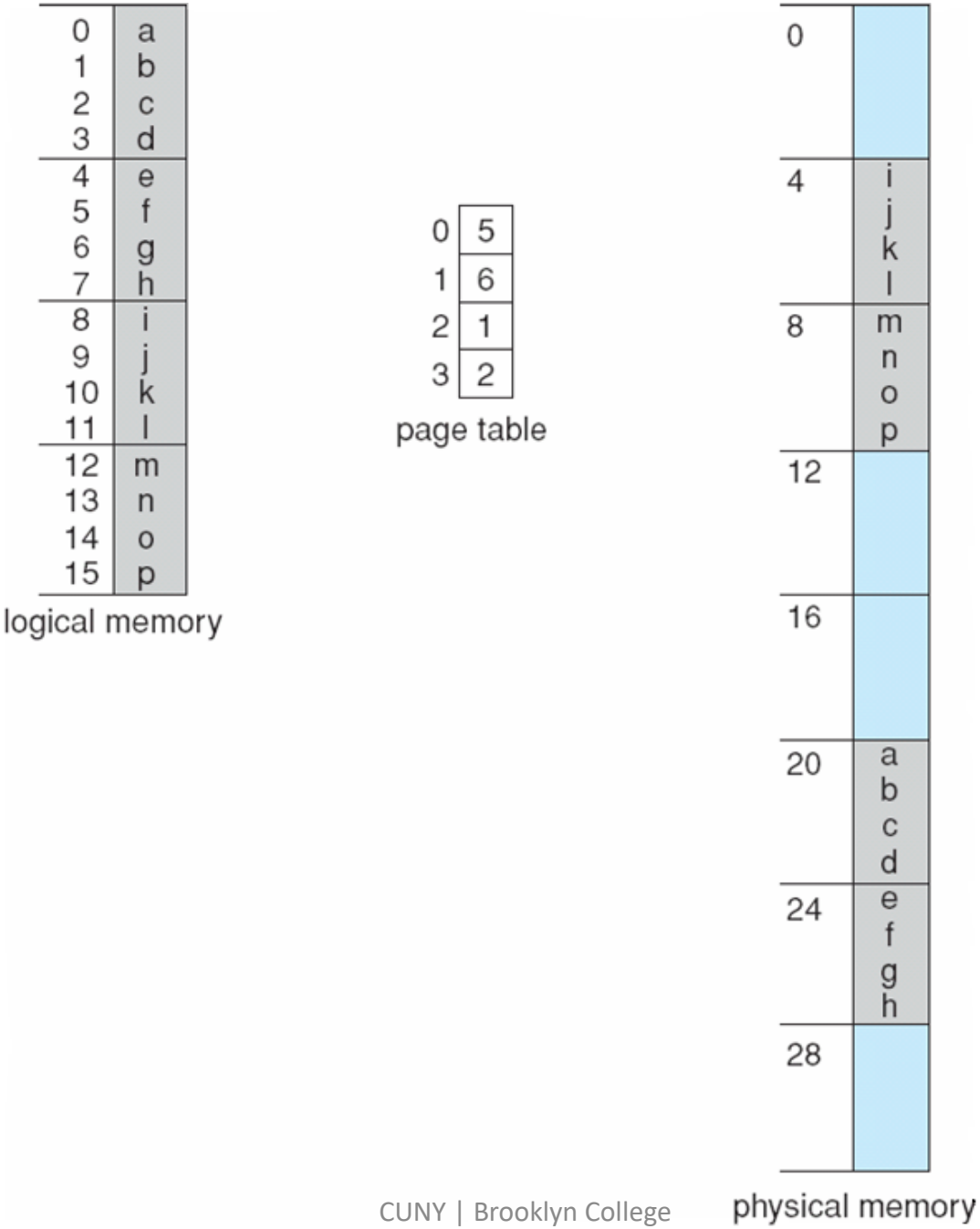
f d = 110 0000 0000 1011 = ?

More Paging Examples



More Paging Examples

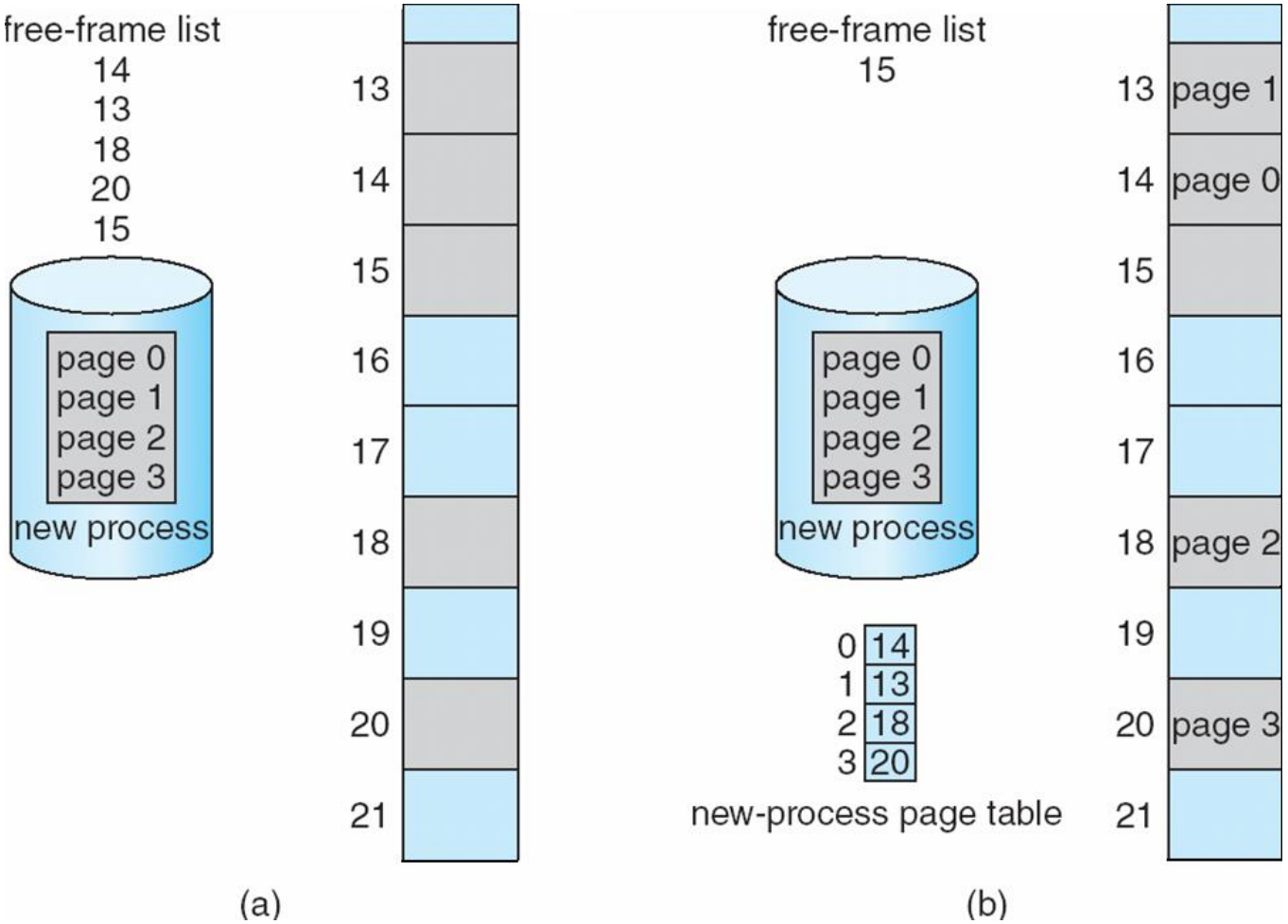
- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)


0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
8 i
9 j
10 k
11 l
12 m
13 n
14 o
15 p
logical memory
0 5
1 6
2 1
3 2
page table
0
4 i j k l
8 m n o p
12
16
20 a b c d
24 e f g h
28
physical memory

Paging: Calculating Internal Fragmentation

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

Free Frames



Before allocation

After allocation

Questions?

- Paging and examples?
- Page and frame
- Page table
- Internal fragmentation
- Allocating and freeing frames

Implementation of Page Table

- Page table is kept in main memory
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**) (also called **associative memory**).

Translation Look-Aside Buffer

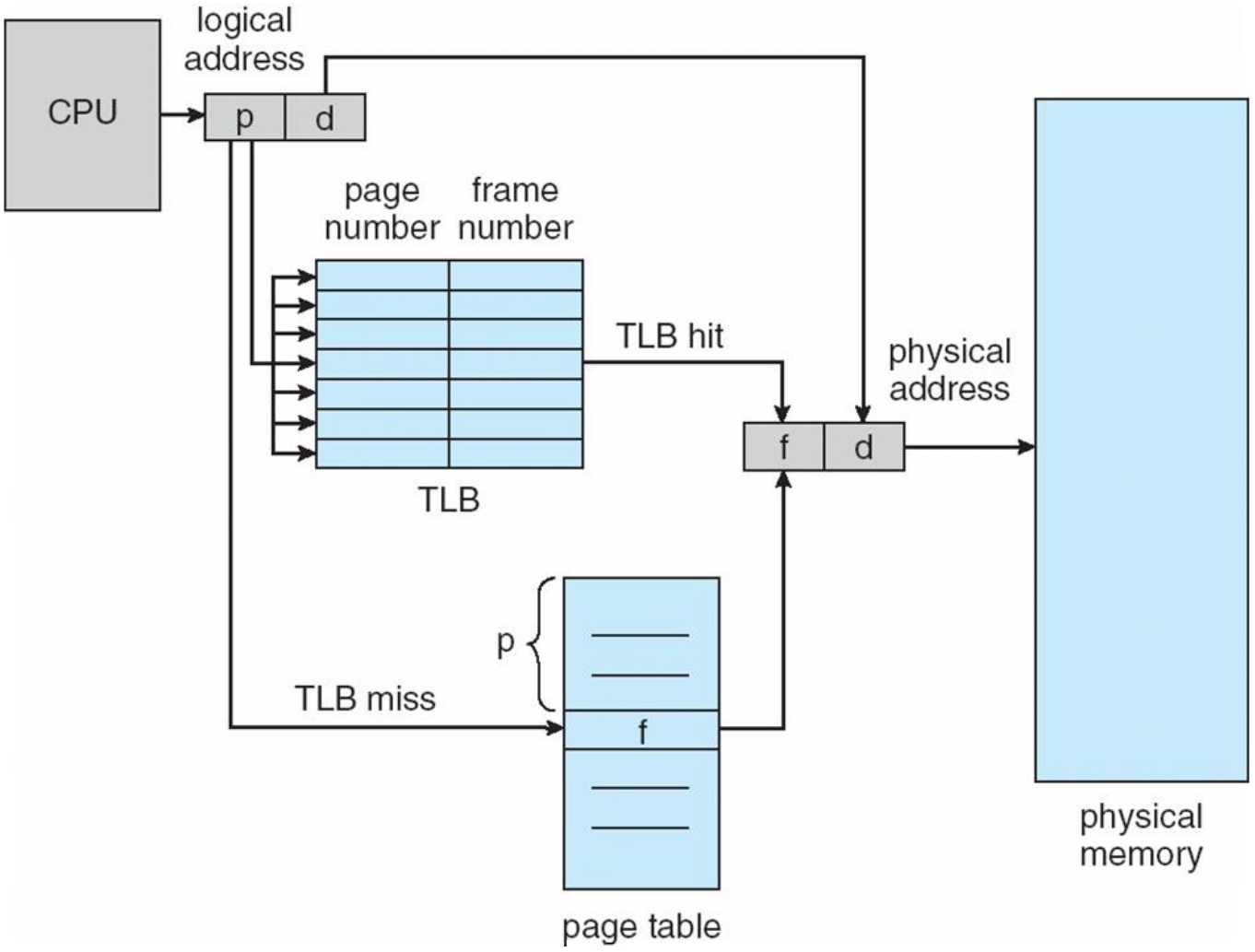
- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

Hardware

- Associative memory – parallel search

- 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

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
- If we find the desired page in TLB then a mapped-memory access take 10 ns
- Otherwise we need two memory access so it is 20 ns
- Effective Access Time (EAT)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,

  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time.

Questions?

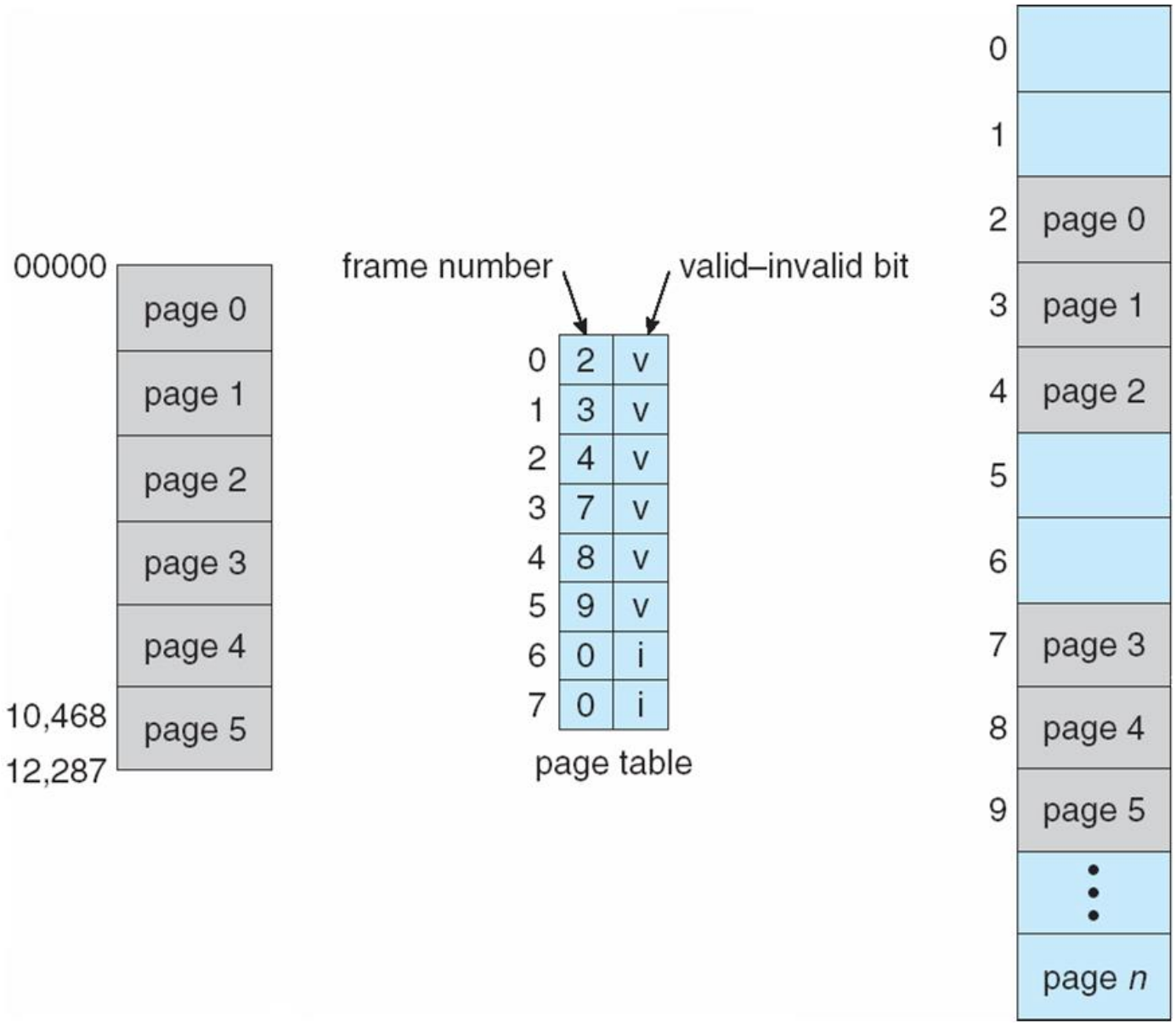
- How to speed up frame lookup?

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate 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 page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

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

- An example


00000
page 0
page 1
page 2
page 3
page 4
page 5
10,468
12,287
frame number
valid–invalid bit
0 2 v
1 3 v
2 4 v
3 7 v
4 8 v
5 9 v
6 0 i
7 0 i
page table
0
1
2 page 0
3 page 1
4 page 2
5
6
7 page 3
8 page 4
9 page 5
page n

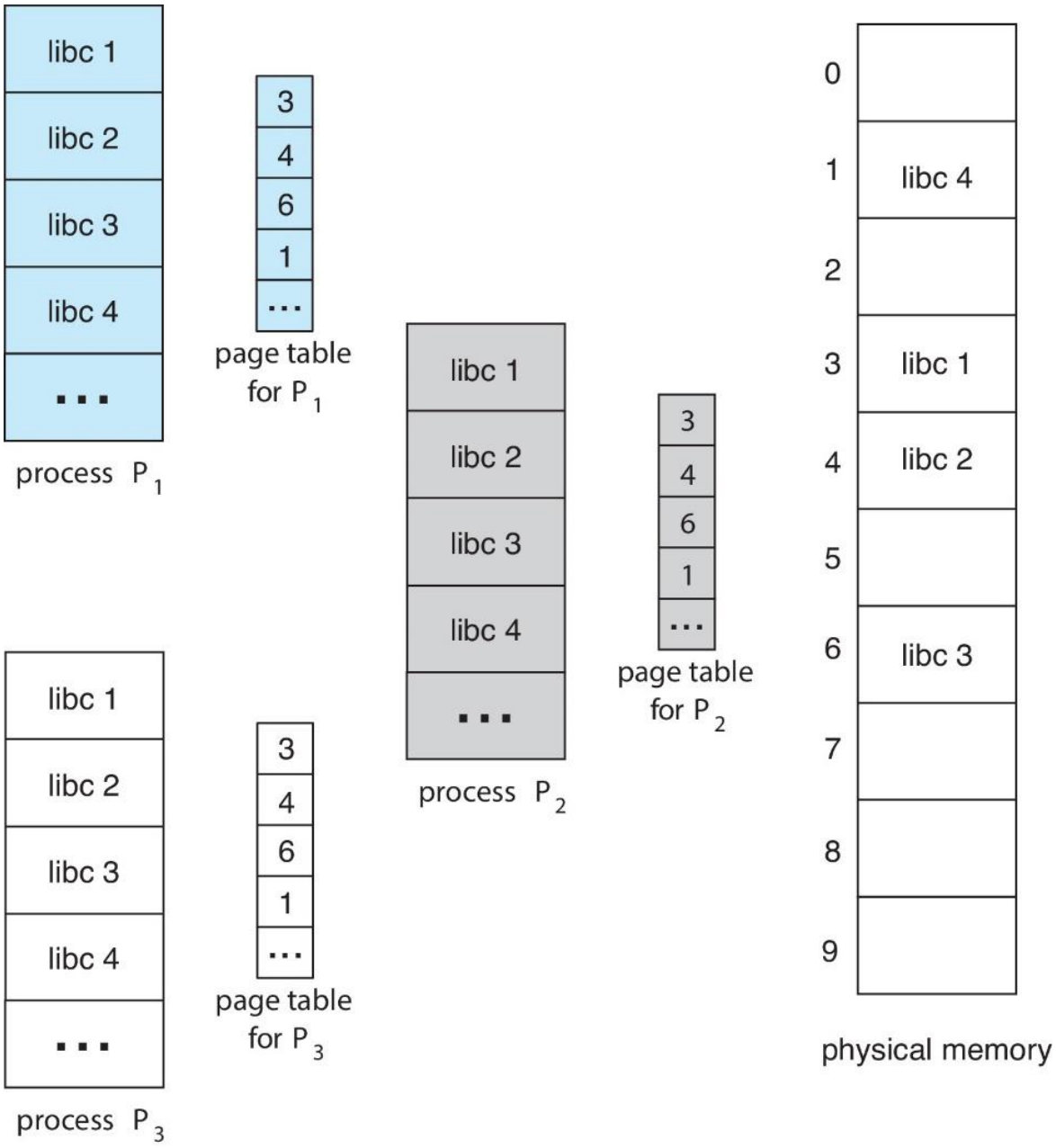
Questions?

- Memory protection?

Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages: Example


libc 1
libc 2
libc 3
libc 4
...
process P1
3
4
6
1
...
page table for P1
libc 1
libc 2
libc 3
libc 4
...
process P2
3
4
6
1
...
page table for P2
libc 1
libc 2
libc 3
libc 4
...
process P3
3
4
6
1
...
page table for P3
0
1
libc 4
2
3
libc 1
4
libc 2
5
6
libc 3
7
8
9
physical memory

Questions

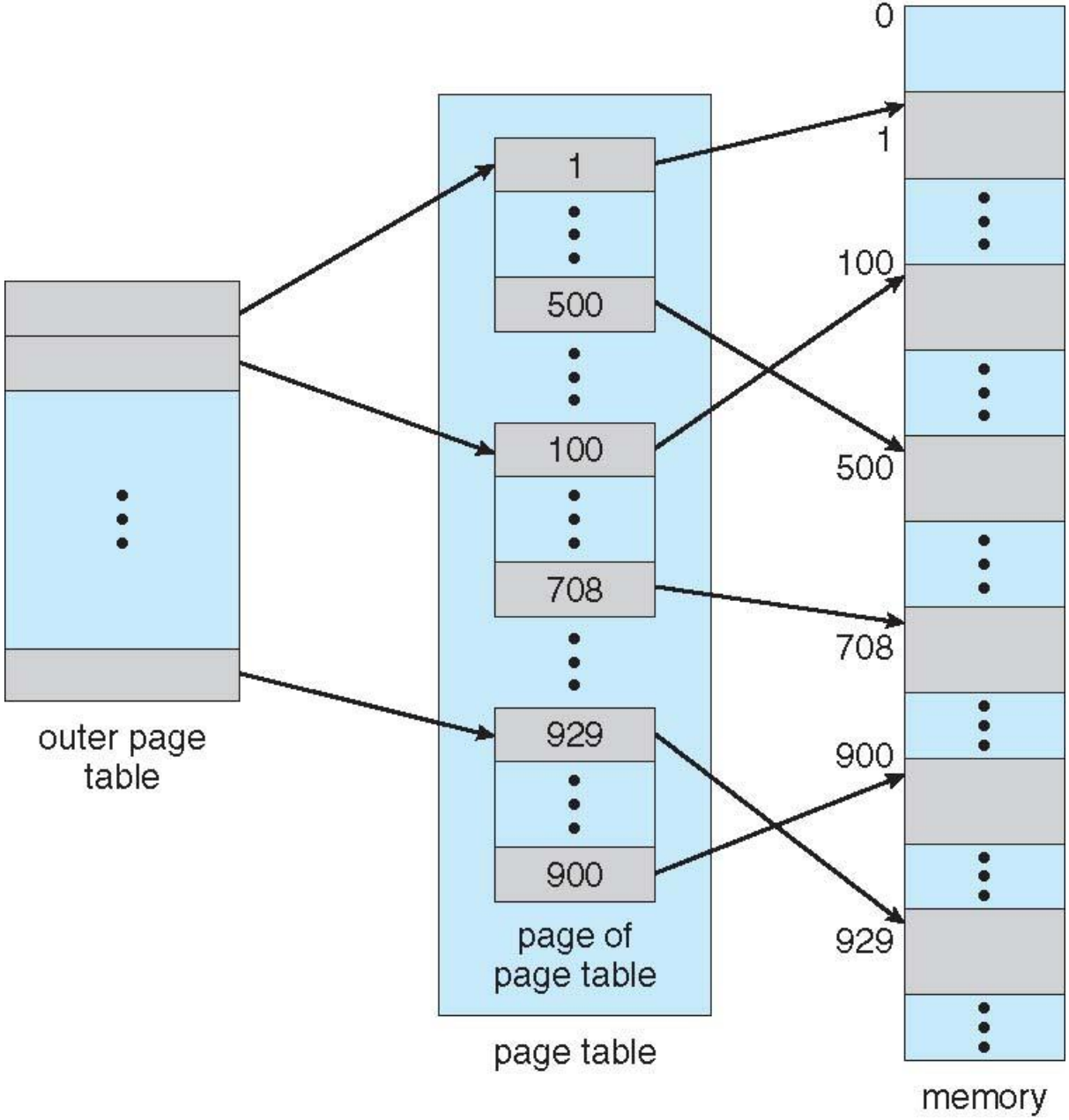
- Shared pages?

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
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes ➔ each process 4 MB of physical address space for the page table alone
    - Don't want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - Hierarchical Paging
    - Hashed Page Tables
    - Inverted Page Tables

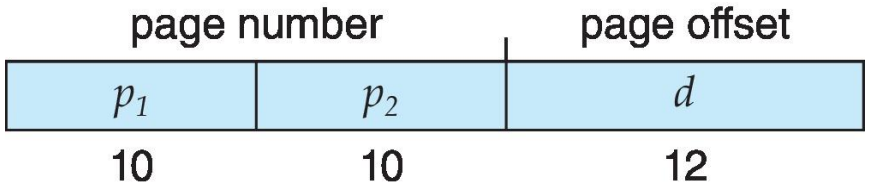
Hierarchical Page Tables

- 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


0
1
100
500
708
900
929
outer page table
1
500
100
708
929
900
page of page table
page table
memory

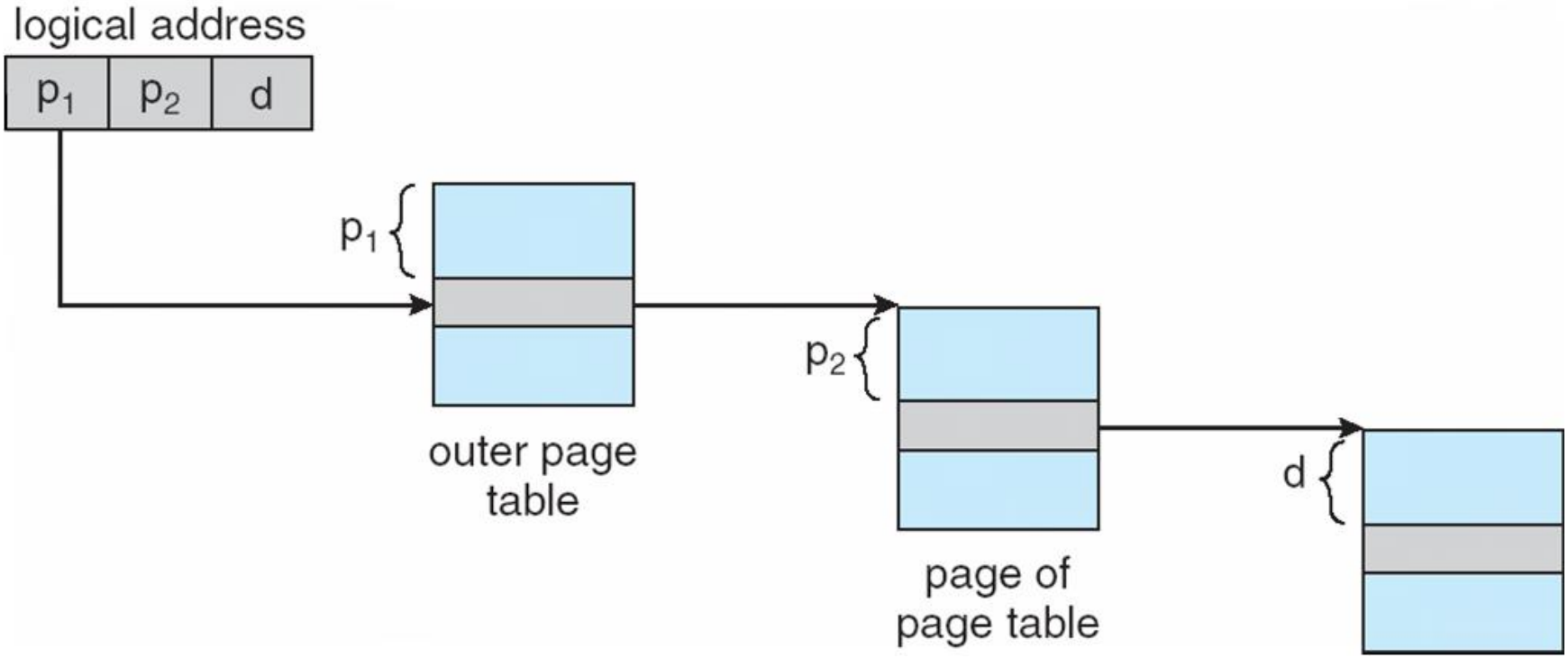
Two-Level Paging: Example

- 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 10-bit page number
  - a 12-bit page offset



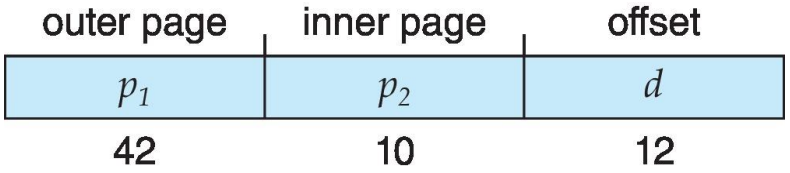
- Thus, a logical address is as follows:
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



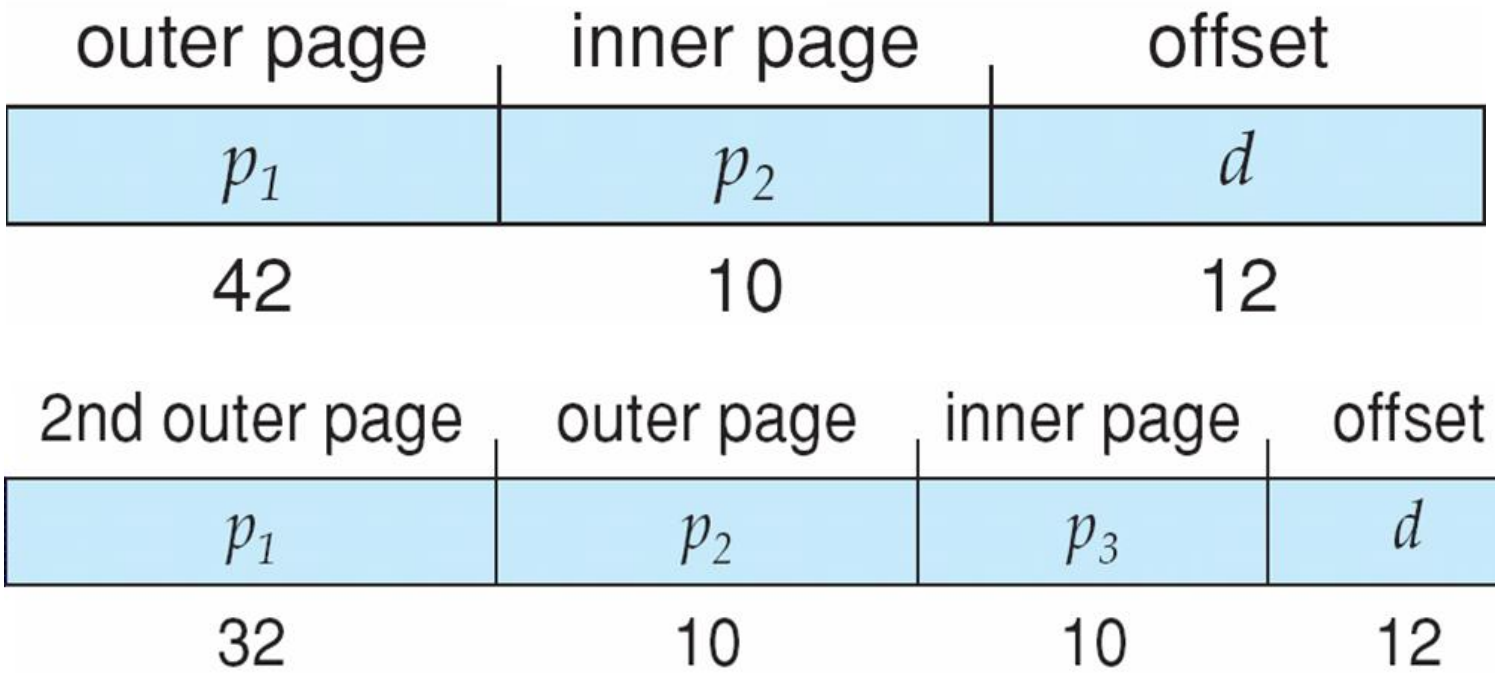
64-bit Logical Address Space

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



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

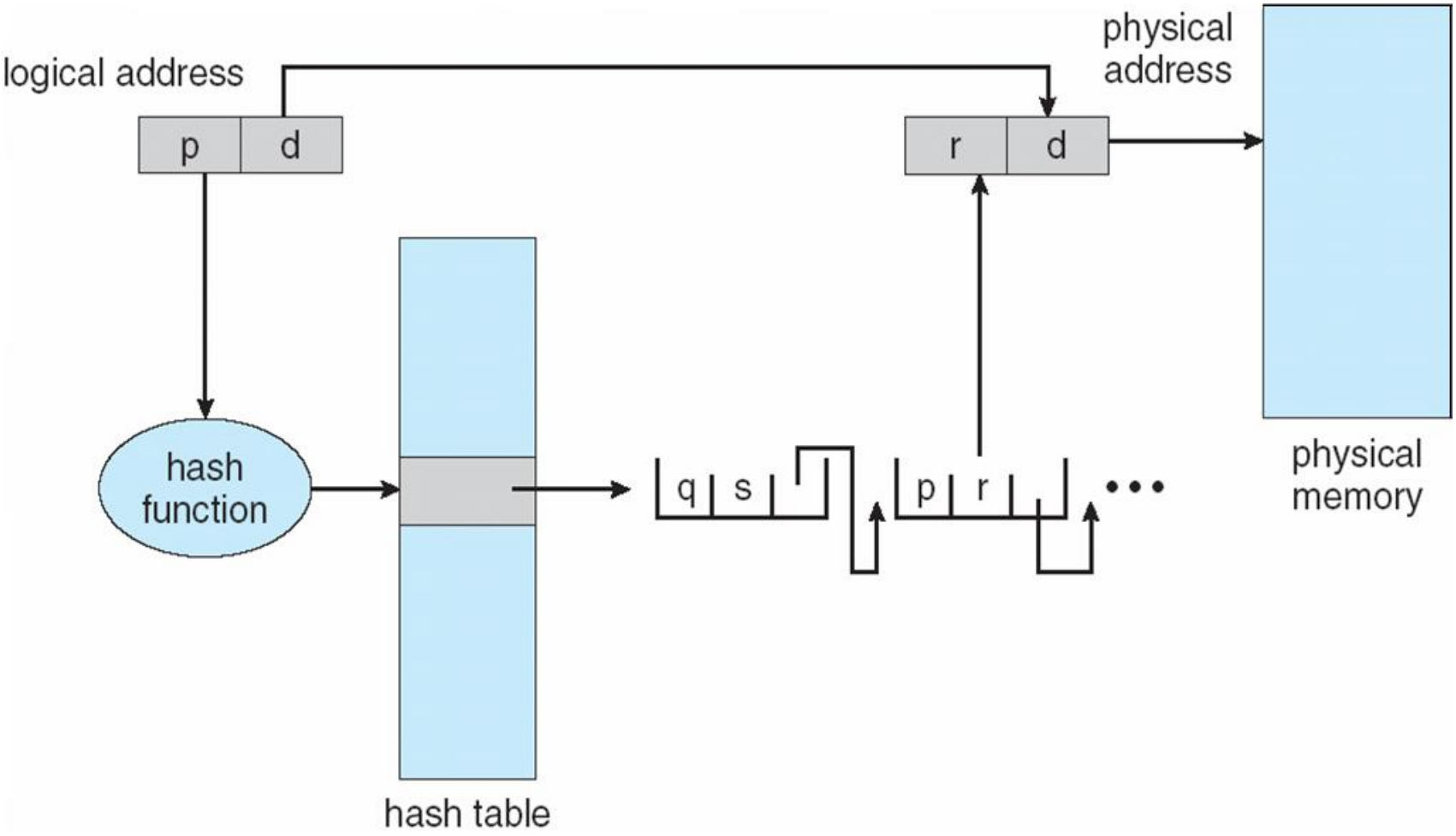
Three-Level Paging Scheme



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 page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

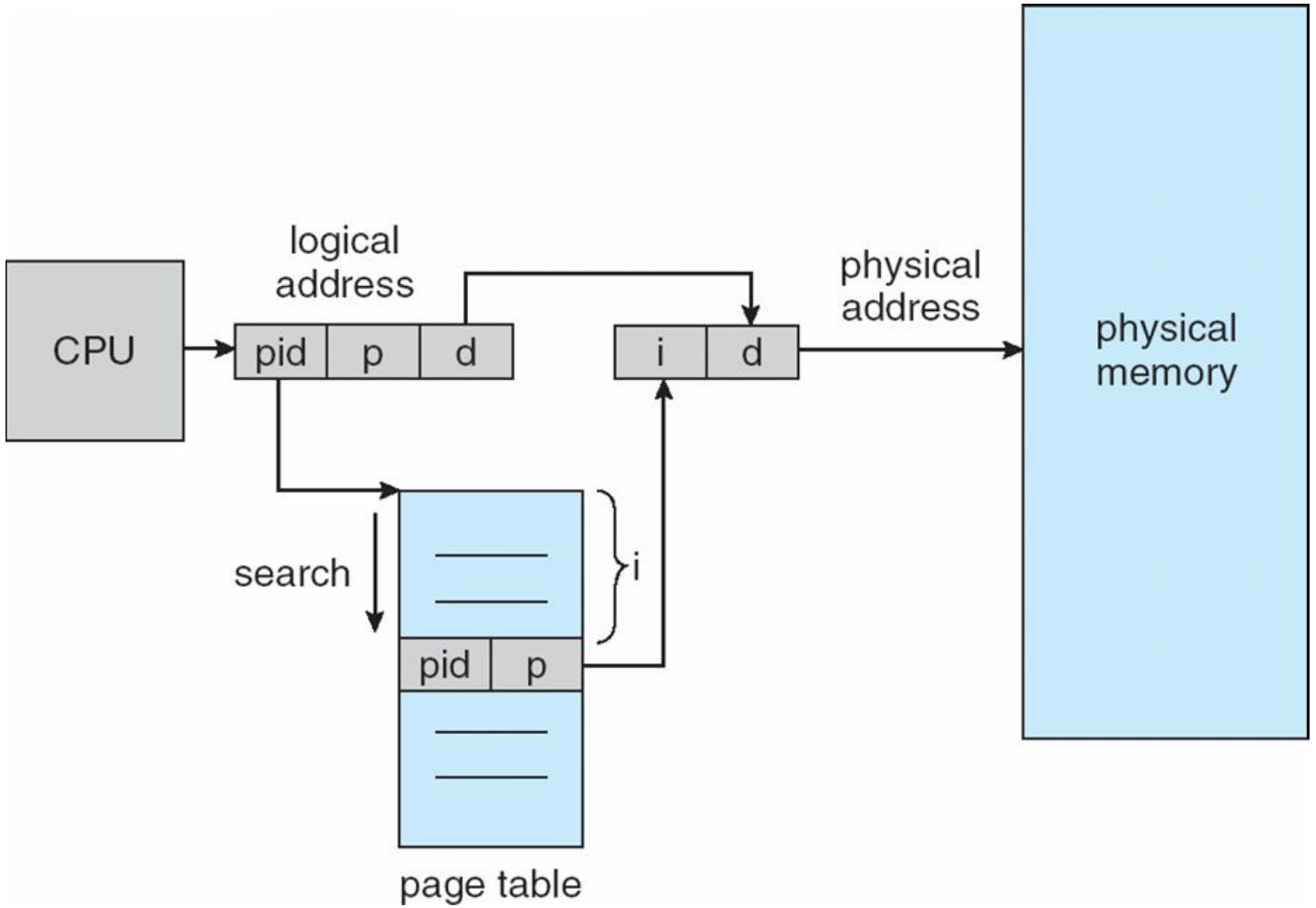
Hashed Page Table



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- 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
- Decreases memory needed to store each page table, but 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
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)

Oracle SPARC Solaris

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

Questions?

- Page table too large?
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables