CISC 3320

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 via the publisher of the textbook

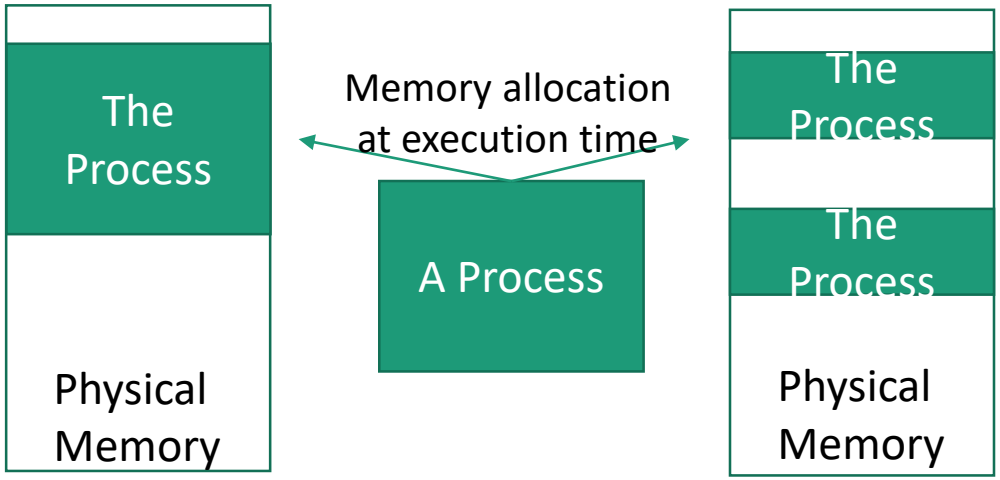
Outline

- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

Noncontiguous Allocation: Paging

- A memory allocation scheme where physical address space of a process can be *noncontiguous*

Continuous vs. Noncontinuous allocation of physical memory



Frames and Pages

- Divide *physical addresses* into fixed-sized blocks called *frames*
  - Size is power of 2, typically between 512 bytes and 16 Mbytes
- Divide *logical addresses* into blocks of same size called *pages*

Paging: Basic Scheme

- OS keeps track of all free frames
- Process is allocated physical memory whenever there is available physical memory

Page Table

- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
  - Map N pages to N frames
- Need to set up a page table to translate logical to physical addresses for a process

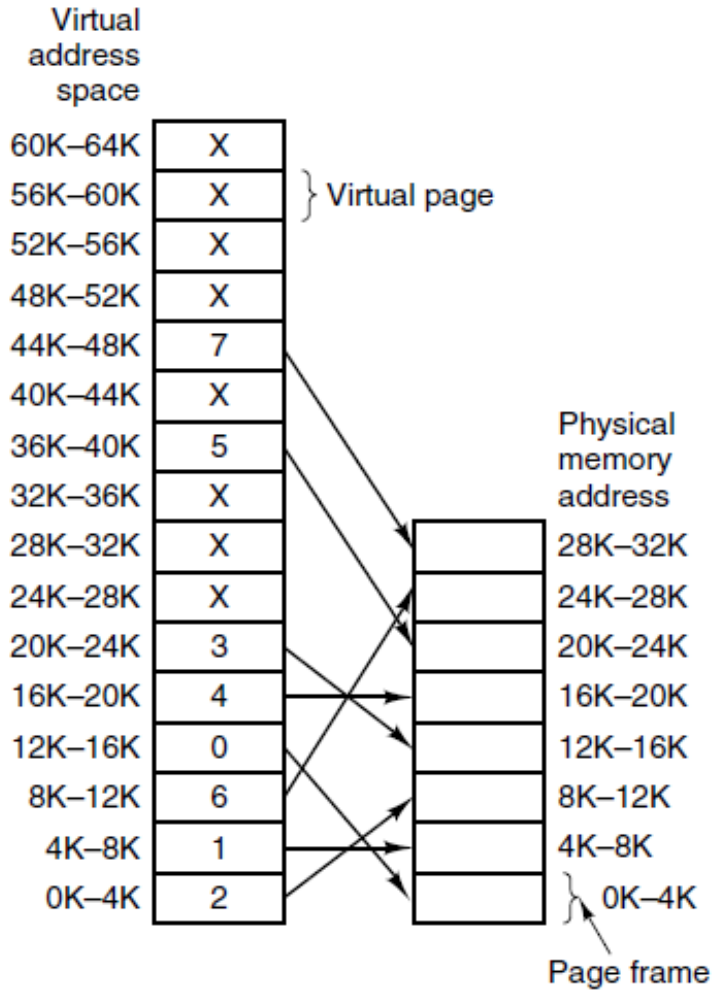
N pages

Page Number	Frame Number
1	3
…	

N frames

Example: Basic Paging Scheme

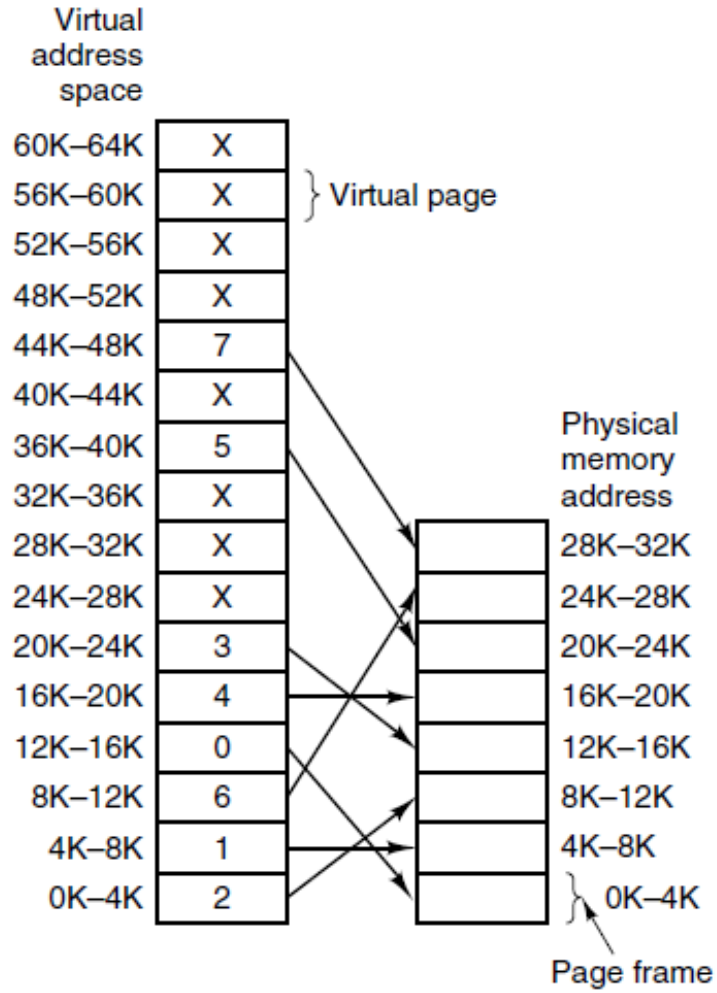
- Virtual/logical address
  - Per process
  - 16-bit address
  - Address space:
    - $0 \sim (2^{16} - 1) = 64K - 1$
  - Divided into pages, each 4KB
- 32 KB physical memory
  - Divide into frames, each 4KB
- 64 KB logical address space: 16 x 4 = 64, so 16 pages
- 32 KB physical memory: 8 x 4 = 32, so 8 frames



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

Example: Allocating Memory

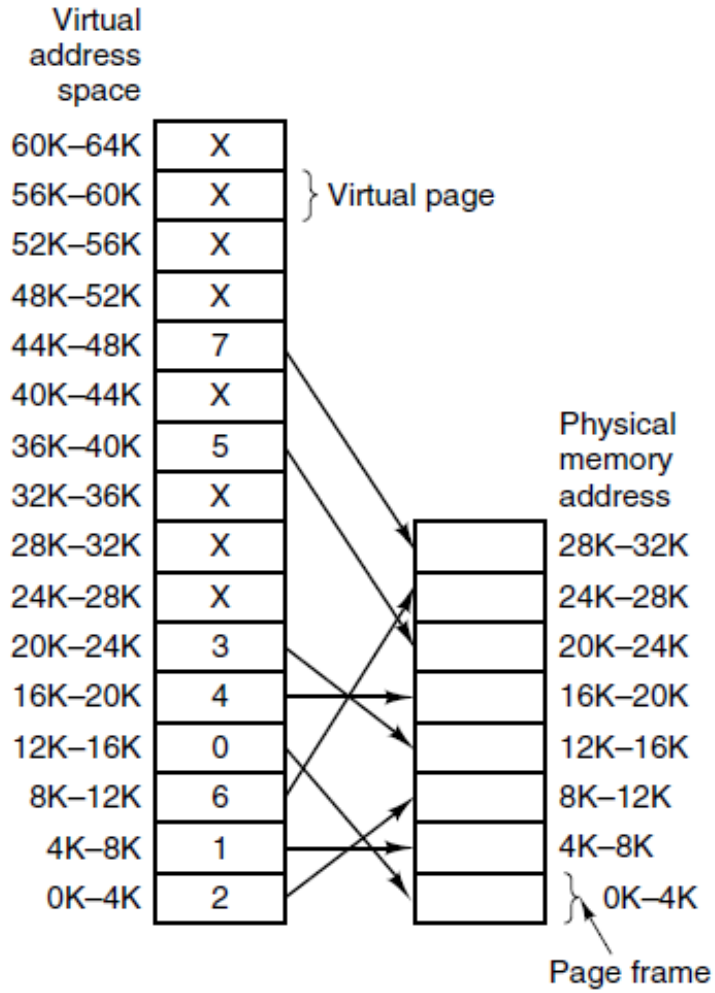
Page number (p)	Frame Number (f)



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

Example: Allocating Memory

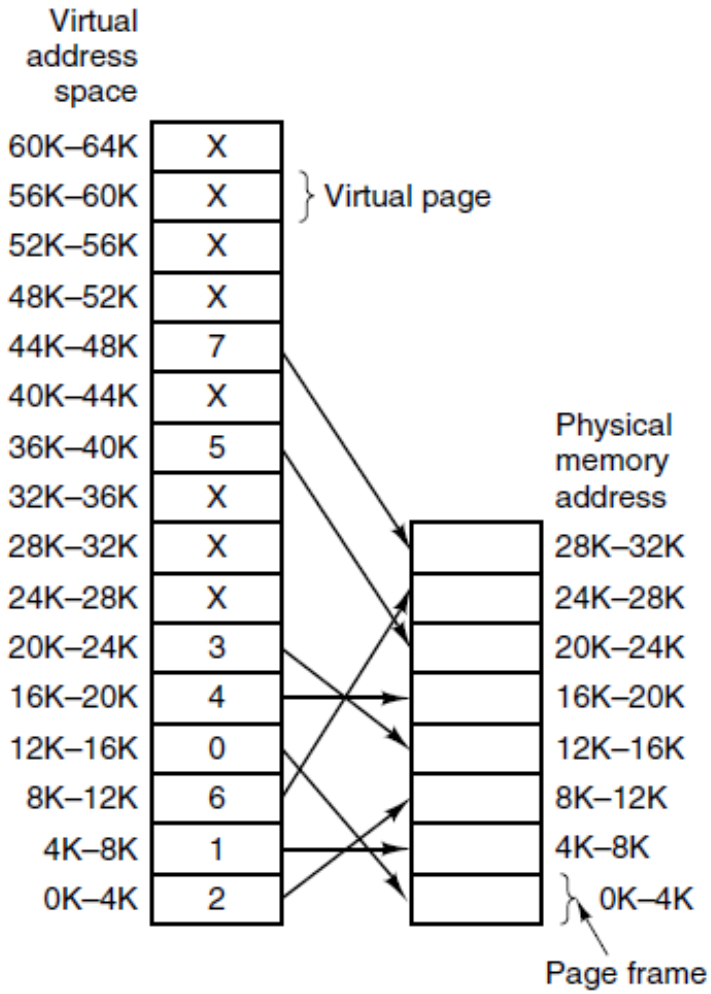
Page number (p)	Frame Number (f)
3 $(0011)_2$	0 $(000)_2$
1 $(0001)_2$	1 $(001)_2$
0 $(0000)_2$	2 $(010)_2$
5 $(0101)_2$	3 $(011)_2$
4 $(0100)_2$	4 $(100)_2$
9 $(1001)_2$	5 $(101)_2$
2 $(0010)_2$	6 $(110)_2$
11 $(1011)_2$	7 $(111)_2$



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

Example: Address Binding

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

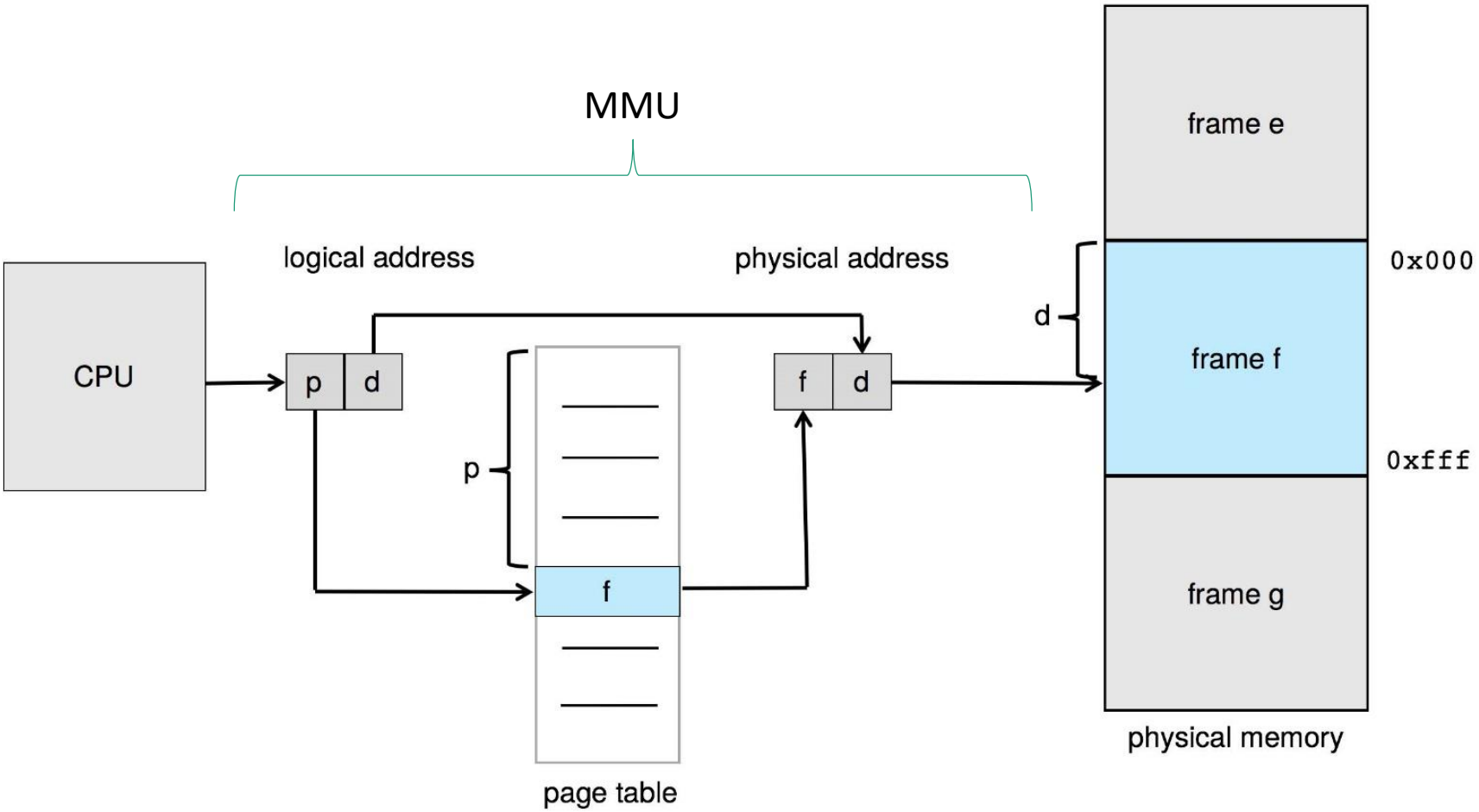


- [Figure 3-9 in Tanenbaum & Bos, 2014]

Questions?

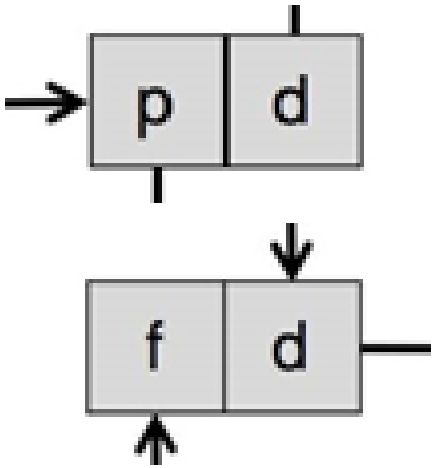
- Paging
  - How does it work?
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Still have Internal fragmentation (some memory may be unused in a frame)

Paging Hardware



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

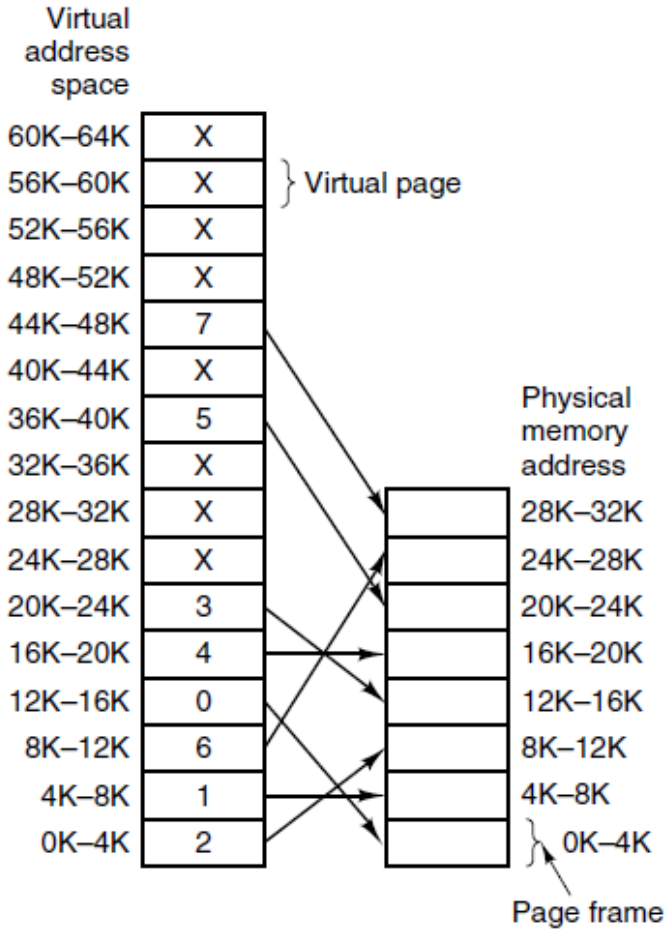


Paging Hardware: Example: Using p to Look up f

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

?



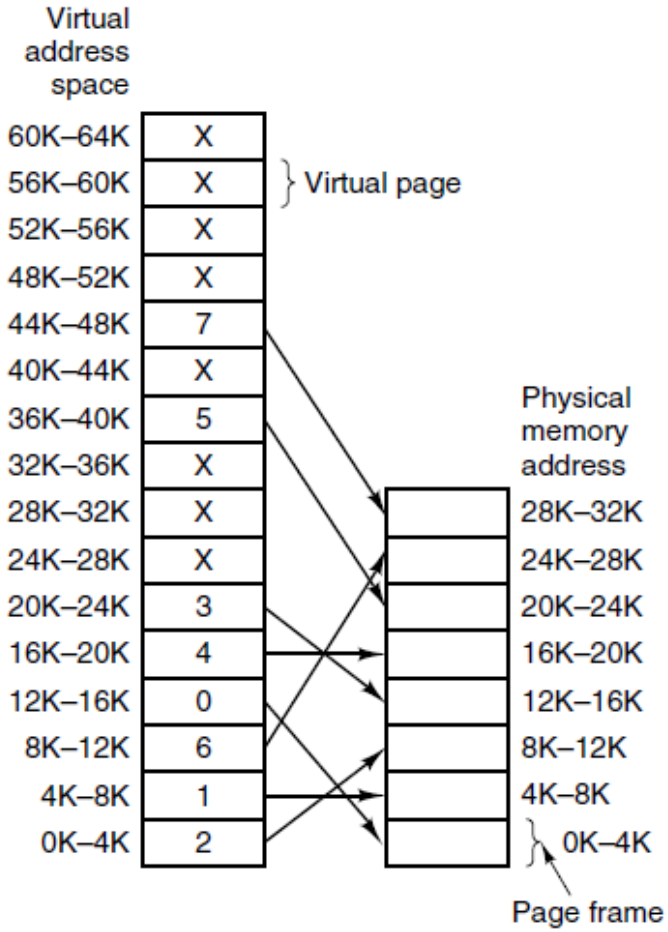
- [Figure 3-9 in Tanenbaum & Bos, 2014]

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]

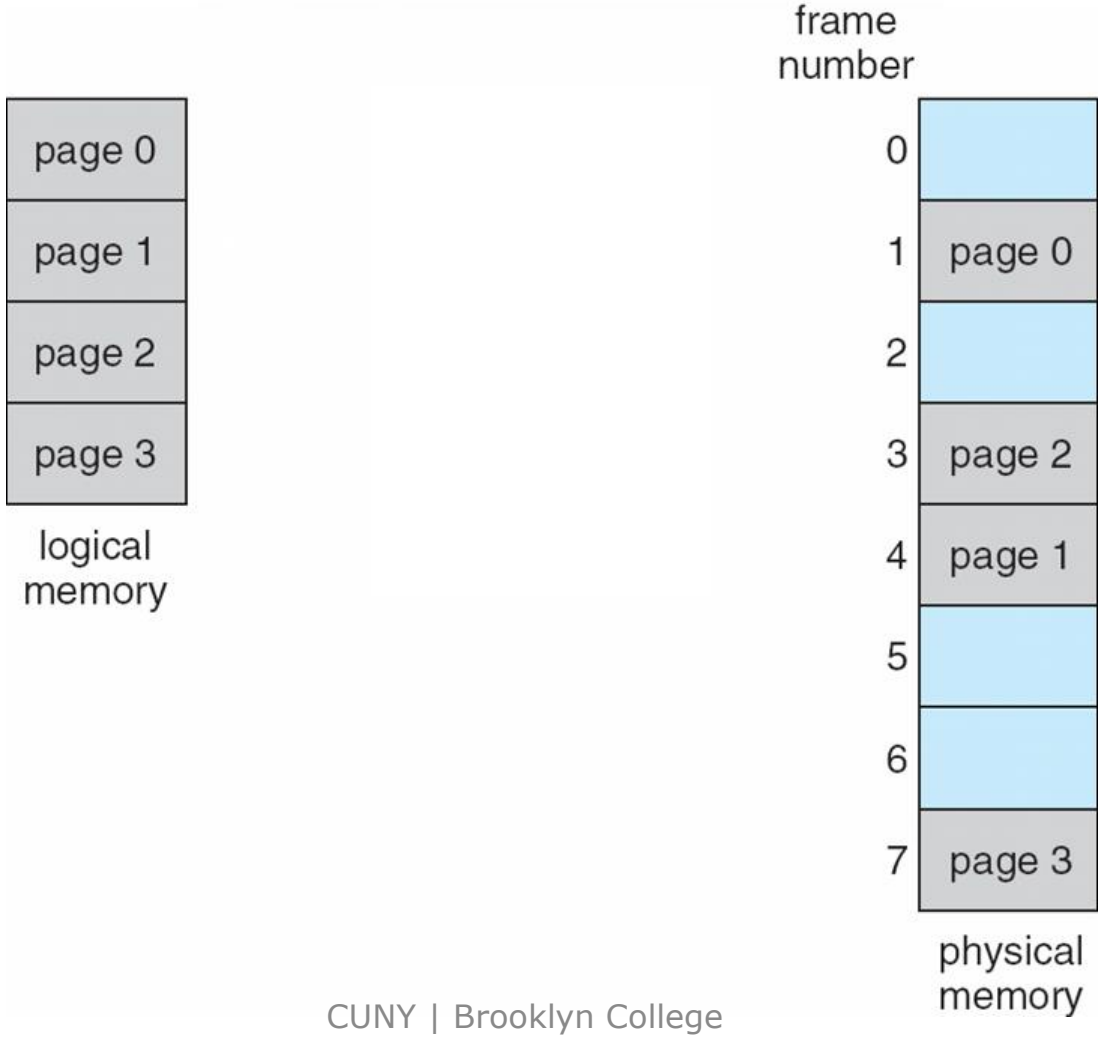
Paging Hardware: Example: f d = ?

8203_{10} = 0010 0000 0000 1011

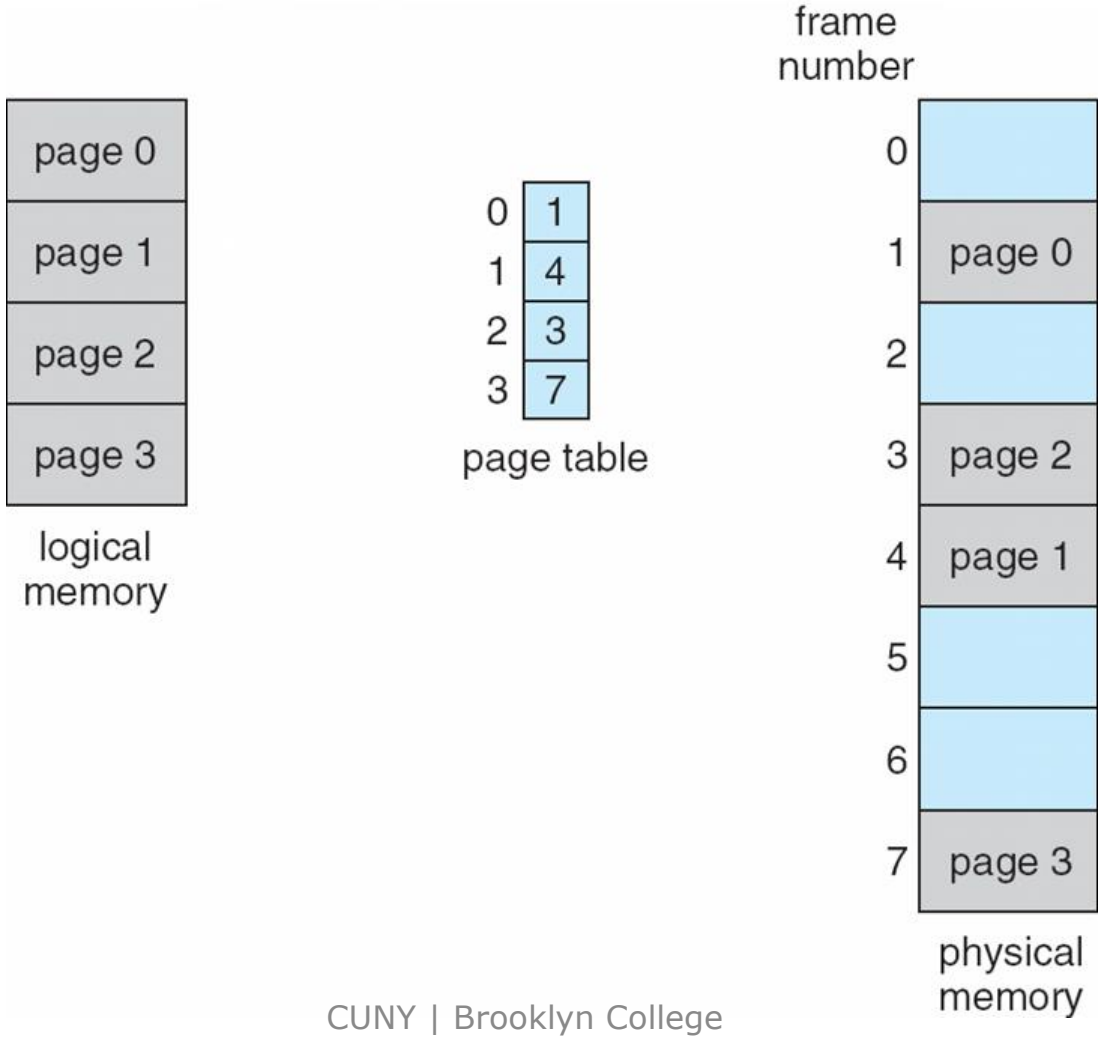
110

f d = 110 0000 0000 1011 = ?

More Paging Examples: Page Table?

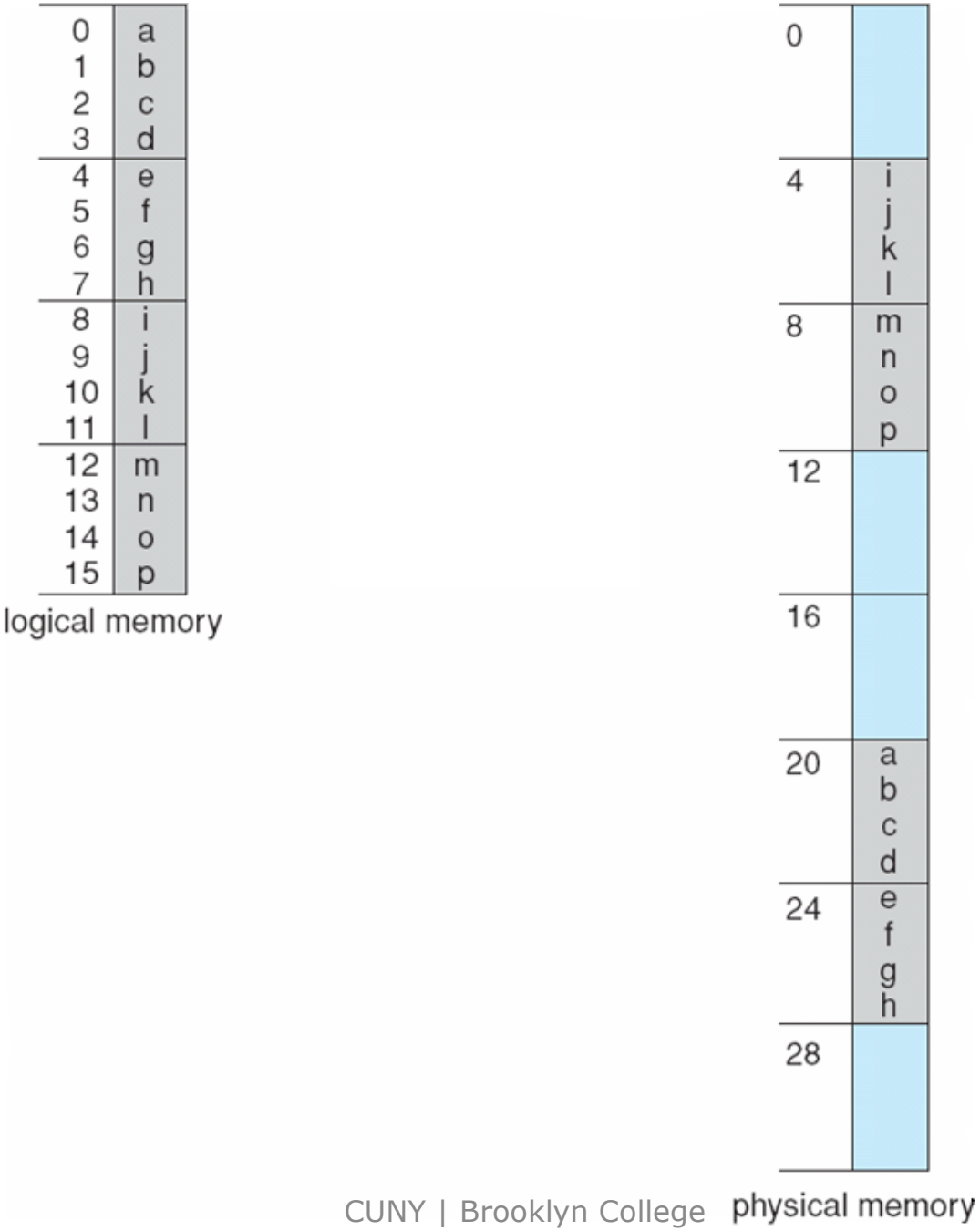


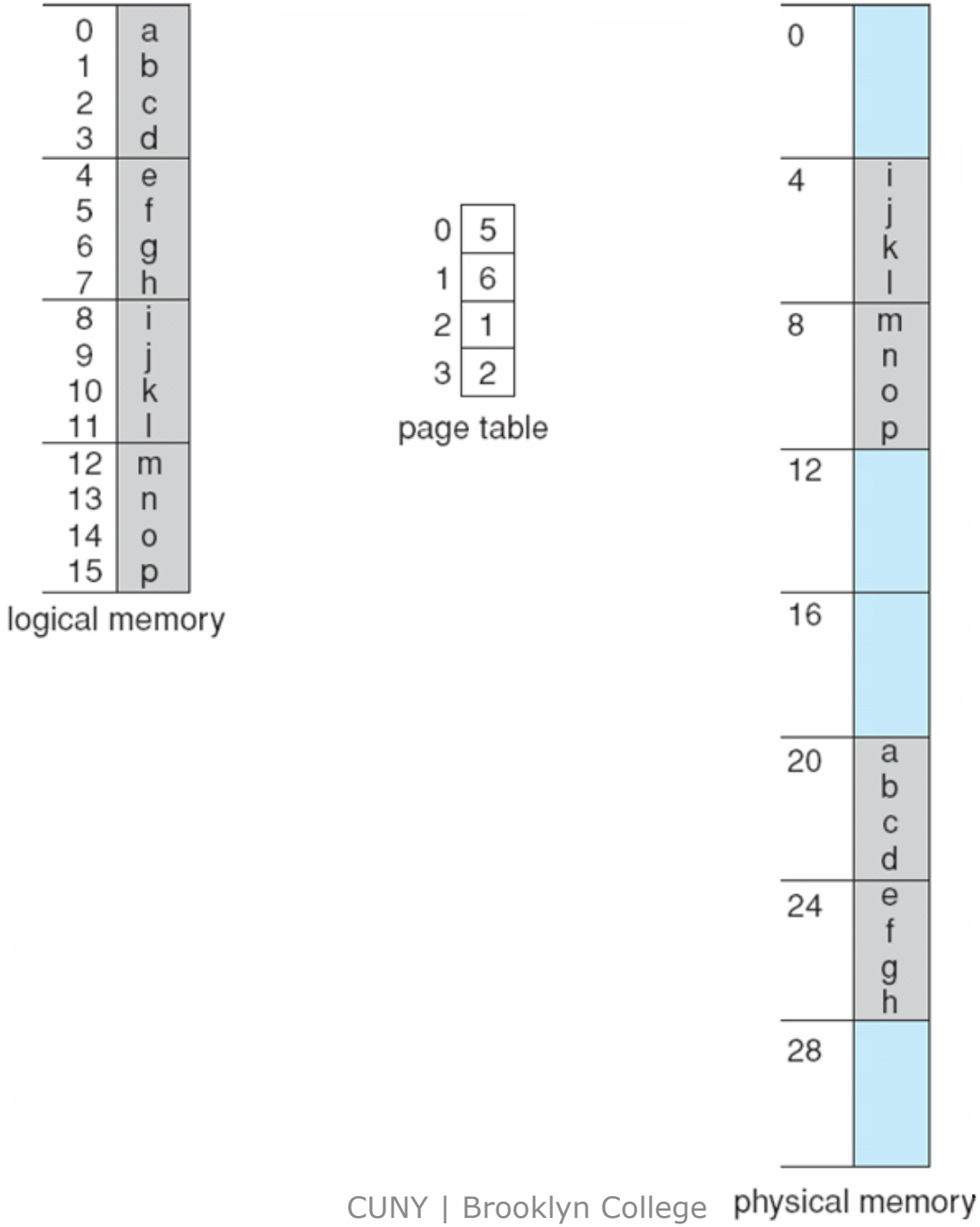
More Paging Examples: Page Table



More Paging Examples: Page Table?

- 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
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


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: Fragmentation

- Avoids external fragmentation
- Still has internal fragmentation

Paging: Internal Fragmentation: Example

- Page size = 2,048 bytes
- Process size = 72,766 bytes
  - i.e., 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes

Paging: Calculating Internal Fragmentation

- Page size = 2,048 bytes
- Best case: no internal fragmentation
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size

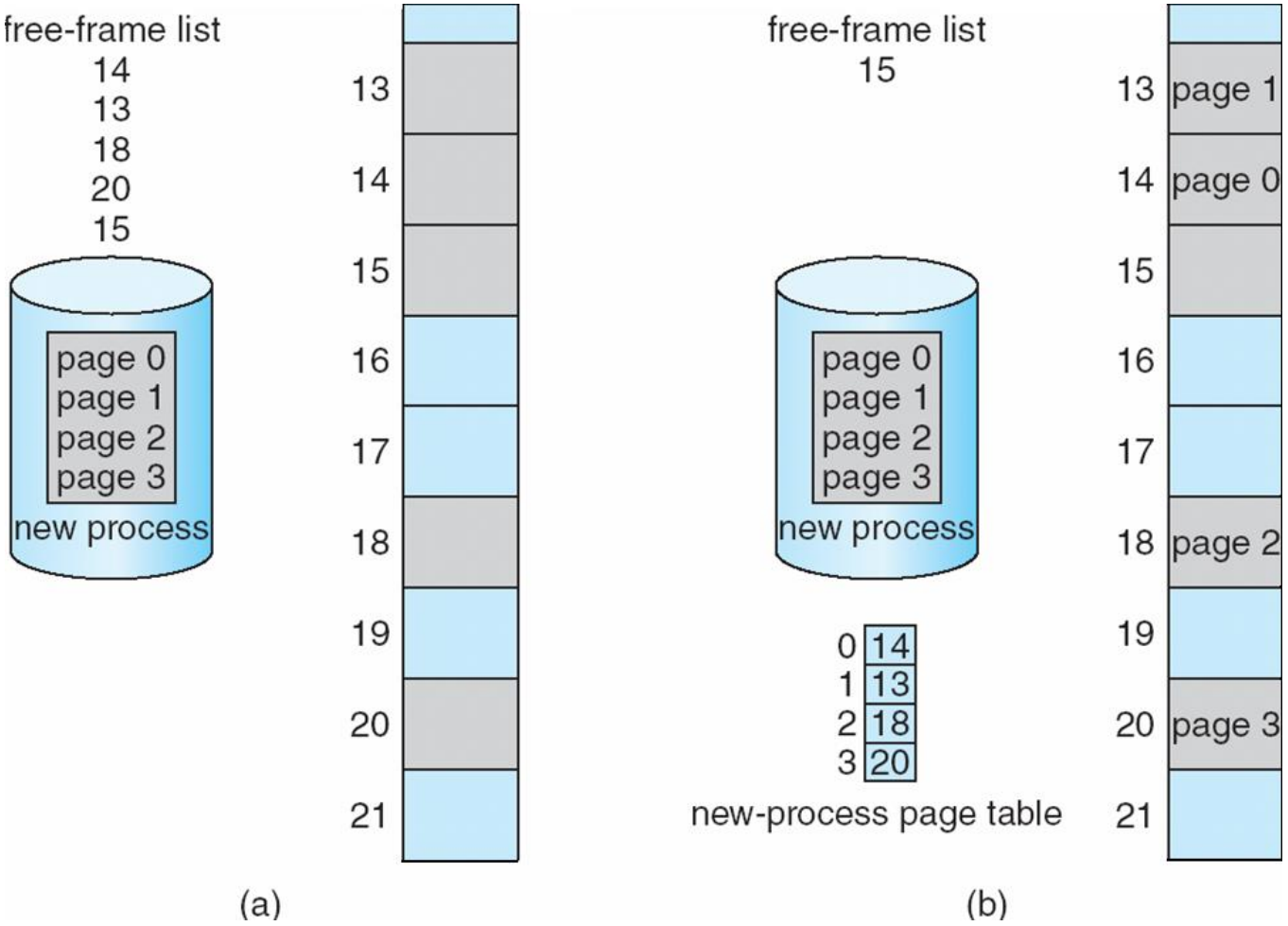
Internal Fragmentation and Frame Size

- So small frame sizes desirable?
- But each page table entry takes memory to track the mapping from a page to a frame
- Page sizes growing over time
- Some operating systems support multiple page sizes
  - Solaris supports two page sizes: 8 KB and 4 MB
  - Windows: 4KB and 2MB
  - Linux: 4KB and an architecture-dependent larger page size

Tracking Frames

- Frame table
  - One entry for each physical page frame
  - Indicate whether the frame is free or allocated and, if it is allocated, to which page of which process (or processes)

Mapping Frames



Before allocation

After allocation

Questions?

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

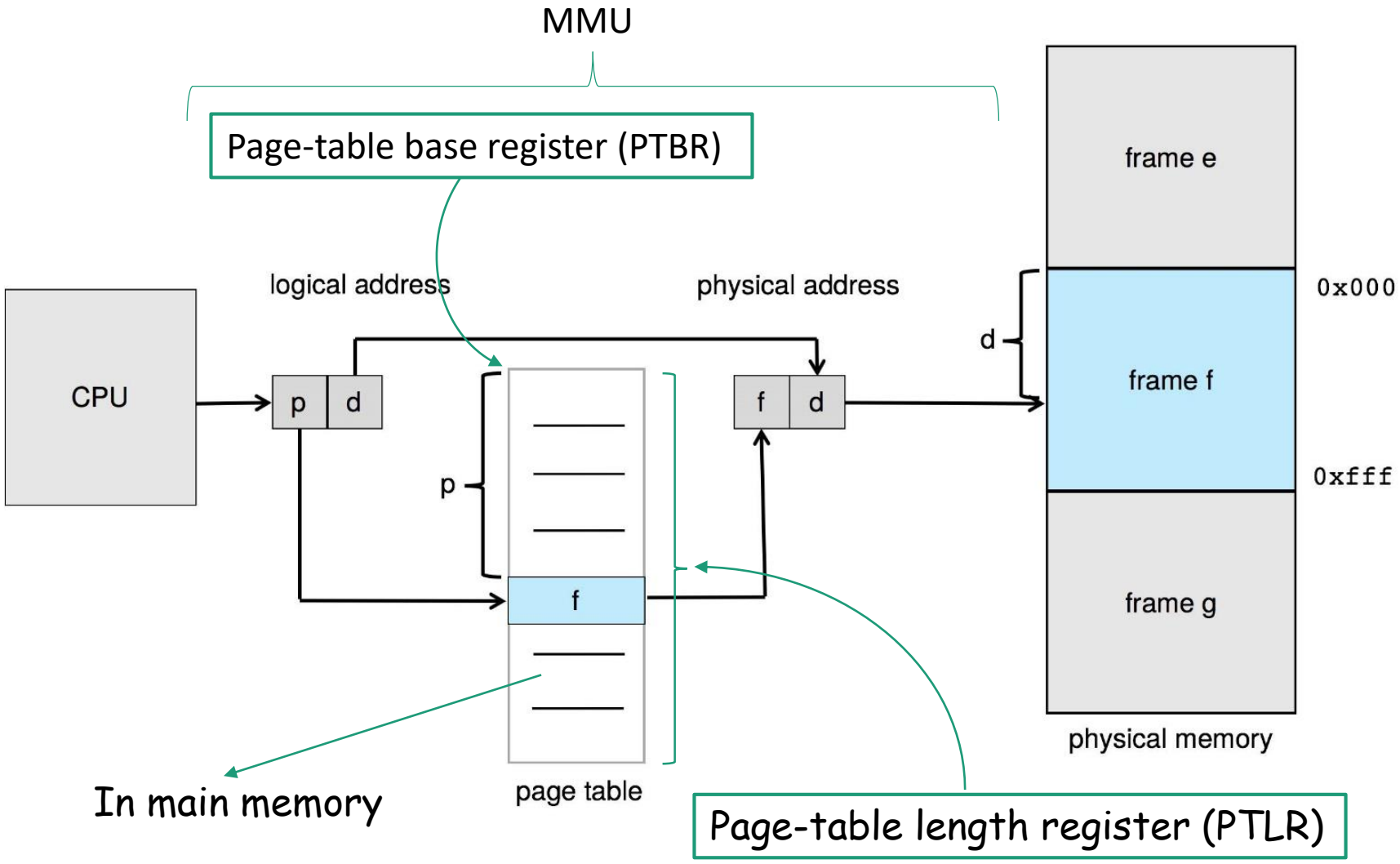
OS Page Table and MMU Page Table

- Operating systems maintain a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents.
- The CPU dispatcher defines the hardware (MMU) page table when a process is to be allocated the CPU using the OS page table.
- Paging therefore increases the context-switch time.

Page Table and Memory Access

- Hardware/MMU 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

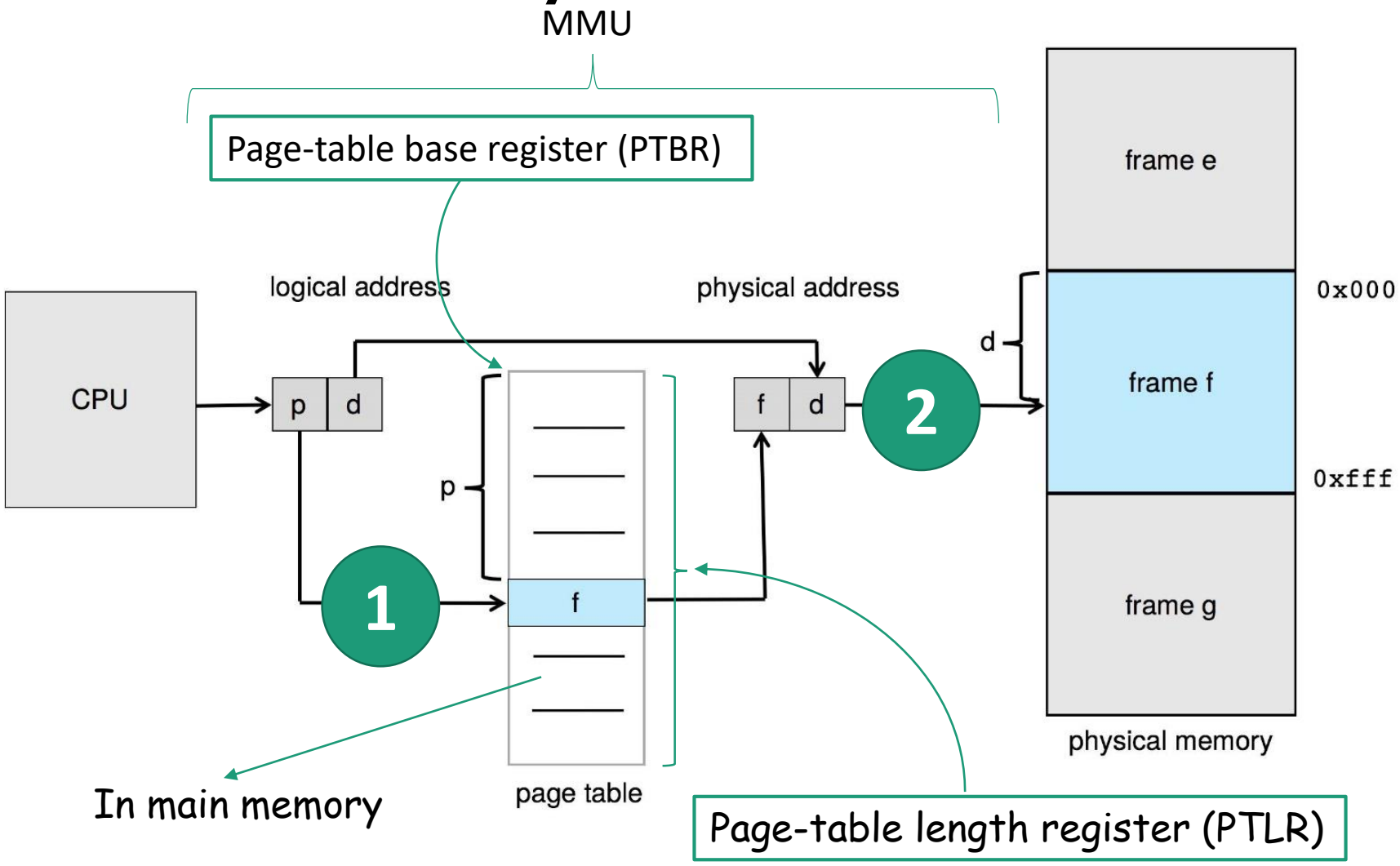
Paging Table in Main Memory



Page Table and Memory Access

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

Two Memory Accesses



Recap: Memory Access Latency

- Registers are fast while memory slow
  - Register access is done in one CPU clock (or less)
  - Main memory can take many cycles, causing a stall (memory stall)
  - e.g., mov -0x8(%rbp),%rax
- Naively implementing paging results twice memory access latencies (~50% slow down)
- Tackling memory stall:
  - Adding cache, fast memory sits between main memory and registers

Introducing TLB

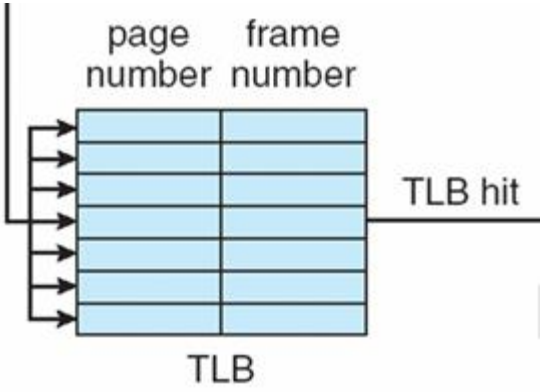
- The two-memory-access problem can be solved by the use of a special fast-lookup hardware *cache* called translation look-aside buffers (TLBs) (constructed as associative memory).

Translation Look-Aside Buffer

- 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
- 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

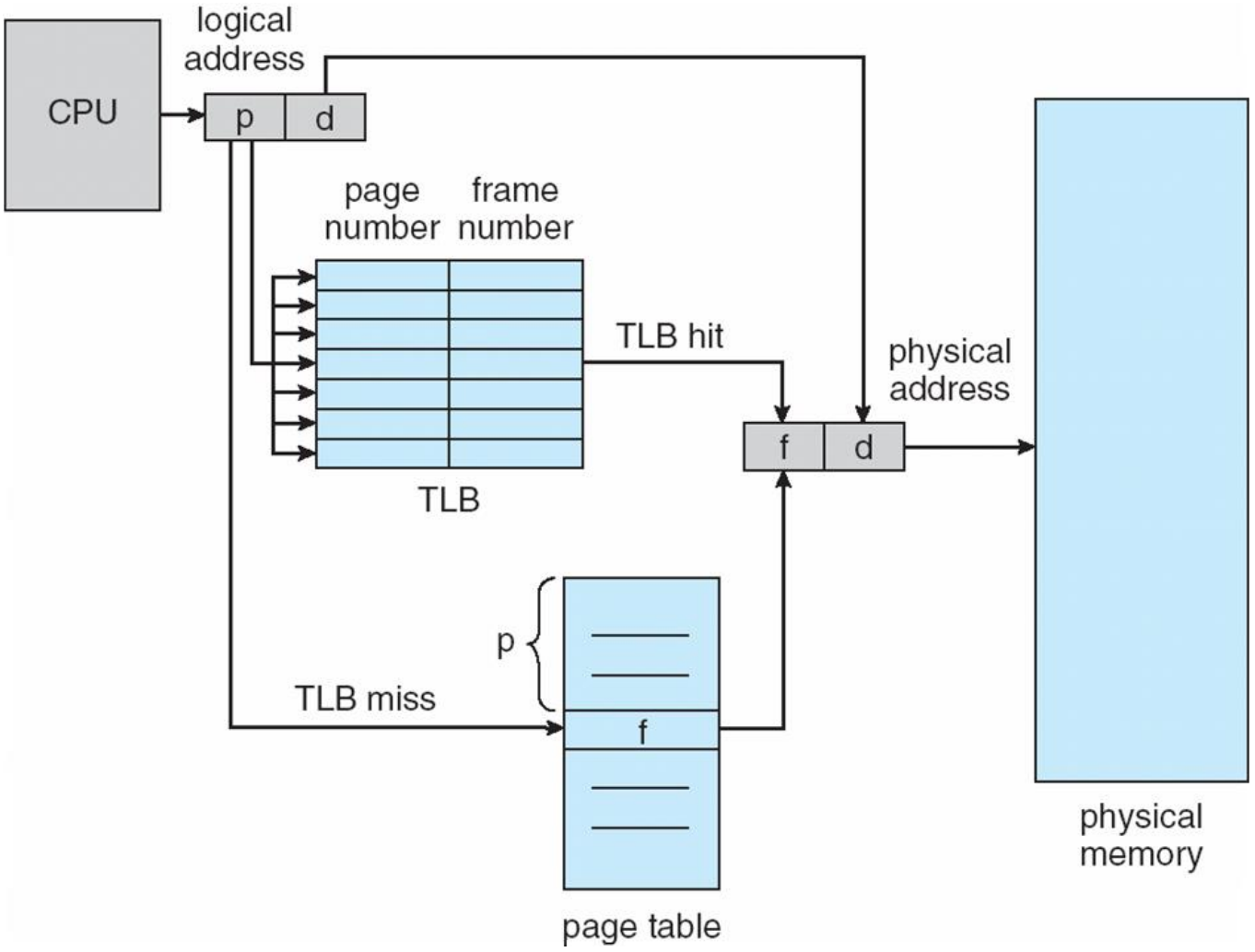
Hardware Supporting Parallel Search

- Associative memory – parallel search

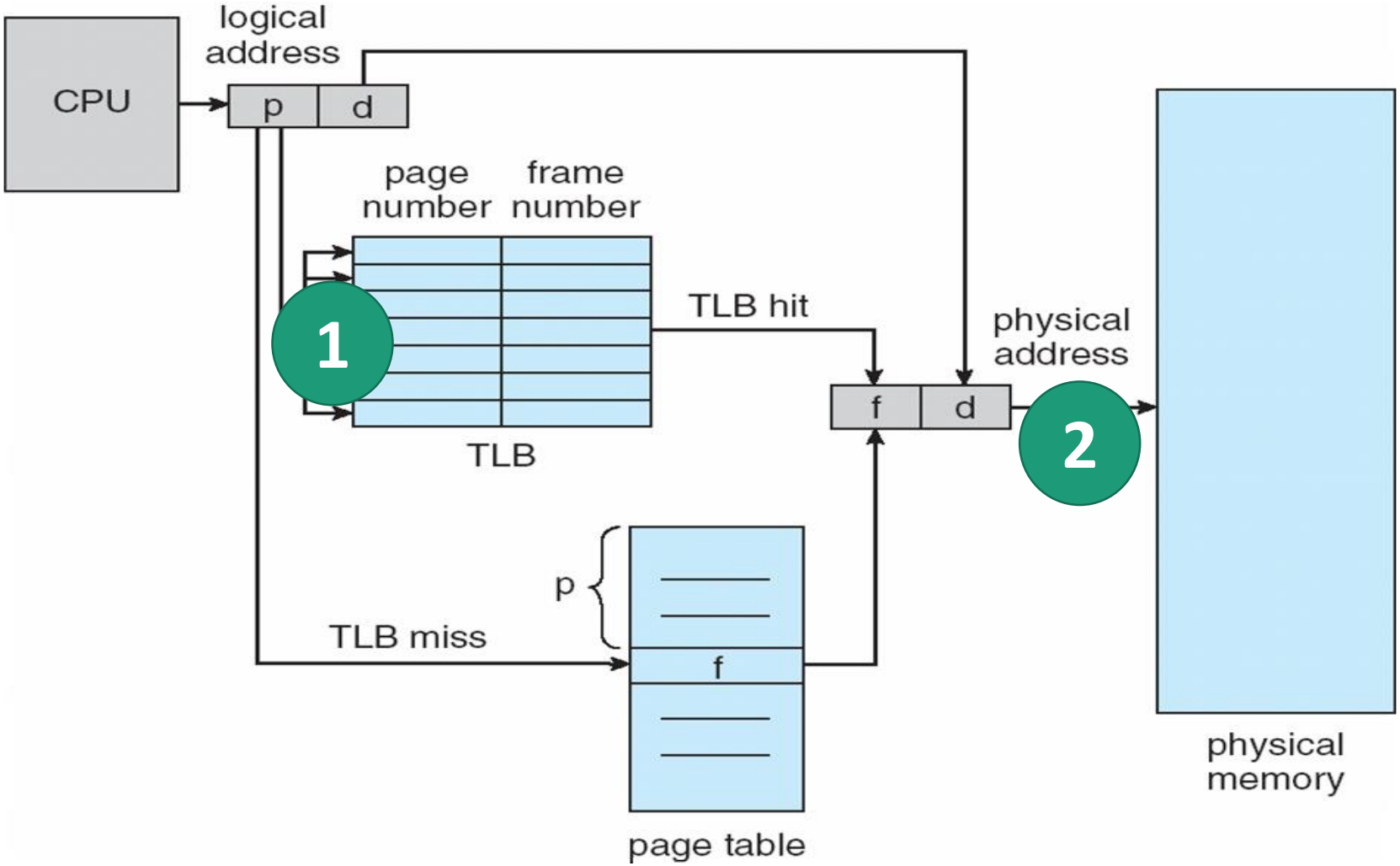


- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

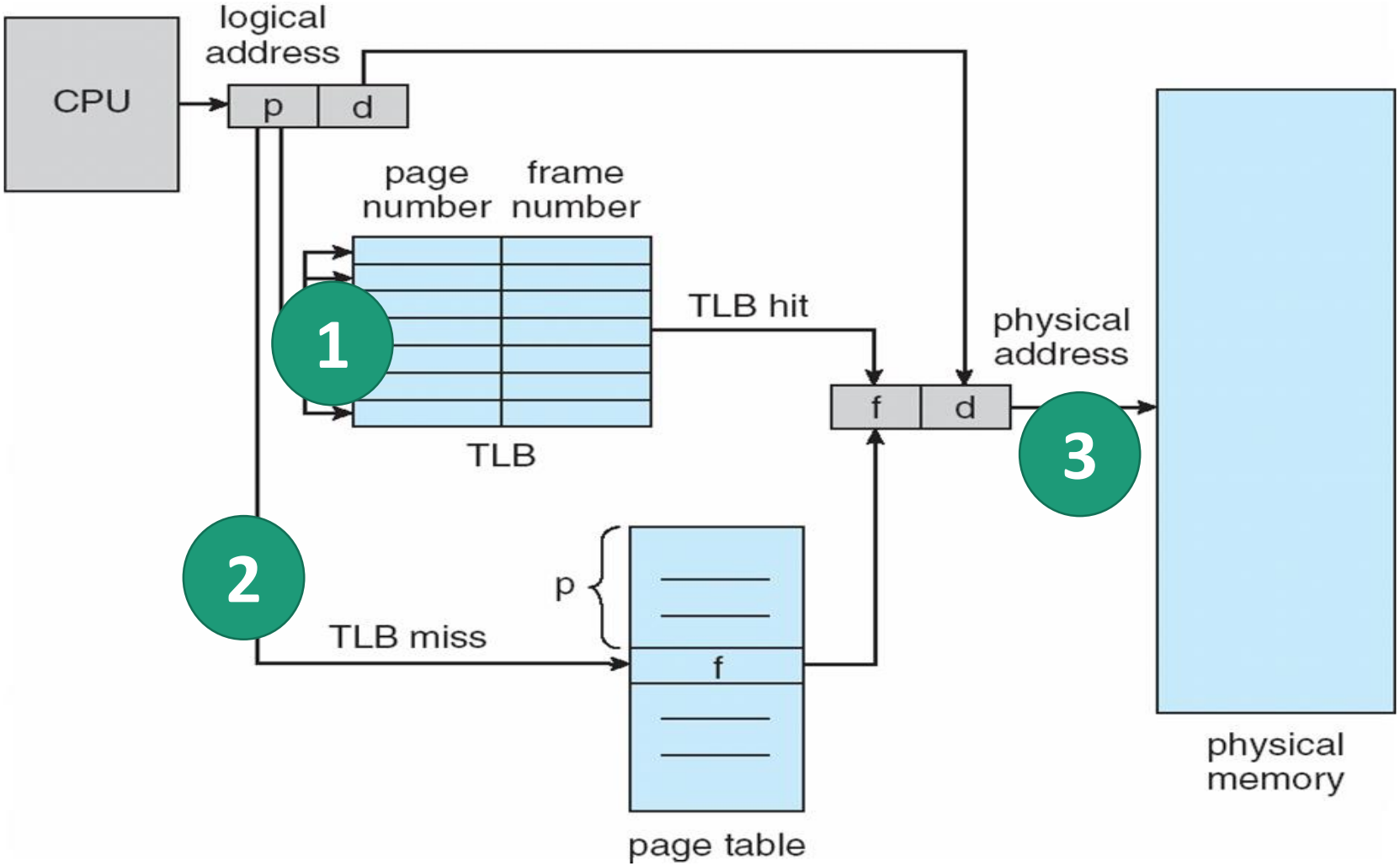
Paging Hardware With TLB



TLB Hit


CPU
logical address
p
d
page number
frame number
TLB
1
TLB hit
physical address
f
d
2
p
TLB miss
f
page table
physical memory

TLB Miss


CPU
logical address
p
d
page number
frame number
TLB hit
1
TLB
physical address
f
d
3
2
TLB miss
p
f
page table
physical memory

Hit Ratio and 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.
- Miss ratio
  - 1 – hit ratio
- The statistical or real measure of how long it takes the CPU to read or write to memory
  - It depends on hit ratio

Effective Access Time: Example

- Suppose that 10 nanoseconds to access main memory, and ignore TLB access time
- If a TLB hit, i.e., 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
  - One for page table, one for mapped-memory access
  - Ignore TLB access time
- Effective Access Time (EAT)

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

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

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

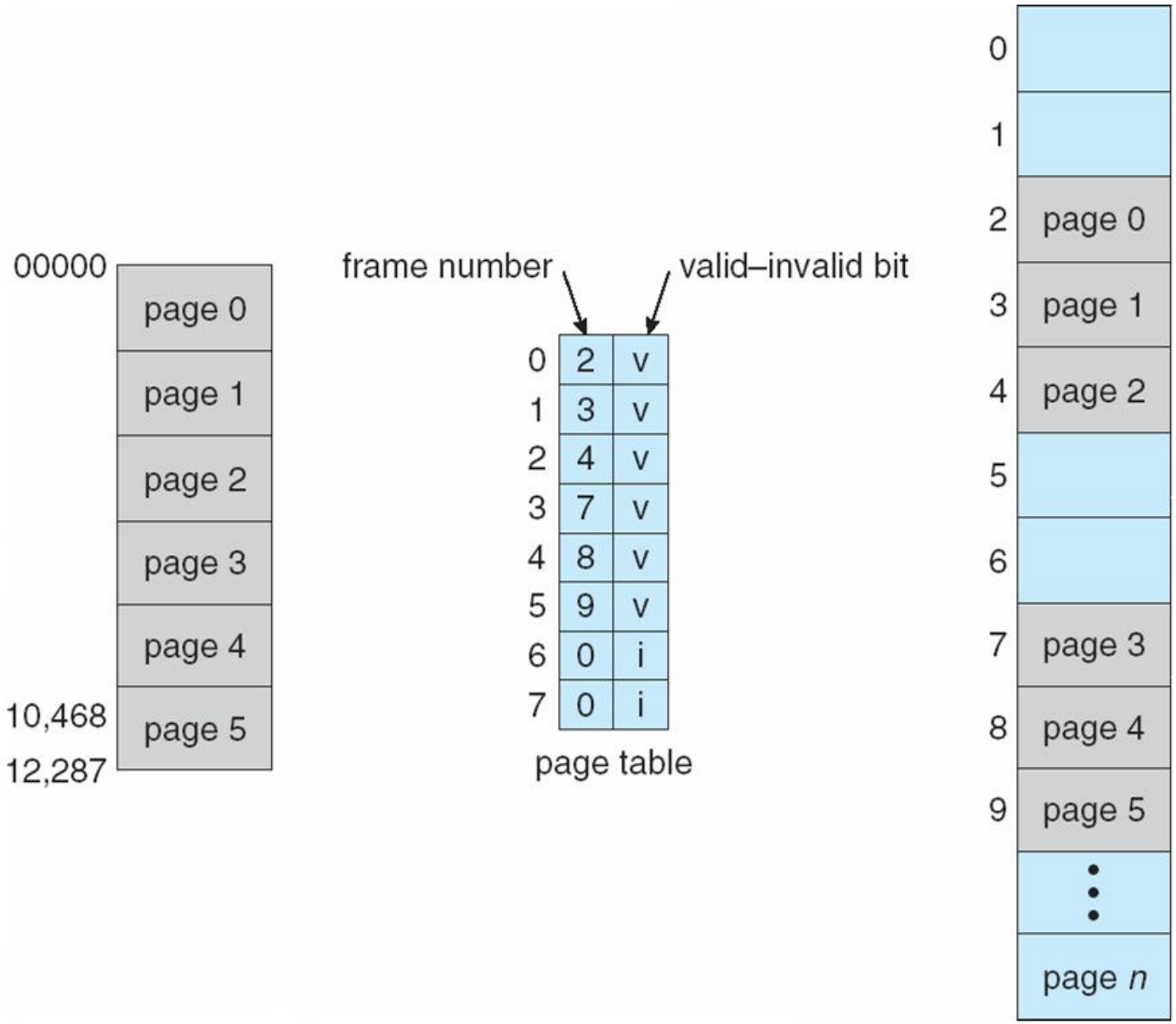
  implying only 1% slowdown in access time.

Questions?

- How to speed up paging?
- What is TLB, TLB hit, TLB miss, associative memory?
- How does TLB make paging feasible? But depends on what factors?

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
  - Use page-table length register (PTLR)
- Any violations result in a trap to the kernel

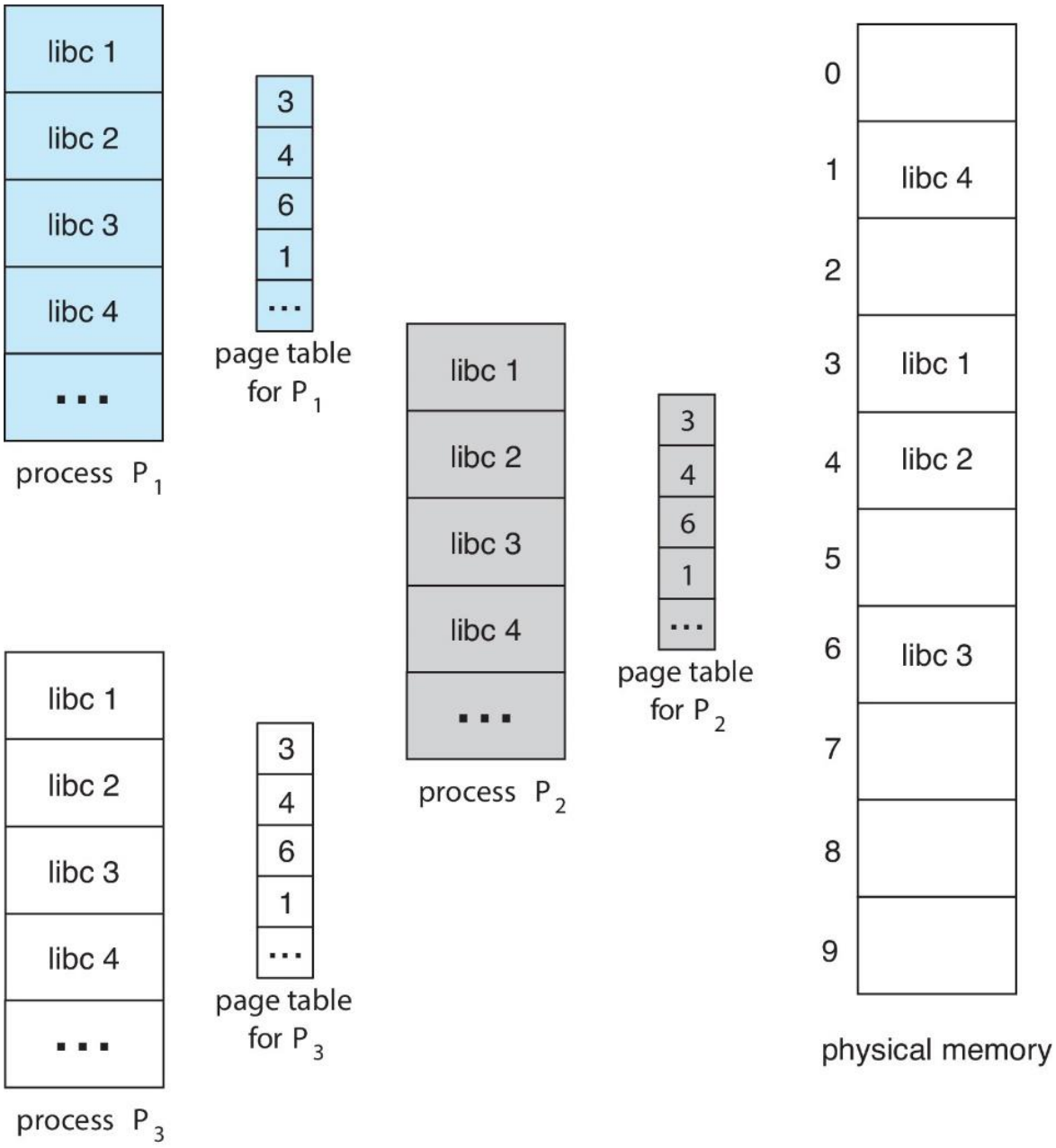

00000
page 0
page 1
page 2
page 3
page 4
page 5
10,468
12,287
frame number
valid–invalid bit
0 2 v
1 3 v
2 4 v
3 7 v
4 8 v
5 9 v
6 0 i
7 0 i
page table
0
1
2 page 0
3 page 1
4 page 2
5
6
7 page 3
8 page 4
9 page 5
page n

Questions?

- Memory protection?

Shared Pages

- Shared code, an advantage of paging
  - 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


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?