CS 370: Operating Systems
[Memory Management]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: Shrideep Pallickara

Topics covered in this lecture

- Hardware support for paging
- Memory Protection in paged environments
- Shared Pages
- Page sizes
- Structure of Page tables

All accesses to memory must go through a map.
Efficiency is important.

HARDWARE SUPPORT FOR PAGING

The purpose of the page table is to map virtual pages onto page frames

- Think of the page table as a **function**
  - Takes virtual page number as an argument
  - Produces physical frame number as result
- Virtual page field in virtual address replaced by frame field
  - Physical memory address

Two major issues facing page tables

- Can be **extremely large**
  - With a 4 KB page size, a 32-bit address space has 1 million pages
  - Also, each process has its own page table
- The **mapping must be fast**
  - Virtual-to-physical mapping must be done on *every memory reference*
  - Page table lookup should not be a bottleneck

Implementing the page table: Dedicated registers

- When a process is assigned the CPU, the dispatcher reloads these registers
- Feasible if the page table is **small**
  - However, for most contemporary systems entries are greater than 10^6

Implementing the page table in memory

- Page table base register (PTBR) points to page table
- 2 memory accesses for each access
  - One for the page-table entry
  - One for the byte

Observation

- Most programs make a *large number of references to a small number of pages*
  - Not the other way around

- Only a small fraction of the page table entries are heavily read
  - Others are barely used at all

Translation look-aside buffer (**TLB**): Small, fast-lookup hardware cache

- Number of TLB entries is small (64 ~ 1024)
  - Contains few page-table entries
- Each entry of the TLB consists of 2 parts
  - A key and a value
- When the associative memory is presented with an item
  - Item is compared with all keys *simultaneously*

Using the TLB with page tables (1)

- TLB contains only a **few** page table entries
- When a logical address is generated by the CPU, the page number is presented to the TLB
  - When frame number is found, use to access memory
  - Usually just 10-20% longer than an unmapped memory reference

Using the TLB with page tables (2)

- What if there is a TLB miss?
  - Memory reference to page table is made
  - Replacement policies for the entries

- Some TLBs allow certain entries to be **wired down**
  - TLB entries for kernel code are wired down

TLB and Address Space Identifiers (ASIDs)

- ASID uniquely **identifies** each process
  - Allows TLB to contain addresses from several different processes simultaneously

- When resolving page numbers
  - TLB ensures that ASIDs match
  - If not, it is treated as a TLB **miss**

Without ASIDs TLB must be flushed with every context switch

- Each process has its own page table

- Without flushing or ASIDs, TLB could include old entries
  - Valid virtual addresses
  - But *incorrect or invalid* physical addresses
    - From **previous** process

Effective memory access times

- 20 ns to search TLB
- 100 ns to access memory

- If page is in TLB: access time = 20 + 100 = 120 ns
- If page is not in TLB:

  20 + 100 + 100 = 220 ns

Access TLB

Access memory to retrieve frame number

Effective access times with different hit ratios

- 80%
  = 0.80 x 120 + 0.20 x 220 = 140 ns

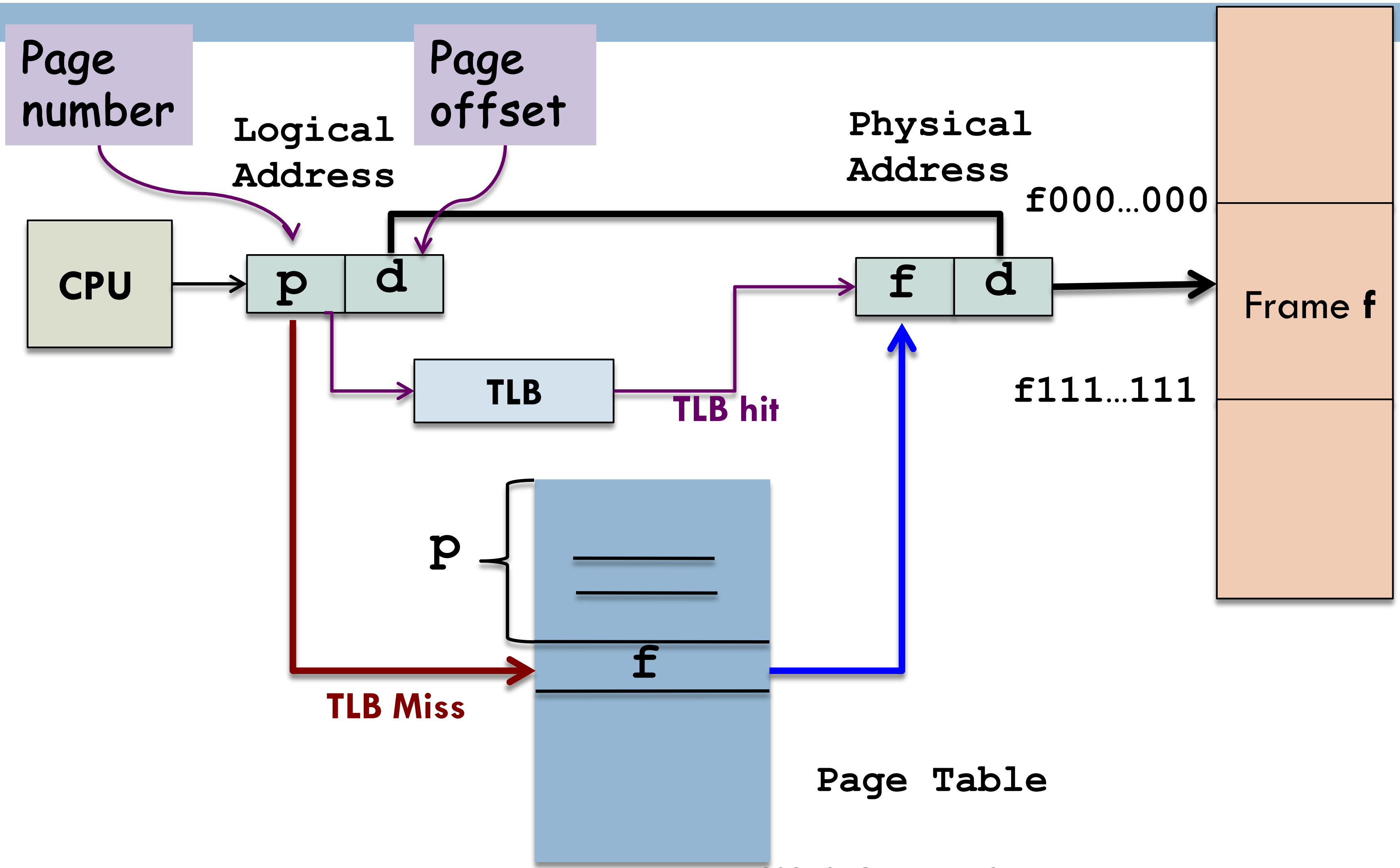
- 98%
  = 0.98 x 120 + 0.02 x 220 = 122 ns

- When hit rate increases from 80% to 98%
  - Results in a 12% reduction in access time

Process and its page table: When page table is entirely in memory

- A **pointer** to the page table is stored in the *page table base register* (PTBR) in the PCB
  - Similar to the program counter
- Often there is also a register which tracks the number of entries in the page table
- Page table need not be memory resident when the process is swapped out
  - But *must be in memory when process is running*

Paging Hardware with a TLB



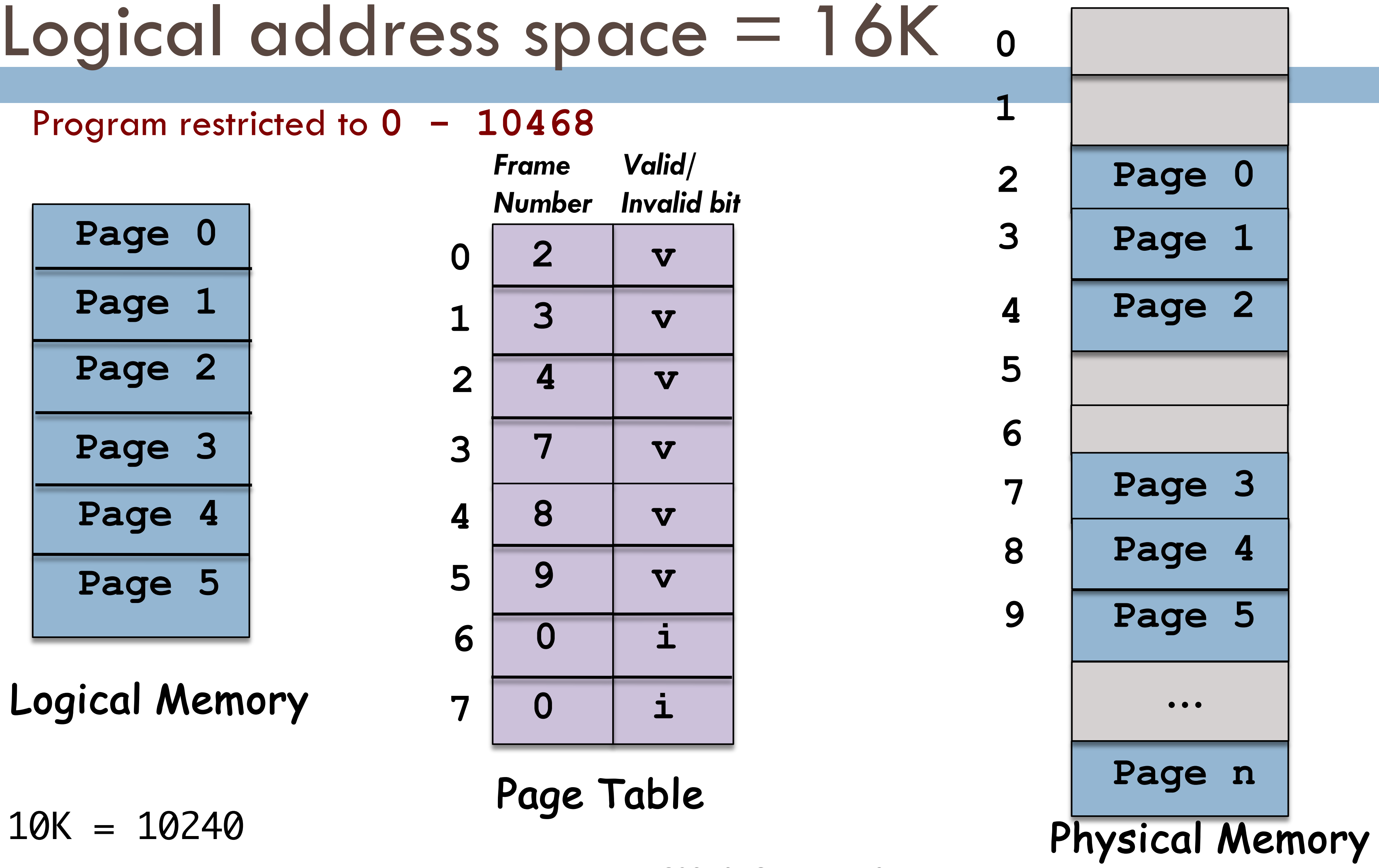
Memory Protection in Paged Environments

Protection bits are associated with each frame

- Kept in the page table
- Bits can indicate
  - Read-write, read-only, execute
  - Illegal accesses can be trapped by the OS
- Valid-invalid bit
  - Indicates if page is in the process's logical address space

Protection Bits: Page size=2K; Logical address space = 16K

Program restricted to 0 – 10468



	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

10K = 10240

Shared Pages

Reentrant Code [1/2]

- A computer program or subroutine is called **reentrant** if:
  - It can be *interrupted* in the middle of its execution and
  - Then safely called again ("re-entered") *before* its previous invocations complete execution

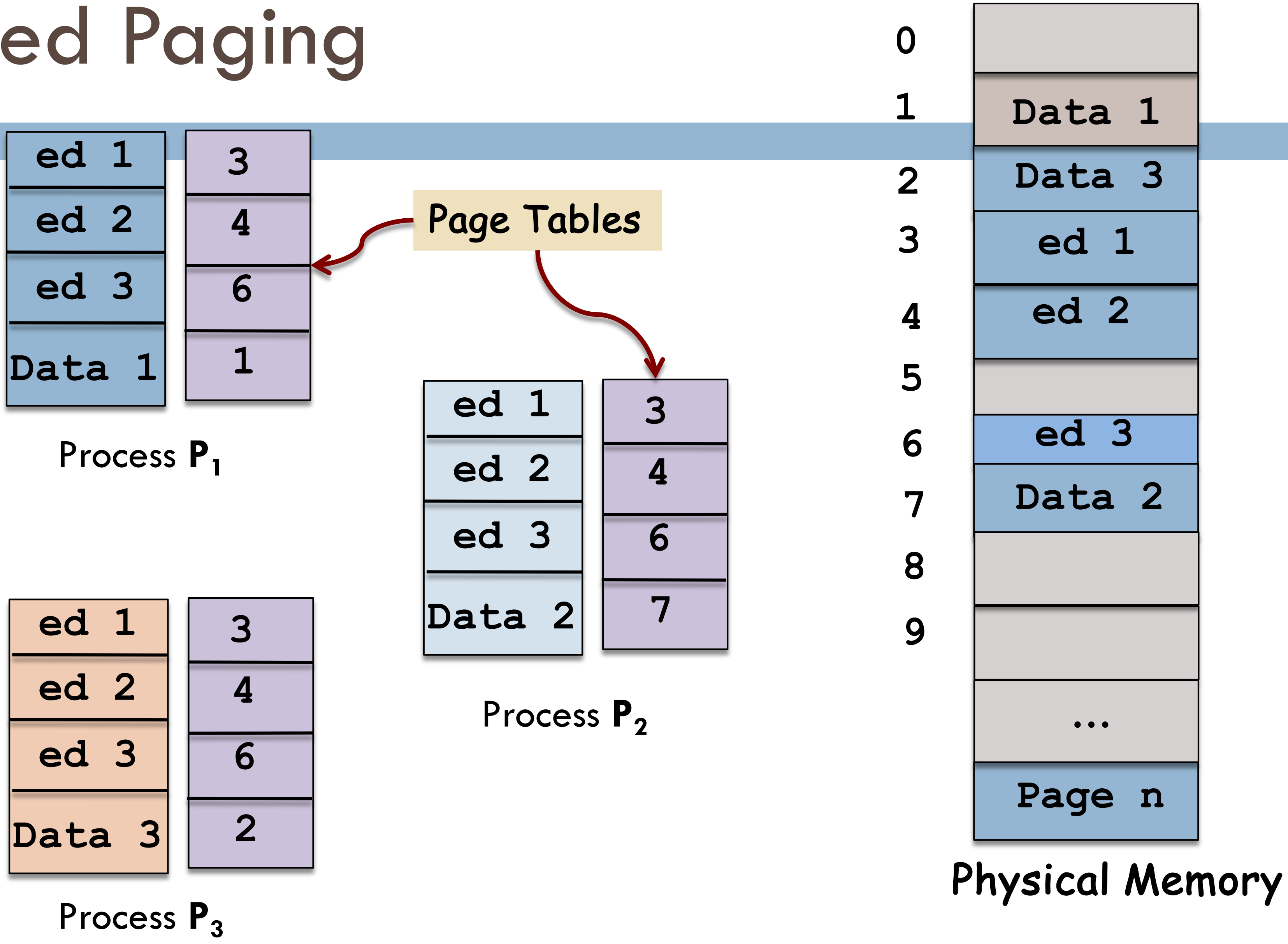
Reentrant Code [2/2]

- **Non-self-modifying**
  - Does not change during execution
- Two or more processes can:
  1. Execute same code at same time
  2. Will have different data
- Each process has:
  - Copy of registers and data storage to hold the data

Shared Pages

- System with N users
  - Each user runs a text editing program
- Text editing program
  - 150 KB of code
  - 50 KB of data space
- 40 users
  - Without sharing: 8000 KB space needed
  - With sharing : 150 + 40 x 50 = 2150 KB needed

Shared Paging



Shared Paging

- Other heavily used programs can be shared
  - Compilers, runtime libraries, database systems, etc.

- To be shareable:
  1. Code must be *reentrant*
  2. The OS *must enforce read-only* nature of the shared code

Page Sizes

Paging and page sizes

- On average, ½ of the final page is empty
  - Internal fragmentation: wasted space
- With n processes in memory, and a page size p
  - Total $np/2$ bytes of internal fragmentation
- **Greater page size = Greater fragmentation**

But having small pages is not necessarily efficient

- Small pages mean programs need more pages
  - **Larger** page tables
  - 32KB program needs
    - 4 8KB pages, but 64 512-byte pages

- **Context switches** can be *more expensive* with small pages
  - Need to reload the page table

Transfers to-and-from disk are a page at a time

- Primary Overheads: Seek and rotational delays

- Transferring a small page almost as expensive as transferring a big page
  - 64 x 15 = **960** msec to load 64 512-bytes pages
  - 4 x 25 = **100** msec to load 4 8KB pages

- Here, **large** pages make sense

Overheads in paging: Page table and internal fragmentation

- Average process size = s
- Page size = p
- Size of each page entry = e
- Pages per process = s/p
  - se/p: Total page table space

- Total Overhead = $\underbrace{se/p}_{\text{Page table overhead}} + \underbrace{p/2}_{\text{Internal fragmentation loss}}$

Looking at the overhead a little closer

- Total Overhead = $se/p + p/2$

Increases if p is small

Increases if p is large

- Optimum is somewhere *in between*
- First derivative with respect to p

$$-se/p^2 + \tfrac{1}{2} = 0 \quad \text{i.e. } p^2 = 2se$$

$$p = \sqrt{2se}$$

Optimal page size: Considering only page size and internal fragmentation

- p = sqrt($2se$)
- s = 128KB and e=8 bytes per entry
- Optimal page size = 1448 bytes
  - In practice we will never use 1448 bytes
  - Instead, either 1K or 2K would be used
    - **Why?** Pages sizes are in powers of 2 i.e. 2^X
    - Deriving offsets and page numbers is also easier

Pages sizes and size of physical memory

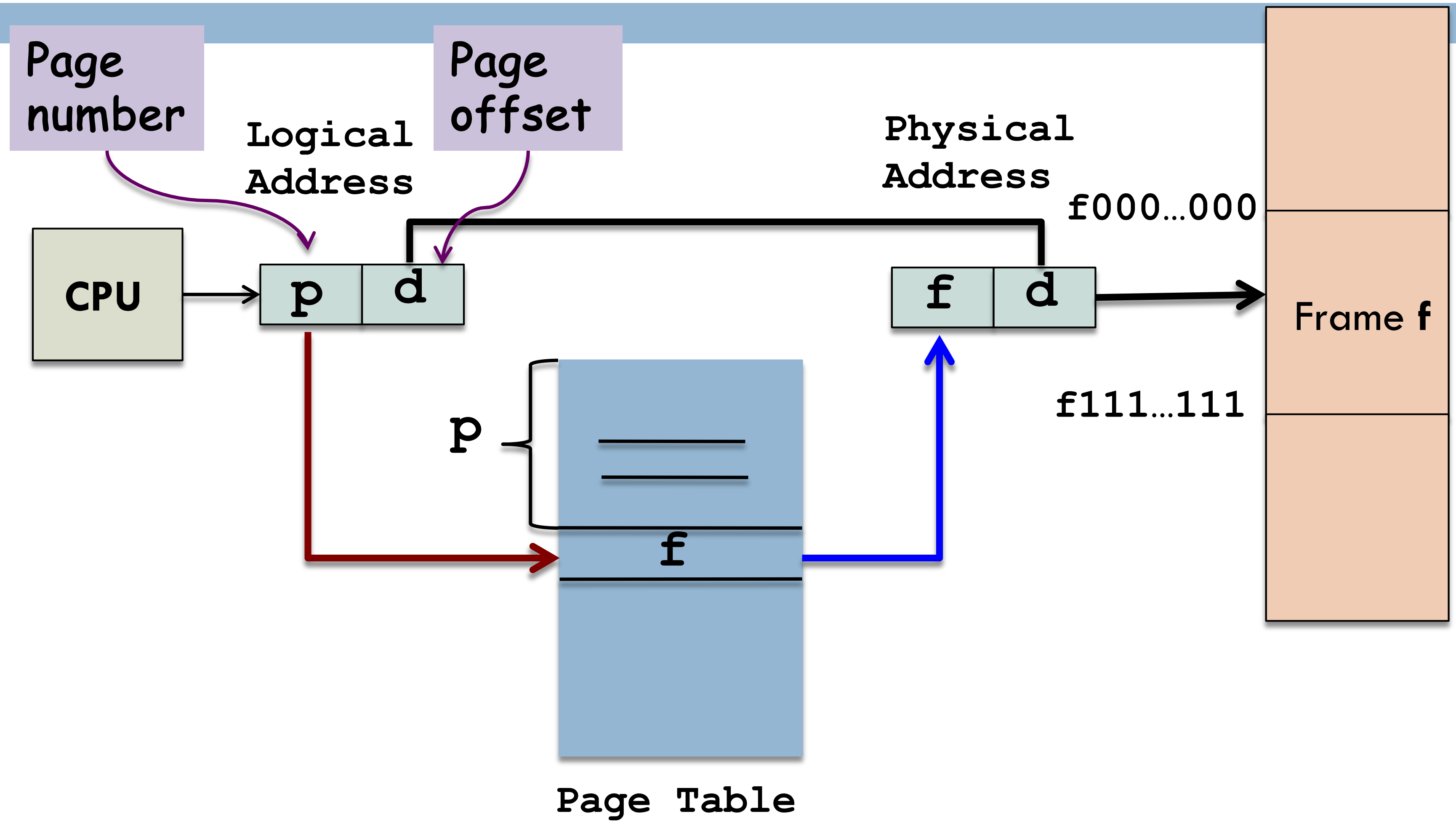
- As physical memories get bigger, page sizes get larger as well
  - Though *not linearly*
- Quadrupling physical memory size rarely even doubles page size

Structure of the Page Table

Typical use of the page table

- Process refers to addresses through pages' **virtual** address
- Process has page table
- Table has entries for pages that process uses
  - One slot for each page
    - Irrespective of whether it is valid or not
- Page table sorted by virtual addresses

Paging Hardware: Paging is a form of dynamic relocation



The contents of this slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 8]
- *Andrew S Tanenbaum. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620. [Chapter 3]*