Memory Management

Disjoint Memory Schemes

Lecture 8

Overview

- Paging Scheme
  - Basic Idea
  - Logical Address Translation
  - Hardware Support
  - Protection & Page Sharing
- Segmentation Scheme
  - Motivation
  - Basic Idea
  - Logical Address Translation
  - Hardware Support
- Segmentation with Paging Scheme

Disjoined Memory Space with Paging

- In lecture 7, memory management was discussed with two assumptions:
  1. Process occupies a contiguous physical memory region
  2. Physical memory is large enough to hold the whole process

- Let us remove assumption(1) in this lecture:
  - Process memory space can now be in **disjoint physical memory locations**
  - Via a main mechanism known as **paging**

Paging Scheme: Basic Idea

- **Physical memory** is split into regions of fixed size (decided by hardware)
  - known as **physical frames**

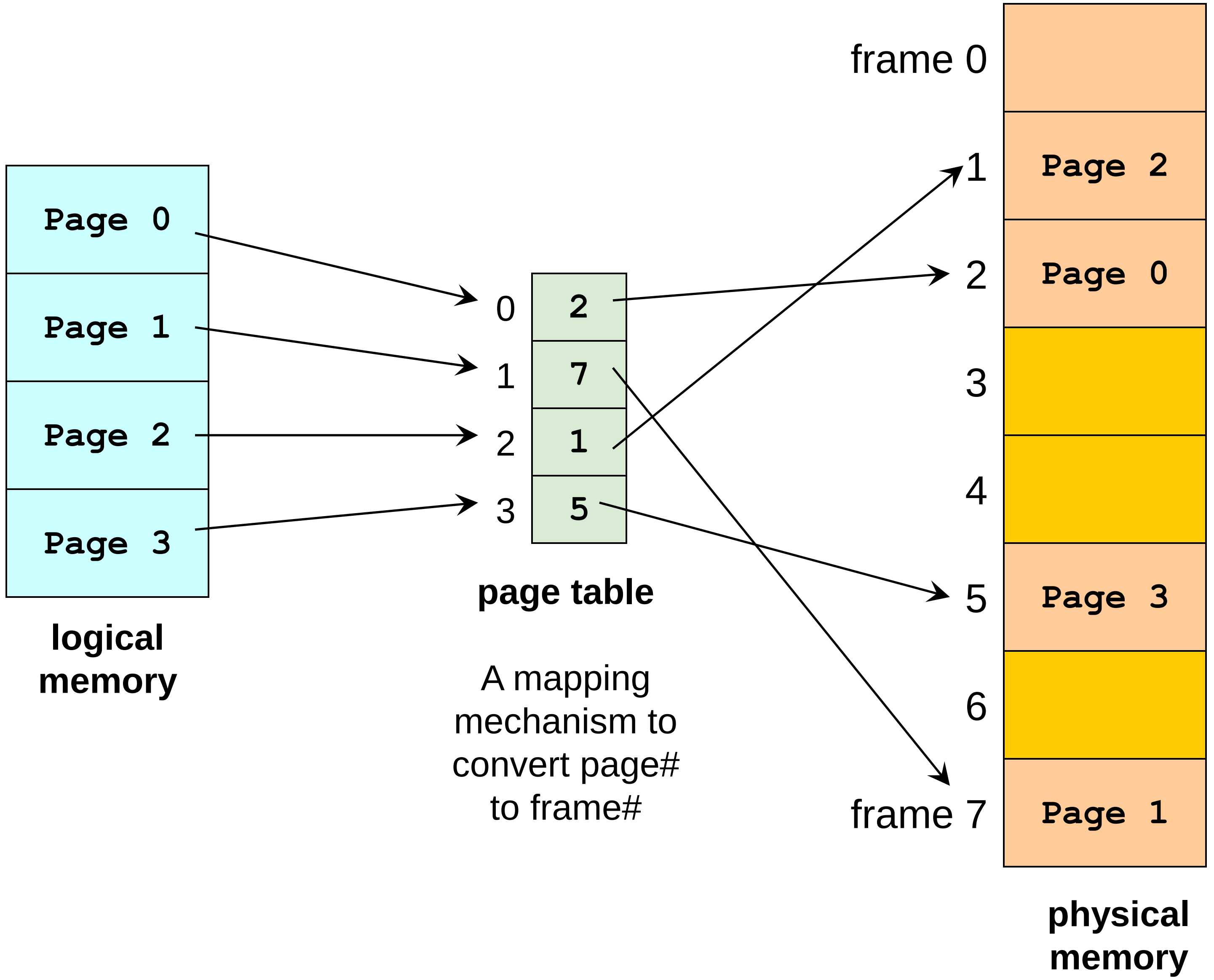
- **Logical memory** of a process is split into regions of the ***same size***
  - known as **logical pages**

- At execution time, pages of a process are loaded into **any available** memory frame
  - → Logical memory space remain contiguous
  - → Occupied physical memory region can be disjoined!

Page Table: Lookup Mechanism

- In contiguous memory allocation, it is very simple to keep track of the usage of a process:
  - Minimally, **starting address** and **size of process**
- Under paging scheme:
  - Logical page ←→ physical frame mapping is no longer straightforward
  - Need a lookup table to provide the translation
  - This structure is known as a **Page Table**

Paging: Illustration



Logical Address Translation

- Program code uses logical memory address
  - However, to actually access the value, physical memory address is needed

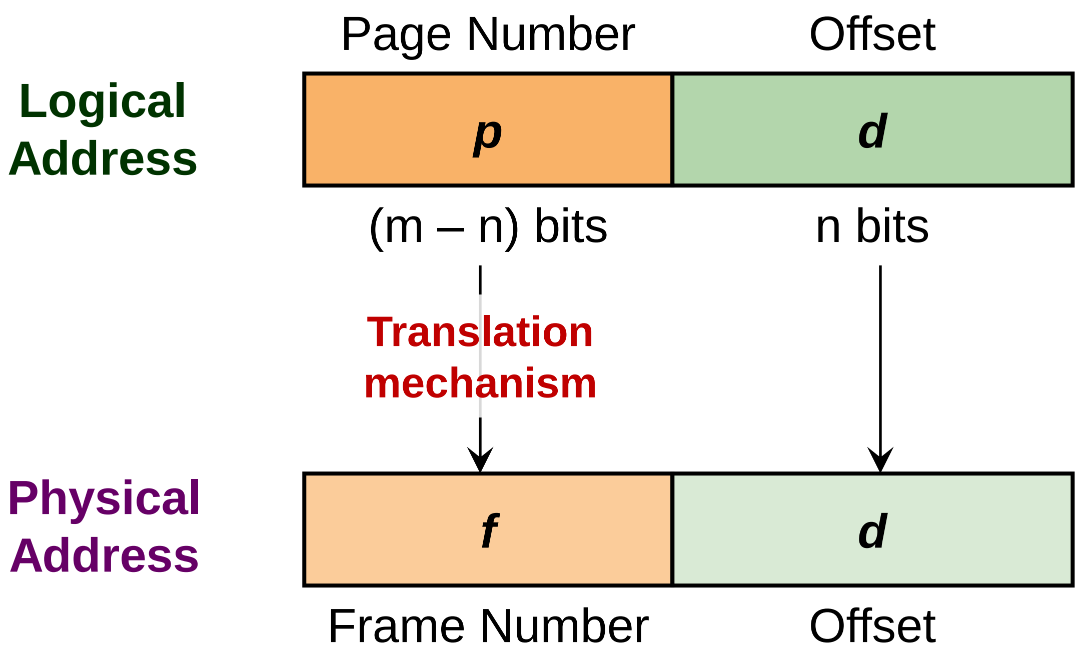
- To locate a value in physical memory in the paging scheme, we need to know:
  - **F**, the physical frame number
  - **Offset**, displacement from the beginning of the physical frame

- Physical Address

= F x size of physical frame + Offset

Logical Address Translation: **Essential Tricks**

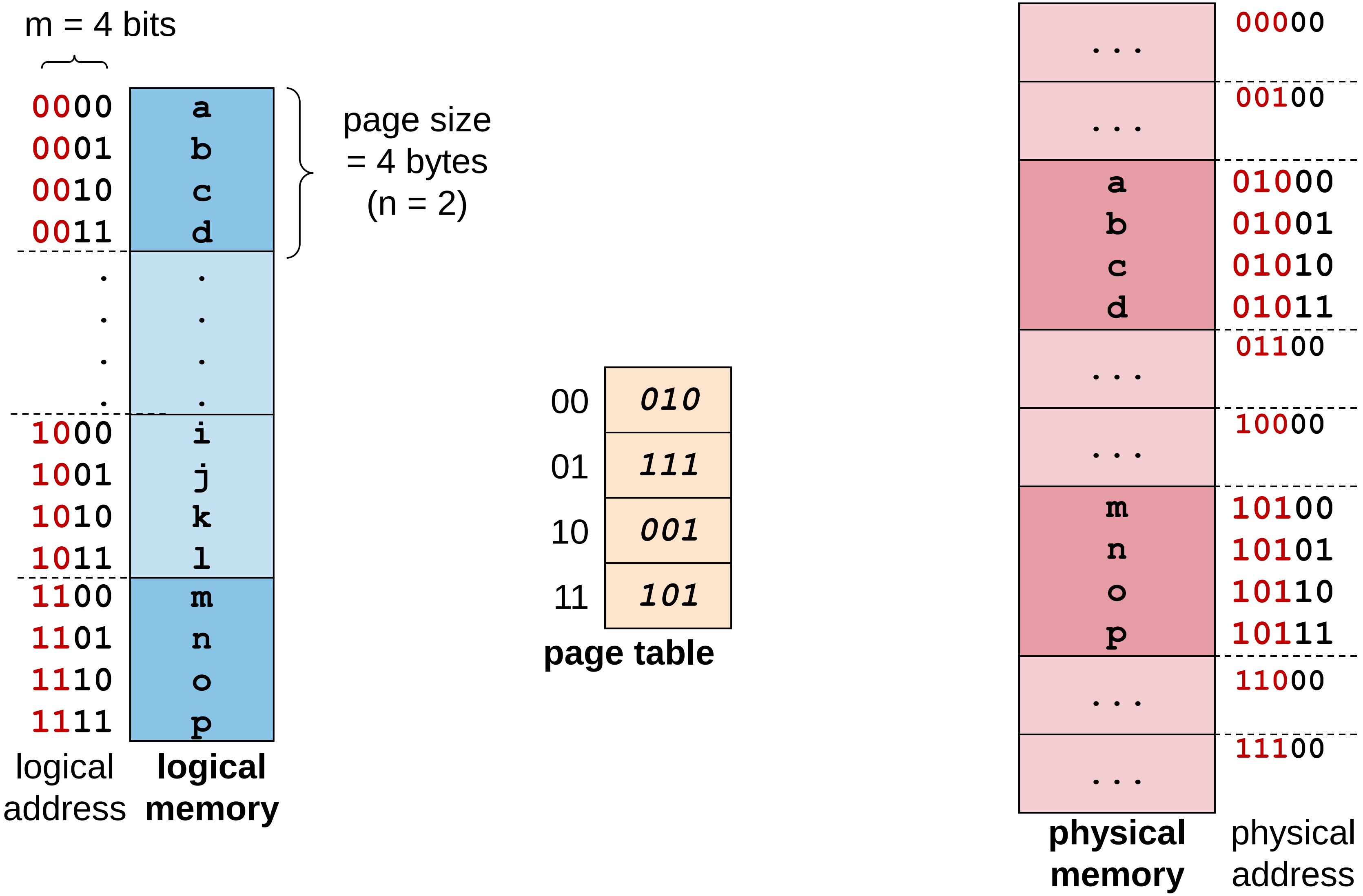
- Two important design decisions simplify address translation calculation
  1. Keep frame size (page size) as a power-of-2
  2. Physical frame size == Logical page size



Logical Address Translation: Formula

- Given:
  - Page/Frame size of 2^n
  - m bits of logical address
- Logical Address **LA**:
  - p = Most significant $m-n$ bits of **LA**
  - d = Remaining n bits of **LA**
- Use p to find frame number f
  - from mapping mechanism like page table
- Physical Address **PA**:
  - $PA = f*2^n + d$

Example: 4 Logical Pages, 8 Physical Frames



Paging: Observations

- Paging removes **external fragmentation**
  - No left-over physical memory region
  - All free frame can be used with no wastage

- Paging can still have **internal fragmentation**
  - Logical memory space may not be multiple of page size

- Clear separation of logical and physical address space
  - Allow great flexibility
  - Simple address translation

Implementing Paging Scheme

- Common pure-software solution:
  - OS stores page table alongside process information (e.g., PCB)
  - **Improved understanding:** Memory context of a process == Page table

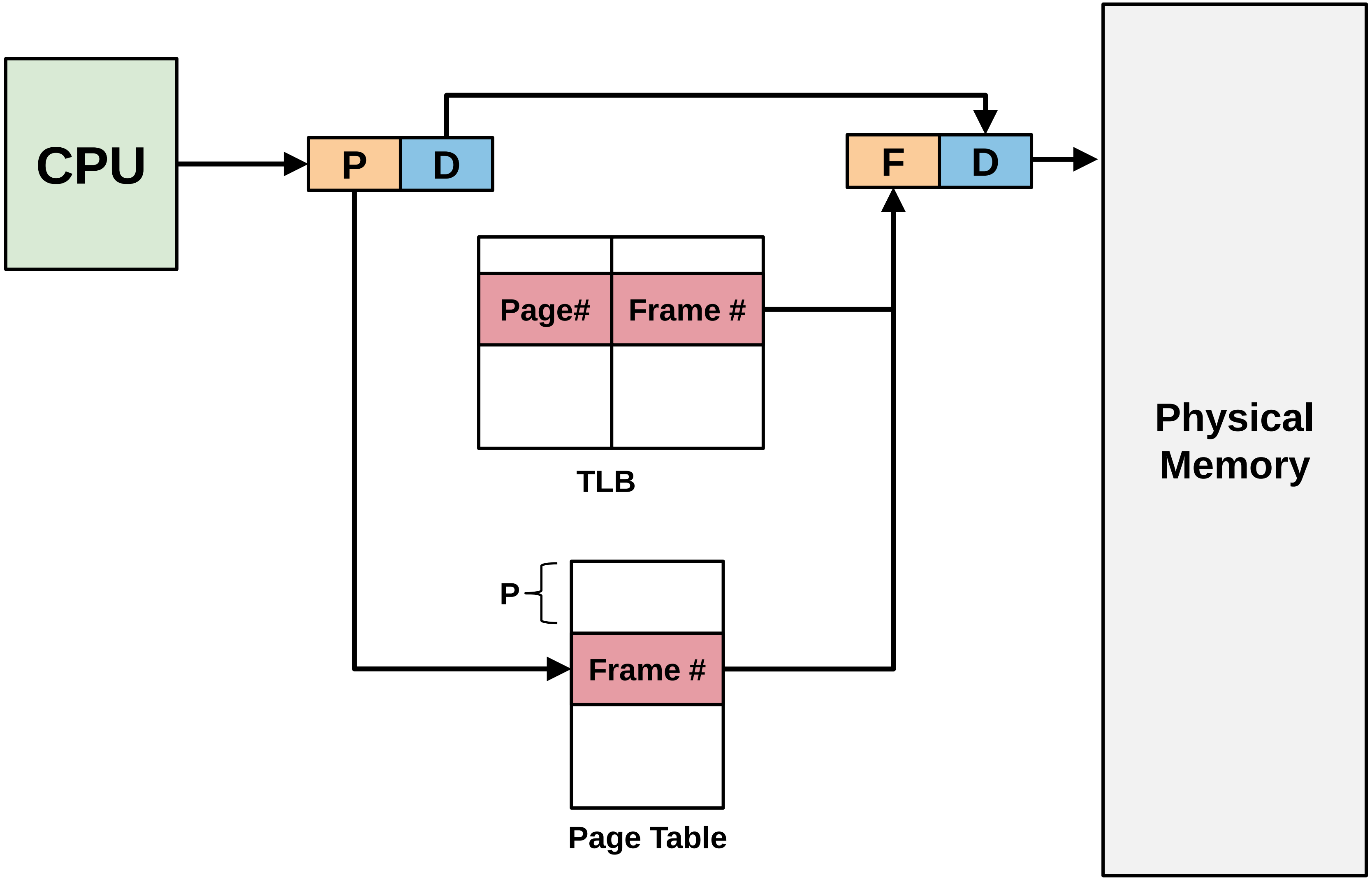
- Issues:
  - Require two memory accesses for every memory reference
    - 1st access is to read the indexed page table entry to get frame number
    - 2nd access is to access the actual memory item

Paging Scheme: **Hardware Support**

- Modern processors provide specialized on-chip component to support paging
  - Known as **T**ranslation **L**ook-Aside **B**uffer (**TLB**)
  - TLB acts as a **cache** for *a few* page table entries

- Logical address translation with TLB:
  - Use page number to search TLB associatively
  - Entry found (**TLB-Hit**):
    - Frame number is retrieved to generate physical address
  - Entry not found (**TLB-Miss**):
    - Access the full page table
    - Retrieved frame number is used to generate physical address and update TLB

Translation Look-Aside Buffer: Illustration



TLB: Impact on Memory Access Time

- Suppose:
  - TLB access takes 1ns
  - Main memory access takes 50ns
  - What is the average memory access time if TLB contains 40% of the whole page table?

- Memory access time

= TLB hit + TLB miss

= 40% x (1ns + 50ns) + 60%(1ns + 50ns + 50ns)

= 81ns

- Note:
  - Overhead of filling in TLB entry and impact of cache ignored.

TLB and Process Switching

- **Improved Understanding:** TLB is part of the **hardware context** of a process

- When a context switch occurs:
  - TLB entries are flushed
    - So that new process will not get incorrect translation

- Hence, when a process resumes running:
  - Will encounter many TLB misses to fill the TLB
    - It is possible to place some entries initially, e.g., the code pages, to reduce TLB misses

Paging Scheme: Protection

- The basic paging scheme can be easily extended to provide memory protection between processes using:
  - Access-Right Bits
  - Valid Bit
- Access Right Bits:
  - Each Page Table Entry includes several bits to indicate:
    - Whether the page is writable, readable, executable
    - E.g., page containing code should be executable, page containing data should be readable and writable, etc.
  - Memory access is checked against the access right bits

Paging Scheme: Protection (cont)

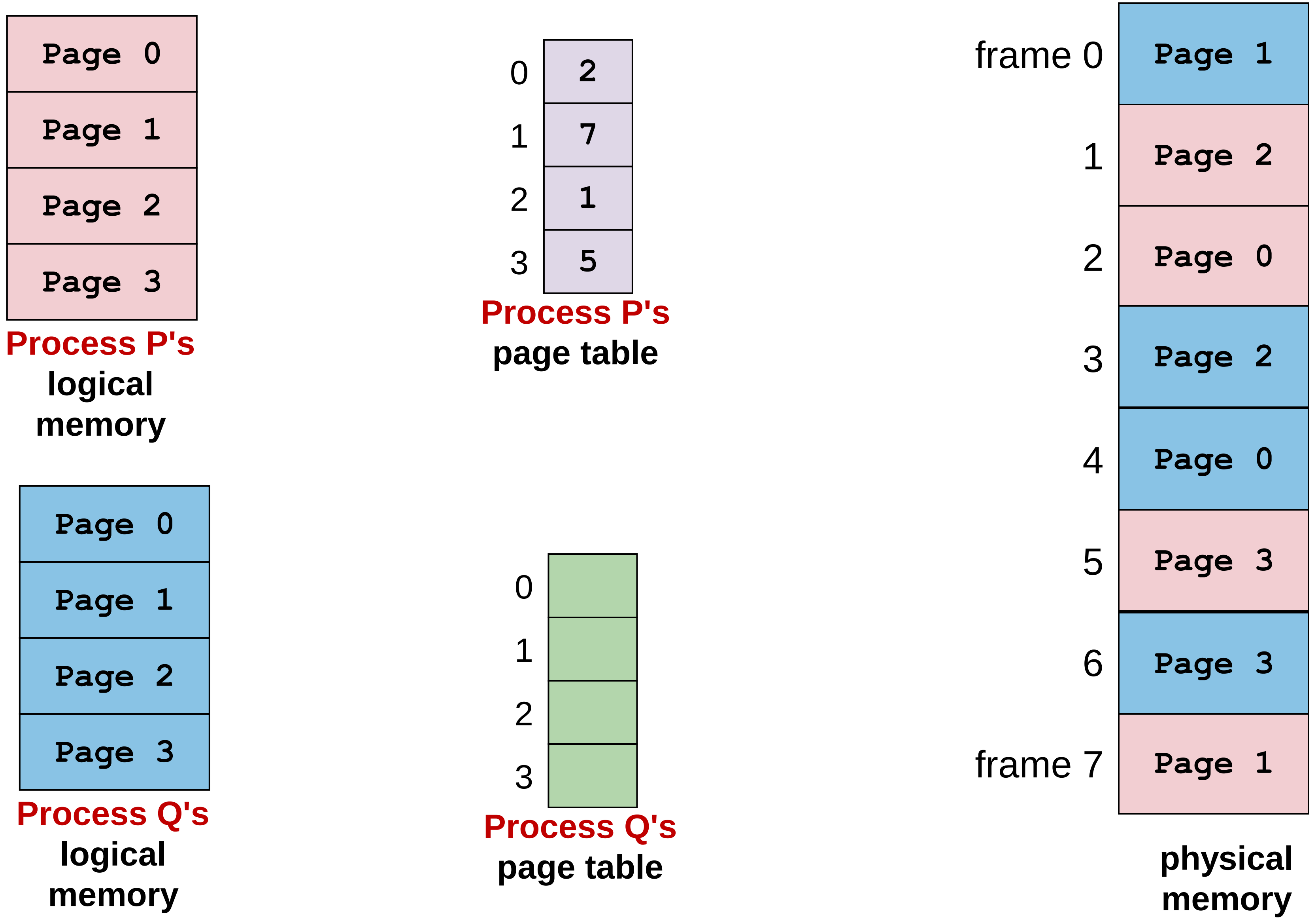
- Observation:
  - The logical memory range is usually the same for **all** processes
  - However, not all processes utilize the whole range
    - → Some pages are out-of-range for a particular process
- Valid bit:
  - Attached to each page table entry to indicate:
    - Whether the page is valid for the process to access
  - OS will set the valid bits when a process is running
  - Memory access is checked against this bit:
    - Out-of-range access will be caught by OS

Paging Scheme: Page sharing

- Page table can allow several processes to share the same physical memory frame
  - Use the same physical frame number in the page table entries

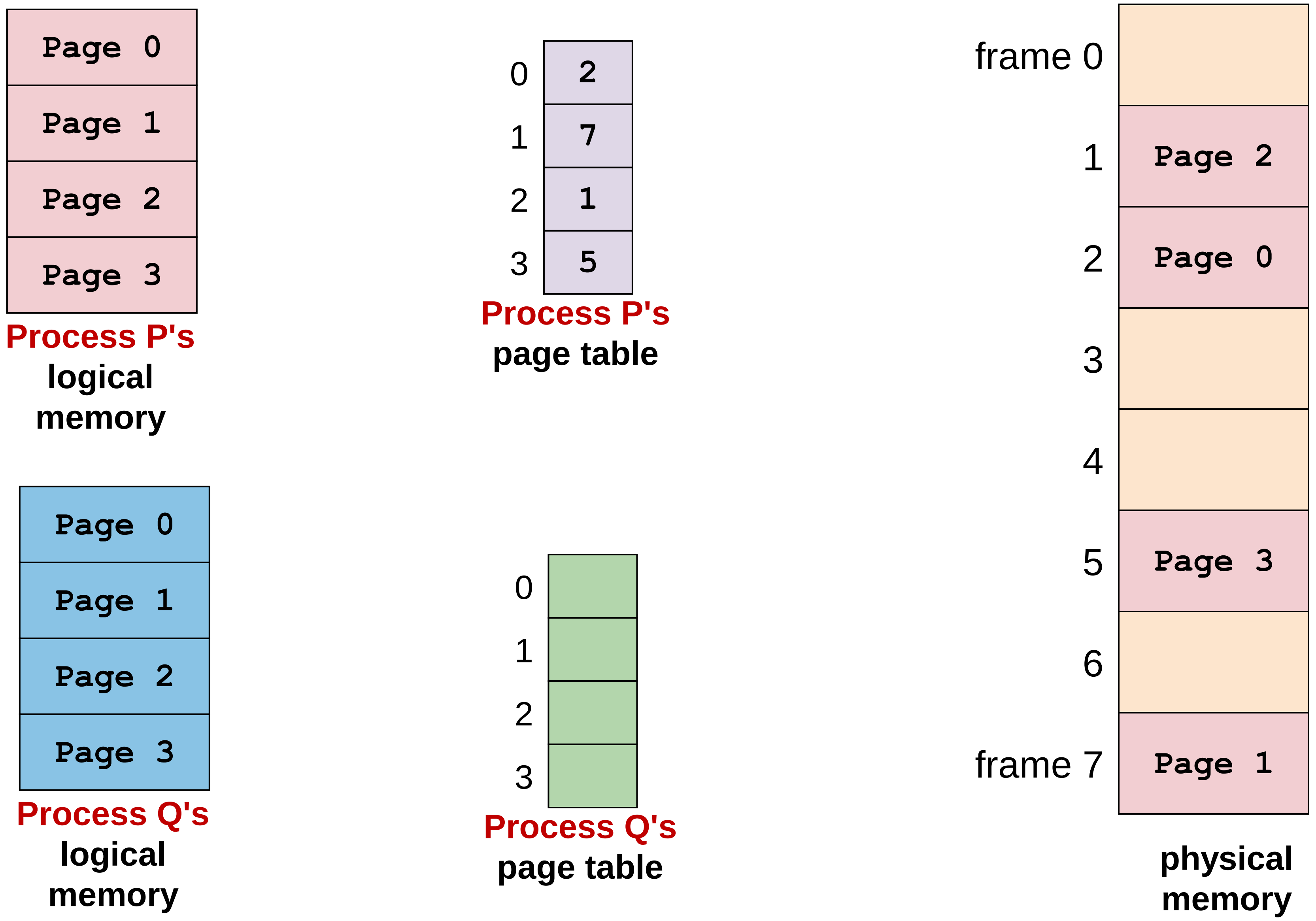
- Possible usage:
  - Shared code page:
    - Some code are being used by many processes, e.g., C standard library, system calls, etc.
  - Implement Copy-On-Write:
    - As discussed earlier in "Process Abstraction" lecture, parent and child process can share a page until one tries to change a value in it

Paging Scheme: **Page sharing**



Incomplete diagram, to be filled in during lecture.

Paging Scheme: **Copy-On-Write**



Incomplete diagram, to be filled in during lecture.

Why memory error is often called ***segmentation fault***?

SEGMENTATION SCHEME

Segmentation Scheme: **Motivation**

- Memory space of a process is treated as a single entity so far
  - However, there are several memory regions with different usage in a process
- For example, in a typical C program:
  - User Code Region
  - Global Variables Region
    - Static Data (persistent as long as program executes)
  - Heap Region:
    - Dynamic data (persistent as long as user doesn't free)
  - Stack Region
  - Library Code Region
  - etc.

Segmentation Scheme: Motivation (cont)

- Some regions may grow/shrink at execution time:
  - E.g., the stack region, the heap region, the library code region

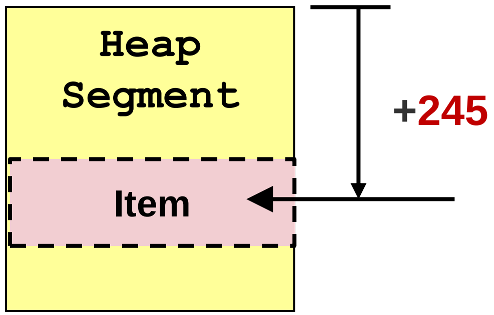
- It is hard to place different regions in a contiguous memory space and still allow them grow/shrink freely
  - Also hard to check whether a memory access in a region is within its bound

Segmentation Scheme: **Basic Idea**

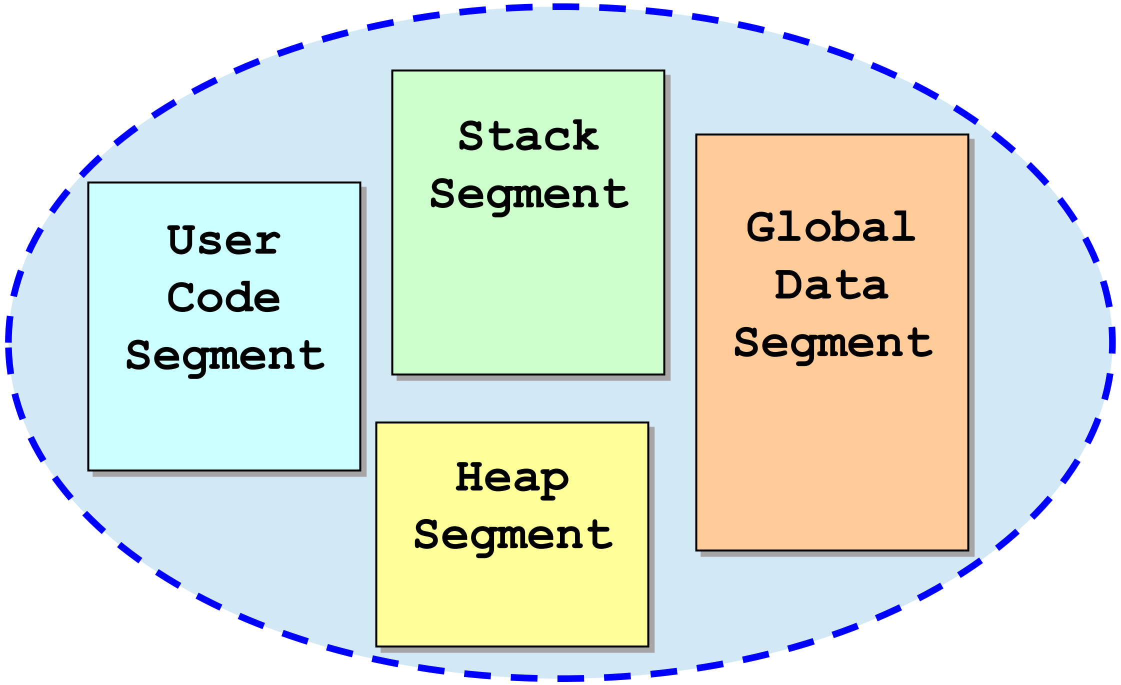
- Separate regions into multiple **memory segments**
  - Logical memory space of a process is now a collection of **segments**
- Each memory segment:
  - Has a name
    - For ease of reference
  - Has a limit
    - Indicate the range of the segment
- All memory reference is now specified as:
  - **Segment name + Offset**

Memory Access Example:

"*Heap*" + 245

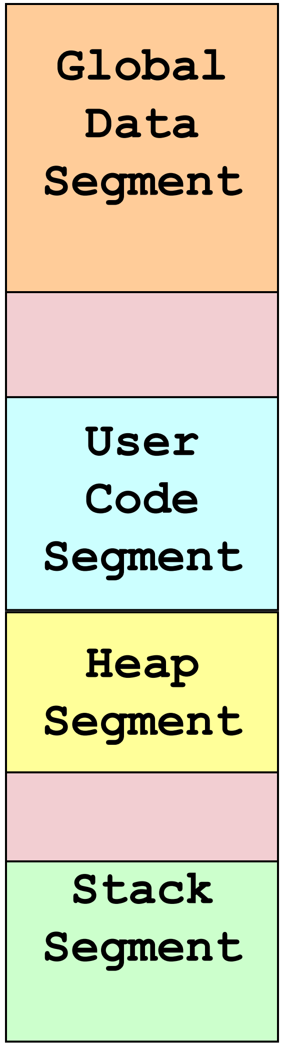


Segmentation: **Illustration**



Logical Address Space

(Segments can be of different sizes)



Physical Memory

(Segments are disjointed in RAM)

Segmentation: Logical Address Translation

- Each segment is mapped to a contiguous physical memory region
  - with a ***base address*** and a ***limit***
- The segmentation name is usually represented as a single number
  - Known as ***segment id***
- Logical address `<SegID, Offset>`:
  - `SegID` is used to look up `<Base, Limit>` of the segment in a **segment table**
  - Physical Address `PA = Base + Offset`
  - `Offset < Limit` for valid access

Logical Address Translation: Illustration

Assume:

```
User Code Segment   = 0
Global Data Segment = 1
Heap Segment        = 2
Stack Segment       = 3
```

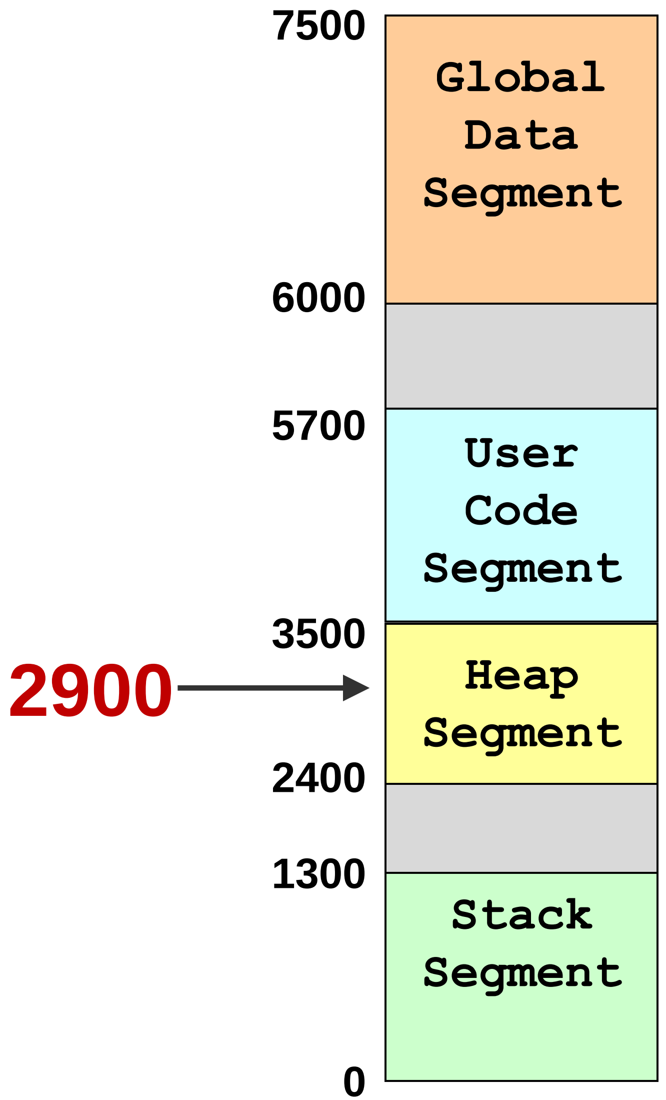
Memory Access

< Segment Id, Offset >

	Base	Limit
0	3500	2200
1	6000	1500
2	2400	1100
3	0	1300

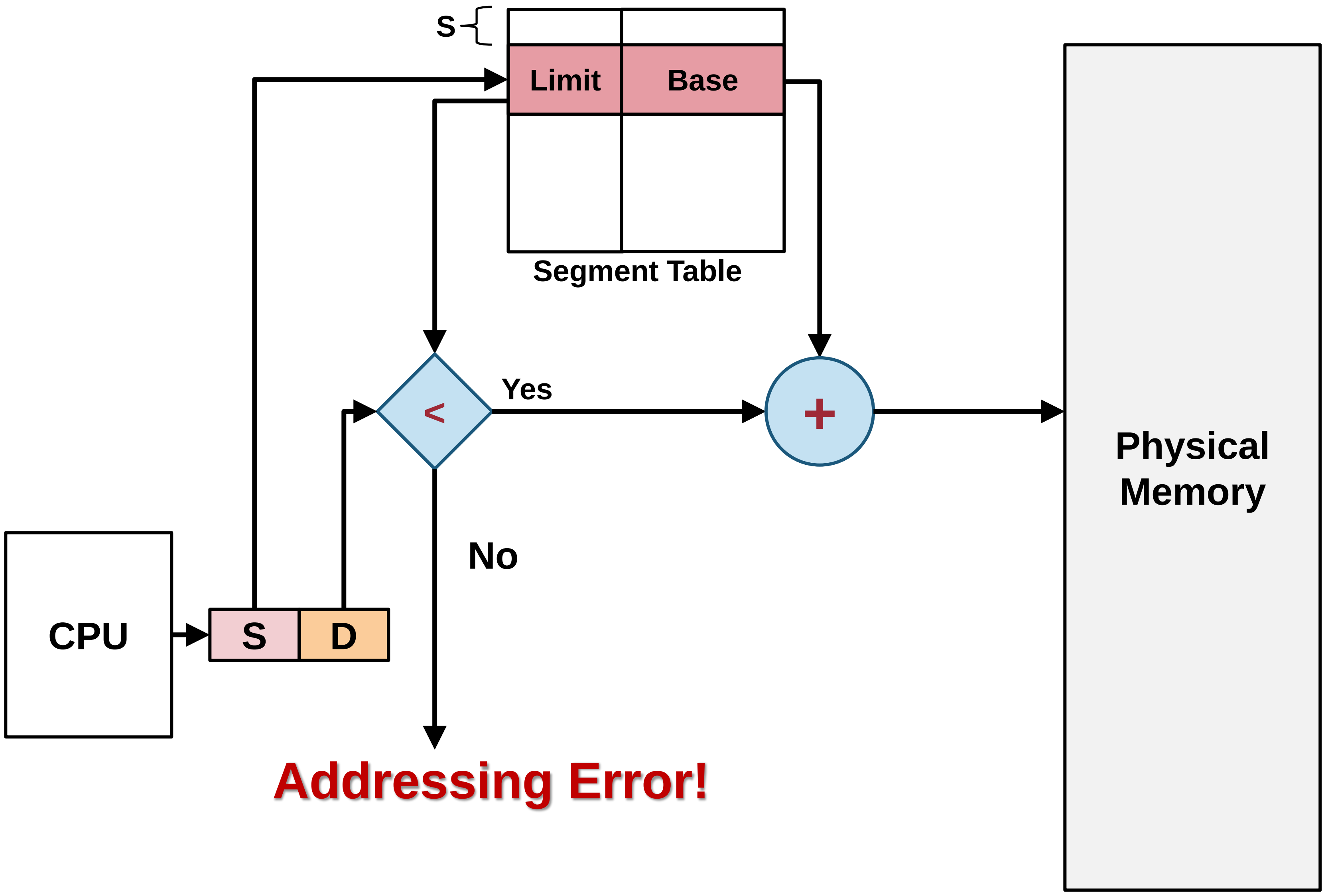
segment table

< 2, 500 >



Physical Memory

Segmentation: **Hardware Support**



Segmentation: Summary

- **Pros:**
  - Each segment is an independent contiguous memory space
  - ➔ Can grow / shrink independently
  - ➔ Can be protected / shared independently

- **Cons:**
  - Segmentation requires variable-size contiguous memory regions
  - ➔ can cause **external fragmentation**

- Important observation:
  - Segmentation is **not** the same as paging
  - Each of them trying to solve a different problem

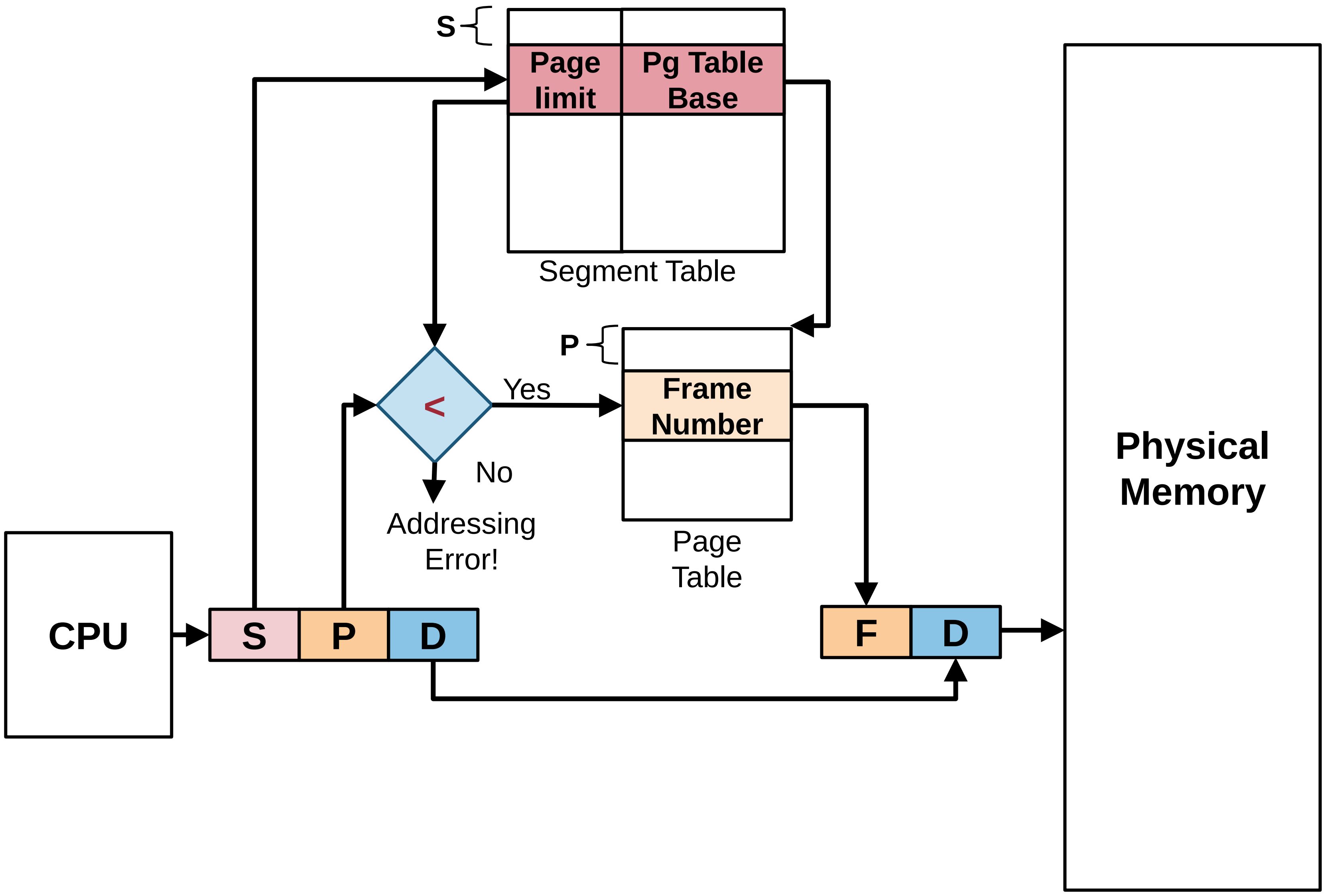
P is good; S is also good; Let's do S+P then!

SEGMENTATION WITH PAGING

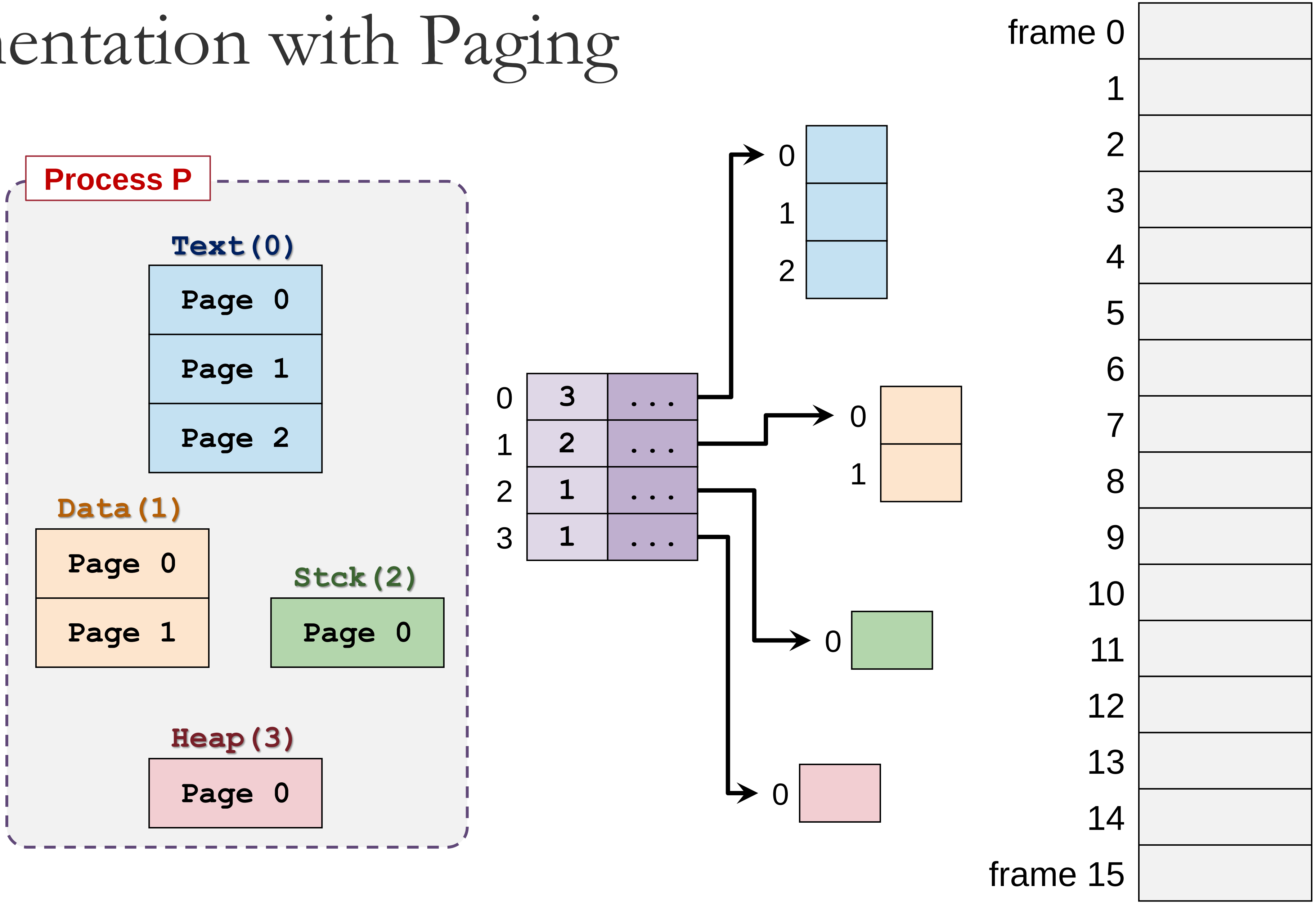
Segmentation with Paging: Basic Idea

- The intuitive next step is to combine segmentation and paging
- Basic Idea:
  - Each segment is now composed of several pages instead of a contiguous memory region
  - ➔ Essentially, each segment has a page table
  - Segment can grow by allocating new pages then add to its page table, and similarly for shrinking

Segmentation with Paging: Illustration



Segmentation with Paging



Incomplete diagram, to be filled in during lecture.

Summary

- Discussed two popular memory management schemes
  - Both allow the logical address space to be in disjoint physical regions
- Paging split the logical address into fixed size pages
  - Store in fixed size physical memory frames
- Segmentation split the logical address into variable size segments according to their usage
  - Stored in variable-sized physical partitions
- Combining segmentation and paging