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

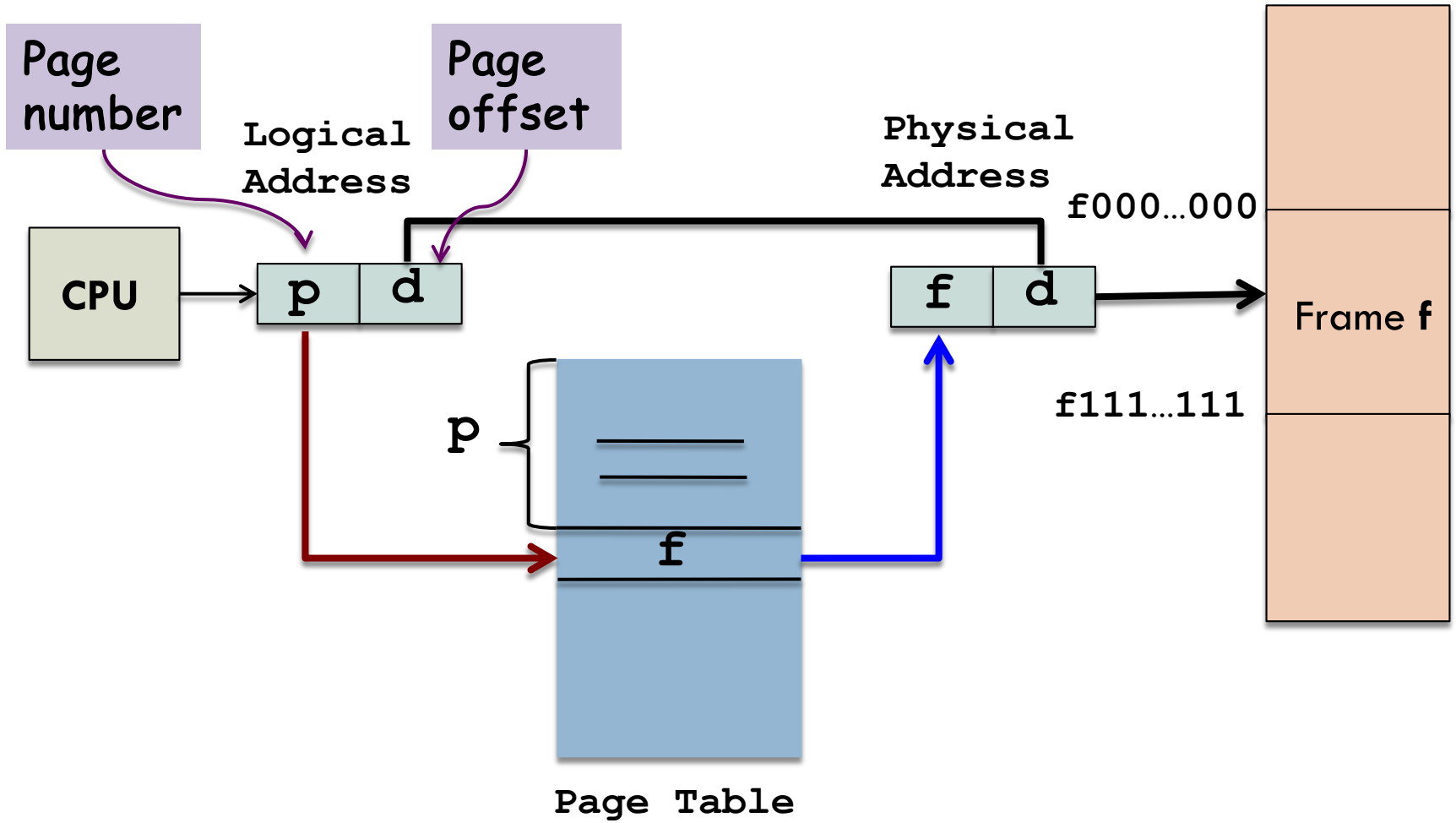
- Structure of Page tables
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables
- Segmentation

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



Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Paging

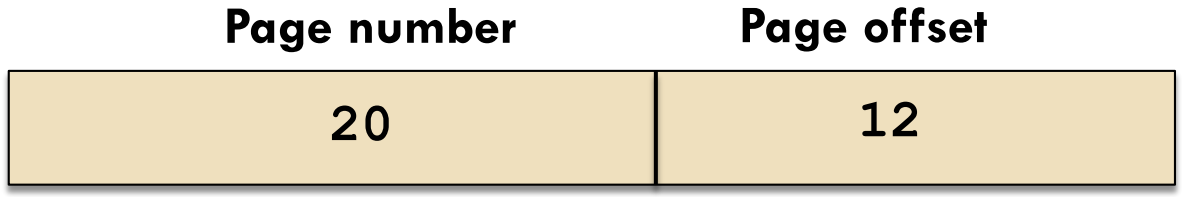
- Logical address spaces: $2^{32} \sim 2^{64}$
- Page size: 4KB = 2^2x 2^{10}= 2^{12}
- Number of page table entries?
  - Logical address space size/page size
  - $2^{32}/2^{12} = 2^{20} \approx$ 1 million entries
- Page table entry = 4 bytes
  - Page table for process = 2^{20} x 4 = 4 MB

Issues with large page tables

- Cannot allocate page table **contiguously** in memory

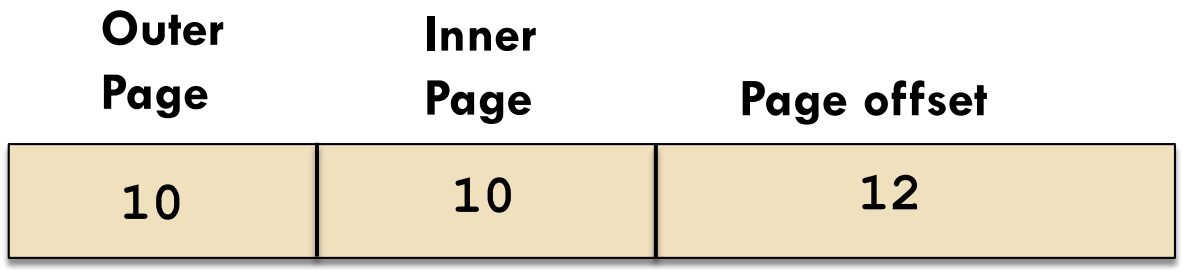
- Solution:
  - Divide the page table into smaller pieces
    - **Page the page-table**

Two-level Paging



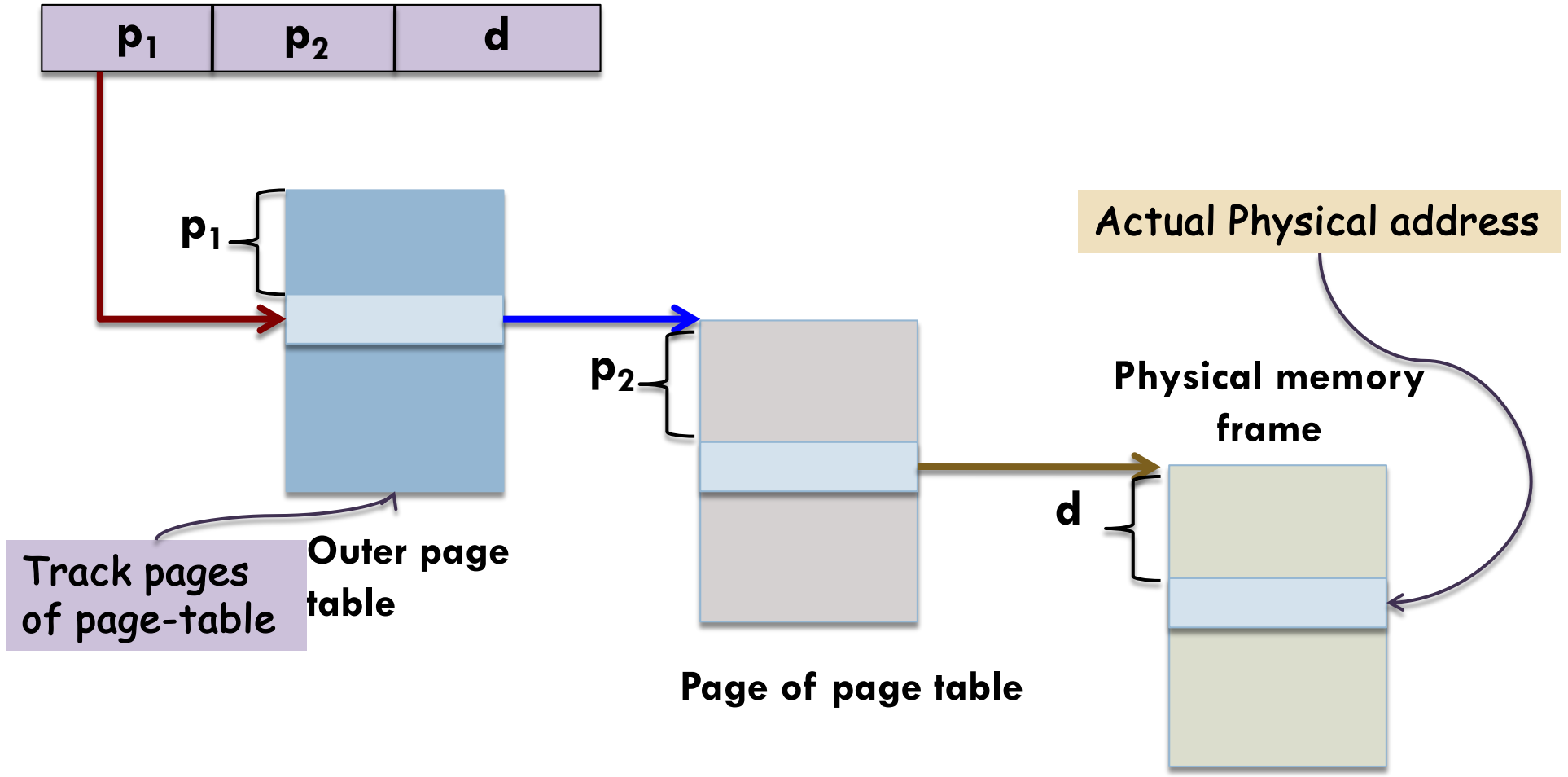
32-bit logical address

Two-level Paging



32-bit logical address

Address translation in two-level paging



Hashed Page Tables

Hashed page tables

- An approach for handling address spaces > 2^{32}
- Virtual page number is **hashed**
  - Hash used as *key* to enter items in the hash table
- The *value* part of table is a **linked list**
  - Each entry has:
    1. Virtual page number
    2. Value of the mapped page frame
    3. Pointer to next element in the list

Searching through the hashed table for the frame number

- Virtual page number is hashed
  - Hashed key has a corresponding value in table
    - Linked List of entries

- **Traverse** linked list to
  - Find a *matching virtual page* number

Hash tables and 64-bit address spaces

- Each entry refers to *several pages* instead of a single page
- **Multiple** page-frame mappings per entry
  - Clustered page tables
- Useful for sparse address spaces where memory references are non-contiguous (and scattered)

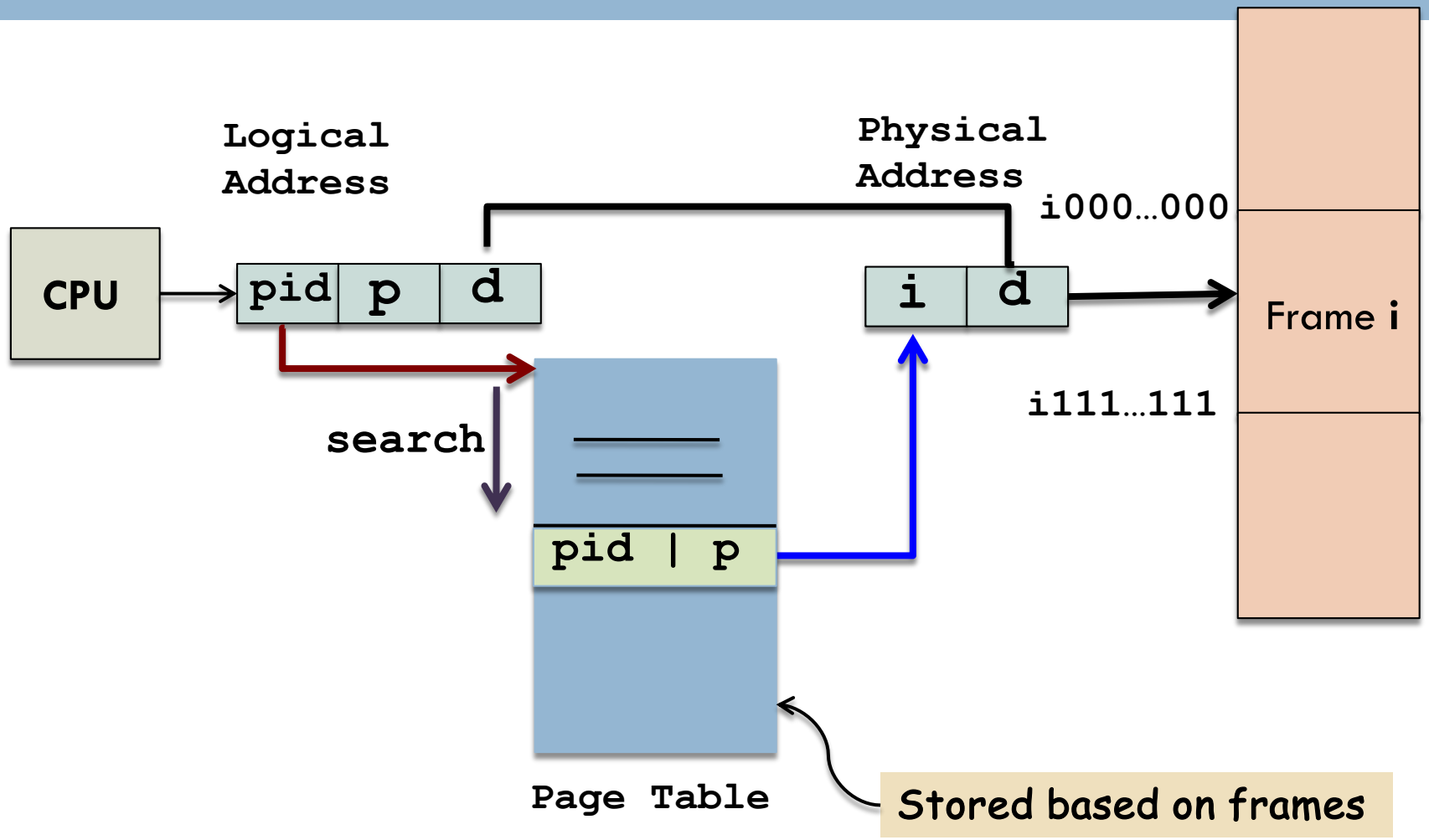
Inverted Page Tables

Inverted page table

- Only **1** page table in the system
  - Has an entry for each **memory frame**

- Each entry tracks
  - Process that owns it (**pid**)
  - Virtual address of page (**page number**)

Inverted Page table



Profiling the inverted page table

- *Decreases* the **amount of memory** needed
- **Search time** *increases*
  - During page dereferencing
- **Stored based on frames**, but searched on pages
  - Whole table might need to be searched!

Other issues with the inverted page table

- Shared paging
  - Multiple pages mapped to same physical memory

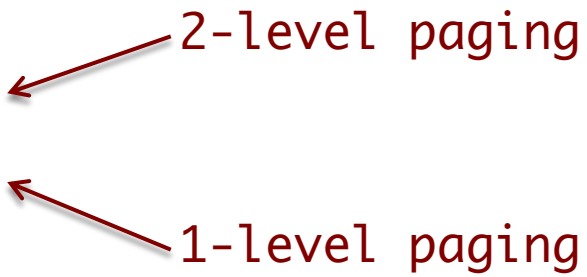
- Shared paging **NOT possible** in inverted tables
  - Only 1 virtual page entry per physical page
    - Stored based on frames

x86-64

- Intel: IA-64 Itanium
  - Not much traction
- AMD: x86-64
  - Intel adopted AMD's x86-64 architecture
- 64-bit address space: 2^{64} (16 exabytes)
- Currently x86-64 provides
  - 48–bit virtual address
  - Page sizes: 4 KB, 2 MB, and 1 GB
  - 4-level paging hierarchy

ARM architectures

- iPhone and Android systems use this
- 32-bit ARM
  - 4 KB and 16 KB pages
  - 1 MB and 16 MB pages



There are two levels for TLBs:
A separate TLB for data
Another for instructions

SEGMENTATION

In our discussions so far …

- Virtual memory is **one-dimensional**
  - Logical addresses go from `0` to some `max` value

- Many problems can benefit from having two or more **separate** virtual address spaces

A compiler has many tables that are built up as compilation proceeds

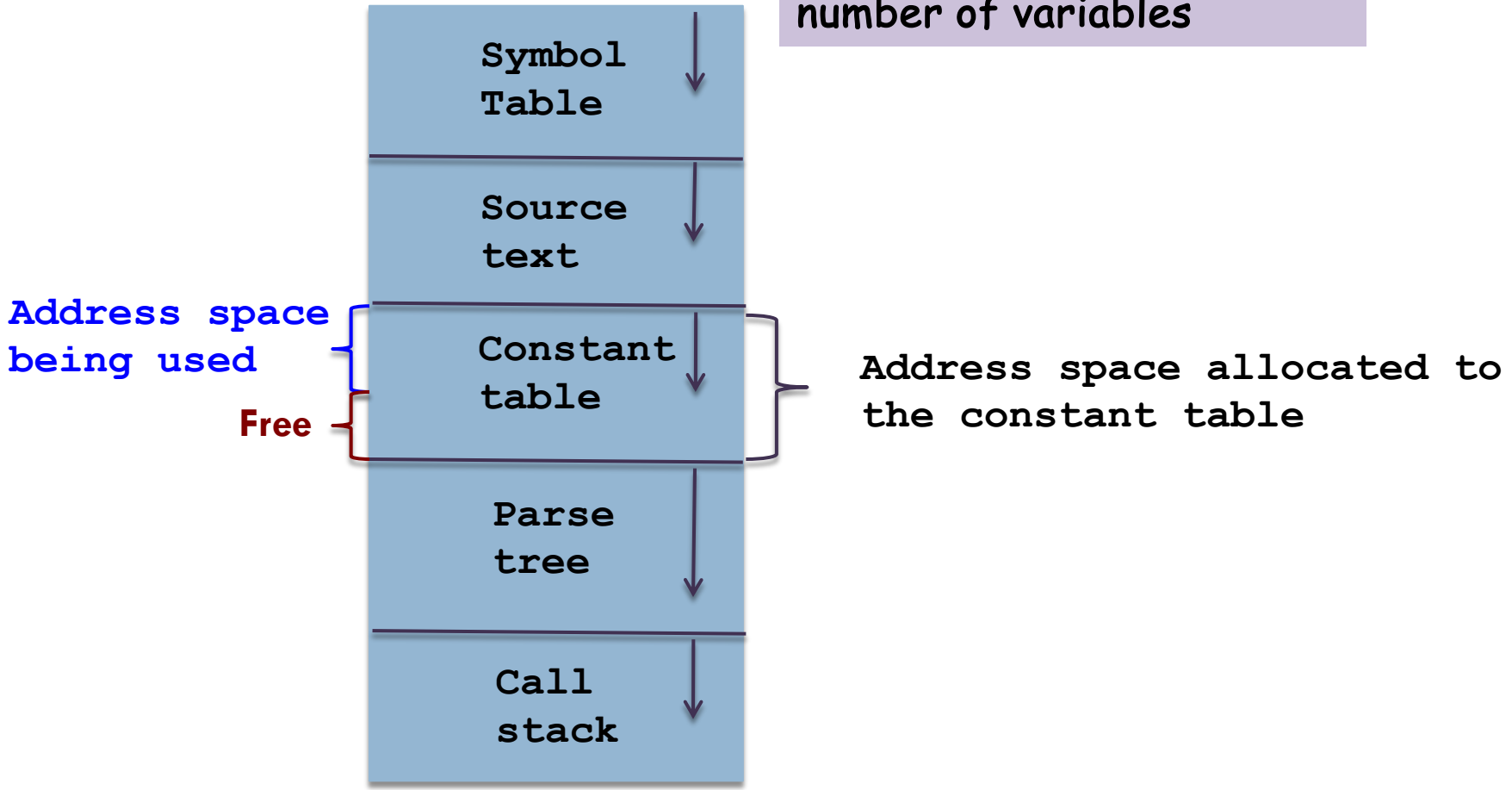
- Source Text
- Symbol table
  - Names and attributes of variables
- Constants Table
  - Integer and floating point constants
- Parse tree
  - Syntactic analysis of program
- Stack
  - Procedure calls within the compiler

Grows continuously as compilation proceeds

Grows and shrinks in unpredictable ways during compilation

One dimensional address space with growing tables

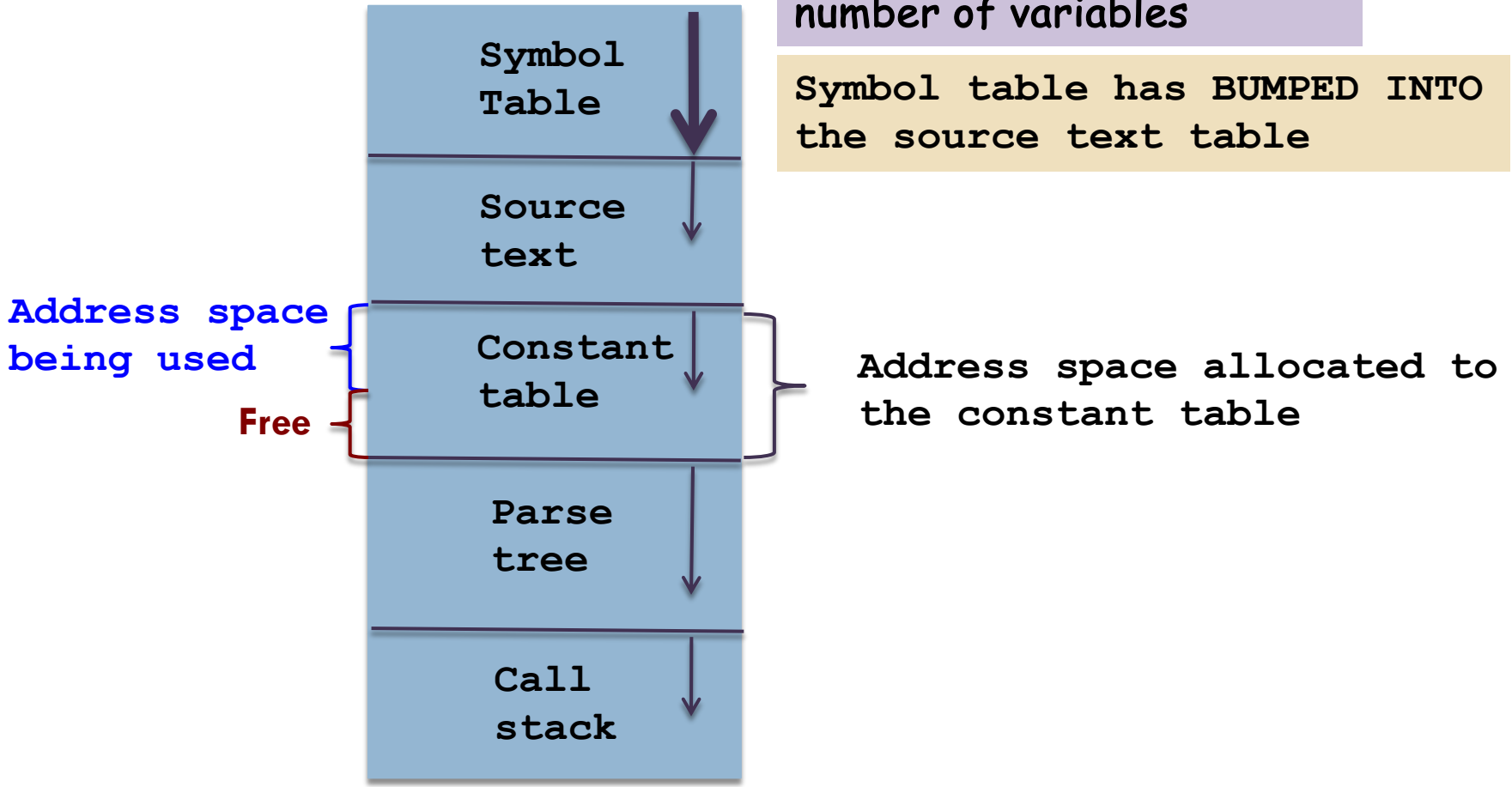
Program has an exceptional number of variables



One dimensional address space with growing tables

Program has an exceptional number of variables

Symbol table has BUMPED INTO the source text table



Options available to the compiler

- Say that compilation cannot continue
  - Not cool
- Play Robin Hood
  - Take space from tables with room
  - Give to tables with little room

What would be really cool …

- Free programmer from having to manage expansion and contraction of tables

But how?

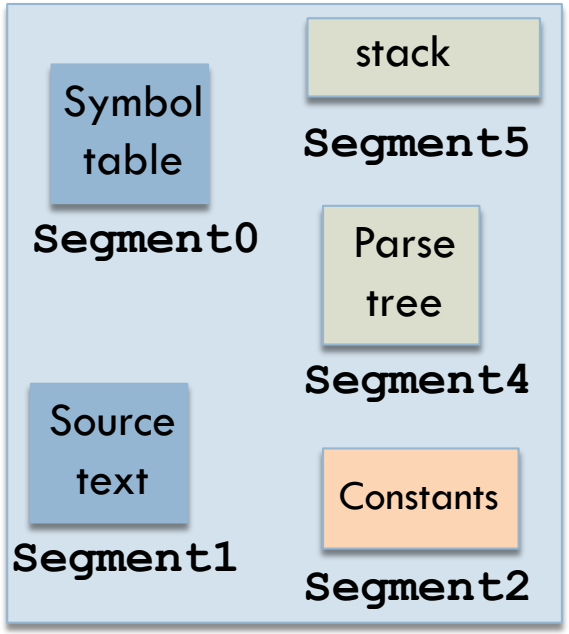
- Provide many completely **independent address spaces**
  - Segments
- A segment has linear sequence of addresses
  - 0 to max

Other things about segments

- Different segments can and do have different lengths

- Segments grow and shrink independently without affecting each other
  - Size increase: something pushed on stack segment
  - Size decrease: something popped off of stack segment

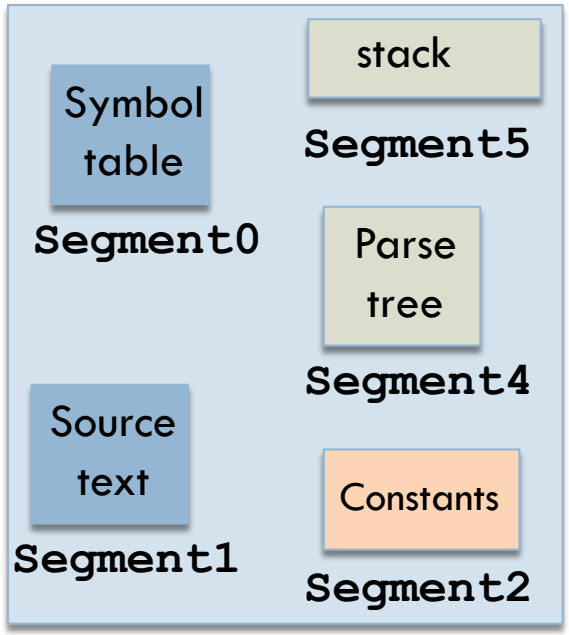
Users view memory as a collection of variable-sized segments



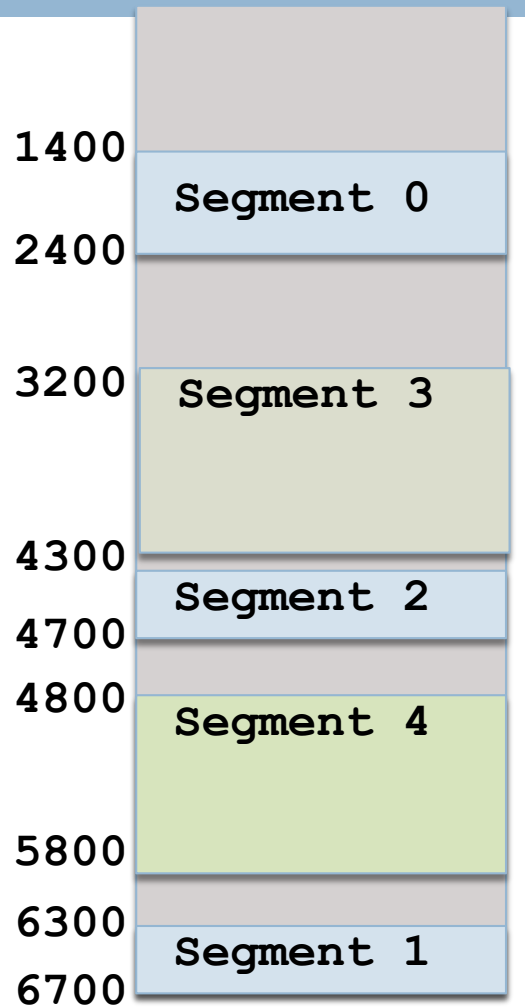
Segmentation

- Logical address space is a collection of segments
- Segments have name and length
- Addresses specify
  - Segment name
  - Offset within the segment
- Tuple: **`<segment-number, offset>`**

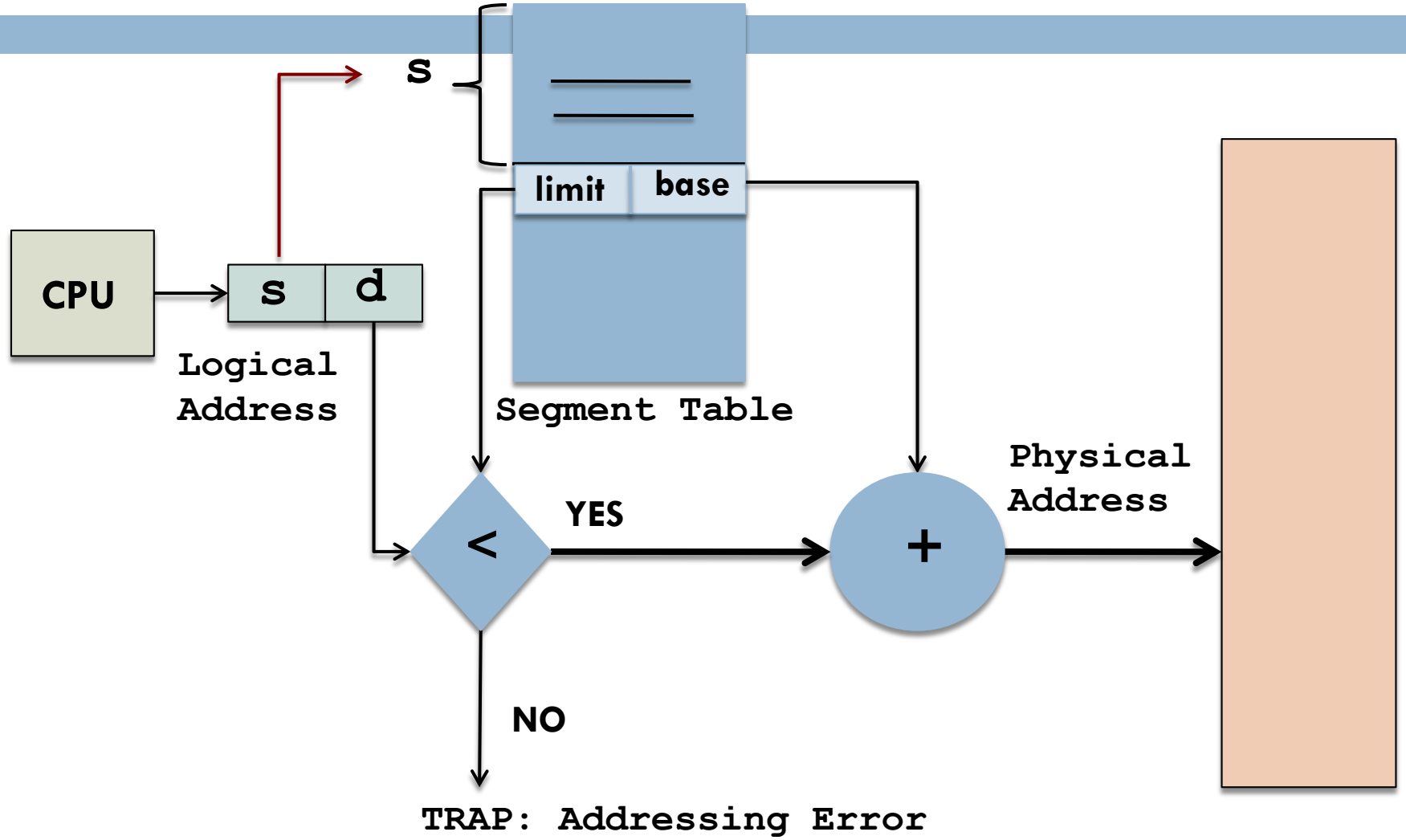
Segmentation Addressing Example



	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1000	3200
4	1000	4800



Segmentation Hardware



Rationale for Paging and Segmentation

- Get a large linear address space **without** having to buy more physical memory
  - PAGING

- Allow programs and data to be broken up into **logically independent** address spaces
  - Aids Sharing AND Protection
    - Segmentation

Comparison of Paging and Segmentation

Consideration	Paging	Segmentation
How many linear address spaces are there?	1	Many
Can total address space exceed physical memory	YES	YES
Can procedures and data be distinguished and protected separately?	NO	YES
Can fluctuating table sizes be accommodated?	NO	YES

Comparison of Paging and Segmentation

Consideration	Paging	Segmentation
Should the programmer be aware the the technique is being used?	NO	YES
Is sharing of procedures between users facilitated?	NO	YES
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to allow sharing and protection

Segmentation with Paging

- Multics: Each program can have up to 256K independent segments
  - Each with 64K 36-bit words
- Intel Pentium
  - 16K independent segments
  - Each segment has 10^9 32-bit words
  - Few programs need more than 1000 segments, but many programs need large segments

VIRTUAL MEMORY

Memory Management: Why?

- Main objective of system is to execute programs
- Programs and data must be **in memory** (*at least partially*) during execution
- To improve CPU utilization and response times
  - Several processes need to be memory resident
  - Memory needs to be **shared**

Requiring the entire process to be in physical memory can be limiting

- **Limits** the size of a program
  - To the size of physical memory

- BUT the entire program is not always needed

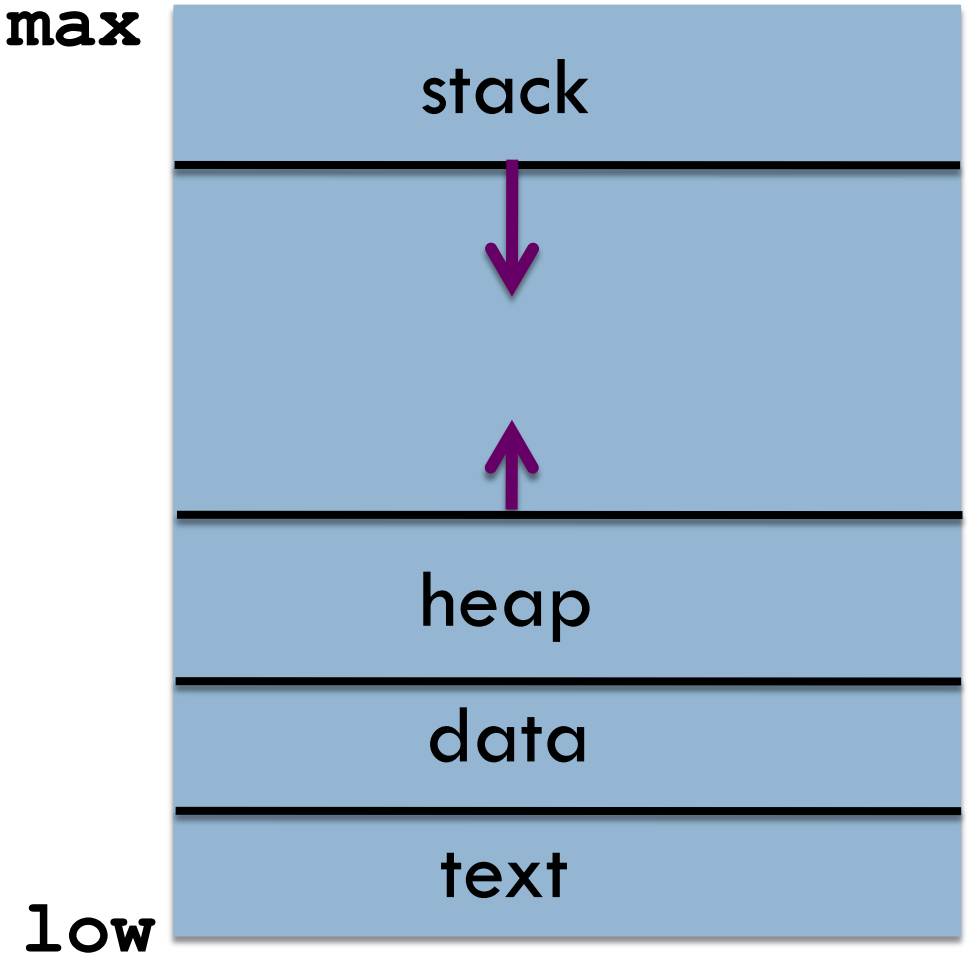
Situations where the entire program need not be memory resident

- Libraries
- Code to handle rare error conditions
- Data structures are often allocated more memory than they need
  - Arrays, lists …
- Rarely used features

What if we could execute a program that is partially in memory?

- Program is **not constrained** by amount of free memory that is available
- Each program uses **less** physical memory
  - So, more programs can run
- **Less I/O** to swap programs back and forth

Logical view of a process in memory



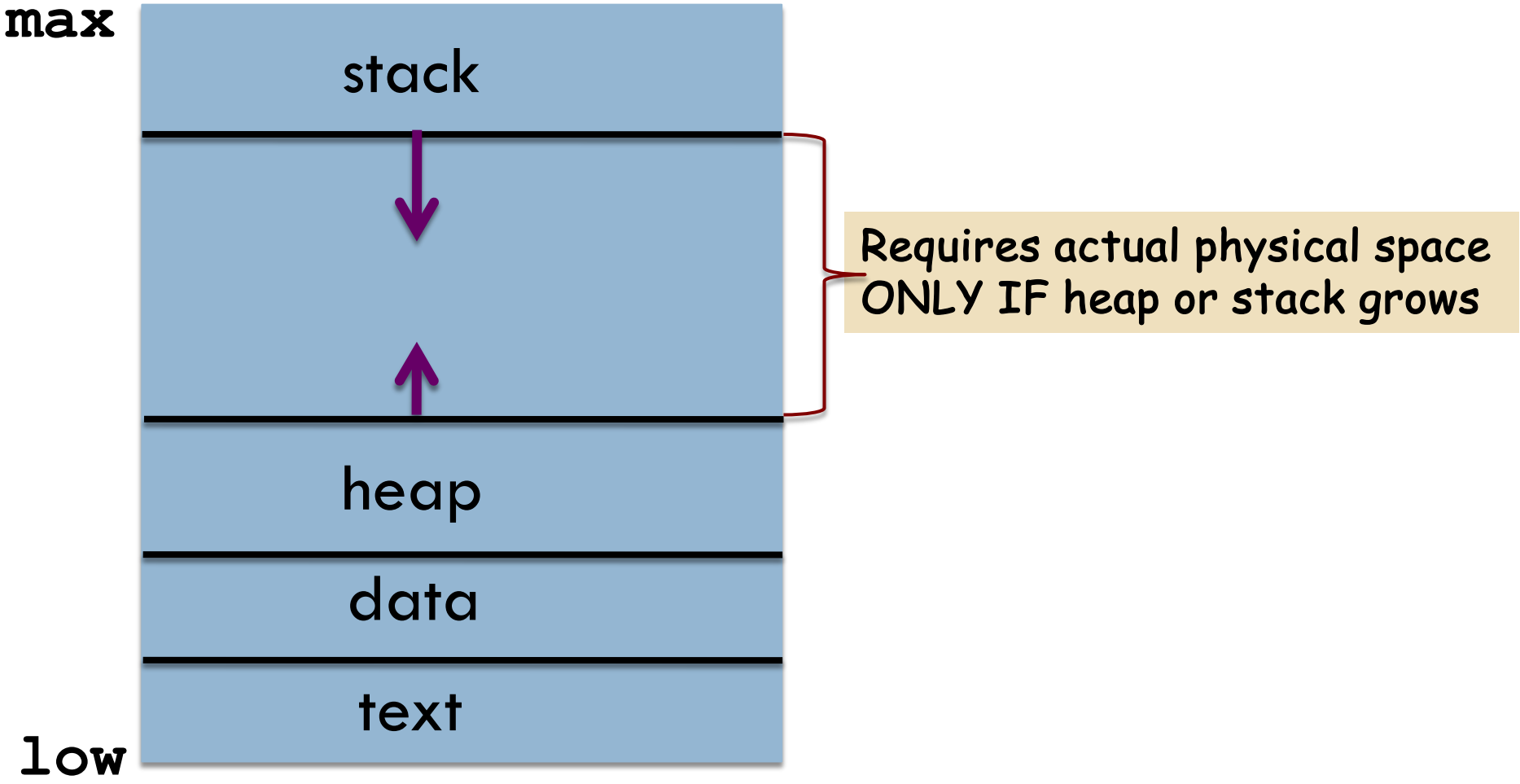
{Function parameters, return addresses, and local variables}

{Memory allocated dynamically during runtime}

{Global variables}

{Program code}

Logical view of a process in memory



Sparse address spaces

- Virtual address spaces with holes

- Harnessed by
  - Heap or stack segments
  - Dynamically linked libraries

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]*