# CS 370: OPERATING SYSTEMS
# [MEMORY MANAGEMENT]

Shrideep Pallickara
Computer Science
Colorado State University



1

## Frequently asked questions from the previous class survey

- Why does the CPU not need to know the physical address?
- Address translation
  - Address translation being performed even memory capacities are low?
  - Is address translation deterministic
- Surely the heap and stack are not based on logical addresses?
- How can a process move around in memory as it executes? Seems inefficient?
- Say I have a variable x; does its logical address change during program execution? What about its physical address?
- Memory implementations: SRAM, DRAM, DFFs, GDDR, HBMs, and such …



2

## Link, load, run: Build it, place it, move in

| Linking | Loading | Execution |
|---|---|---|
| *Build the house* | *Place it on a lot* | *Move in* |
| Linker · at build time | Loader · at launch | CPU · at run time |
| **THE HOUSE** Prefab modules arrive separately, each pipe end is tagged with what it joins. | **THE HOUSE** The crew sets the finished house on a free lot and readies it for occupancy. | **THE HOUSE** The occupants move in: they walk the halls and use the kitchen. |
| **THE PROGRAM** Object files refer to symbols defined elsewhere. The linker joins them and resolves each reference. | **THE PROGRAM** The loader places the executable in memory and binds relocatable addresses to absolute ones. | **THE PROGRAM** The CPU fetches instructions and reads and writes data in memory. |

Seen `undefined reference to 'foo'`? The compiler was happy; the linker found a pipe tagged for a part that never shipped.



3

## Three ways a house gets what it needs: Built in, piped in, or brought in

| | Static linking *Built in* | Dynamic linking *Piped in* | Dynamic loading *Brought in* |
|---|---|---|---|
| **The house** | Every house has its own well and water heater. 100 houses, 100 wells. | A capped pipe tagged "city water"; the first tap connects it to the shared main. | Your holiday decorations stay boxed in storage; you fetch them when needed. |
| **The program** | Each executable carries its own copy of the library code. | A stub locates (or loads) the routine, then replaces itself with the routine's address. | A routine stays on disk until first called. Never called, never loaded. |
| **Joined when?** | Before it runs | On first call | On first call |
| **Whose code?** | Library, copied in | Library, shared | The program's own |
| **Needs the OS?** | No | **Yes: shared pages** | No: program design |

**Dynamic linking vs. dynamic loading**: same timing, different owner. The city's shared main needs the **OS**; your own boxes don't.



4

# Relocation: Steps from the front door




With an MMU, occupants only ever say "346"; it adds the lot's address: `14000 + 346 = 14346`

**Compile-time binding**

Blueprints say "pantry at `14346` Elm St." Move the house? Redraw the blueprints (recompile).

**Load-time binding**

The crew rewrites every "346 from the door" into a street address as it sets the house down. Move it? Redo them all.

**Execution-time binding**

Directions stay **relative**; the MMU adds the lot address on *every* trip. The house can move after you've moved in → **swapping**



# Logical address spaces in action

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char *argv[]) {
4    printf("location of code : %p\n", main);
5    printf("location of heap : %p\n", malloc(100e6));
6    int x = 3;
7    printf("location of stack: %p\n", &x);
8    return x;
9  }
```

Output when run on a 64-bit Mac

```
location of code : 0x1095afe50
location of heap : 0x1096008c0
location of stack: 0x7fff691aea64
```

## Topics covered in this lecture

- Contiguous memory allocations
- Fragmentations
  - External and Internal
- Segmentation
- Paging



7

I know, I can't be late
Supper's waiting on the table
Tomorrow's just an excuse away
So I pull my collar up and face the cold, on my own

The earth laughs beneath my heavy feet
At the blasphemy in my old jangly walk
Steeple, guide me to my heart and home
The sun is out and up and down again

Thirty-three, William Patrick Corgan; Smashing Pumpkins

*Each process is contained in a single continuous section of memory*

## CONTIGUOUS MEMORY ALLOCATION

8

## Partitioning of memory

- Main memory needs to **accommodate** the OS and user processes
- Divided into two partitions
  - Resident OS
    - Usually low memory
  - User processes



## Memory Mapping and Protection

- Base register (also referred to as a *relocation* register)
  - Smallest physical address
- Limit register
  - Range of logical addresses

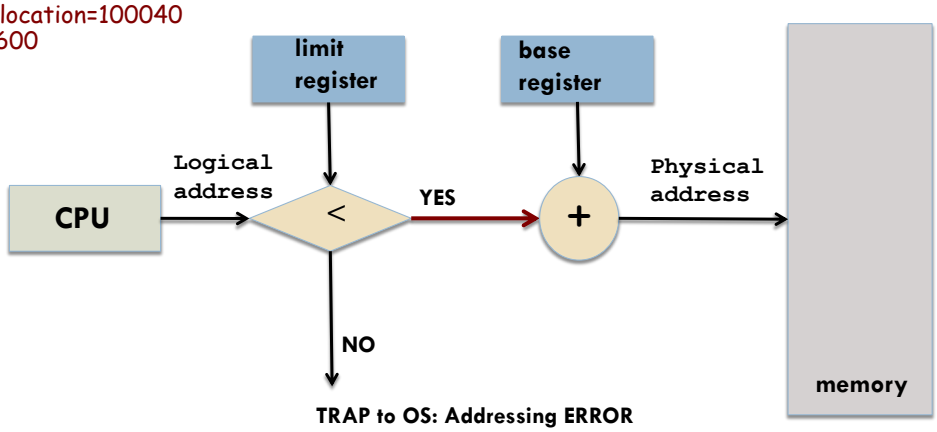
## Memory Mapping and Protection

- When CPU scheduler selects a process for execution
  - Base and limit registers reloaded as part of context switch
- Every address generated by the CPU
  - Checked against the relocation(base)/limit registers



## Memory Mapping and Protection

E.g.: base/relocation=100040 and limit=74600



## Memory Allocation: Fixed Partition method

- **Divide** memory into several **fixed-size** partitions
  - Each partition contains exactly one process
- Degree of multiprogramming
  - Bound by the number of partitions



## Memory allocation: Variable-partition method [1/2]

- Used in batch environments
- OS maintains table tracking memory utilization
  - What is available?
  - Which ones are occupied?
- Initially all memory is available
  - Considered a large **memory gap**
  - Eventually *many* memory gaps will exist

## Memory allocation: Variable-partition method [2/2]

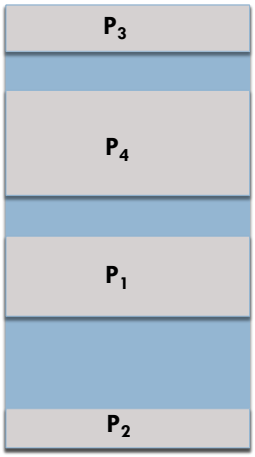
- OS orders processes according to the scheduling algorithm
- Memory allocated to processes until requirements of the next process cannot be met
  - *Wait* till a larger block is available
  - *Check* if smaller requirements of other processes can be met



## Variable-partition method: Reclaiming spaces

- When process arrives, if space is too large
  - Split into two
- When process terminates?
  - If released memory is adjacent to other *memory gaps*
    - **Fuse** to form a larger space

## Splitting and Fusing Memory spaces






## Dynamic Storage Allocation Problem

- Satisfying a request of size ***n*** from the set of available spaces
  - First fit
  - Best fit
  - Worst fit

## First fit

- Scan list of segments until you find a memory-gap that is big enough
- Gap is broken up into two pieces
  - One for the process
  - The other is unused memory



## Best Fit

- Scan the entire list from beginning to the end
- Pick the smallest memory-gap that is adequate to host the process

## Comparing Best Fit and First Fit

- Best fit is **slower** than first fit
- Surprisingly, best fit also results in more **wasted memory** than first fit
  - Tends to fill up memory with tiny, useless gaps



## Worst fit

- How about going to the other extreme?
  - Always take the largest available memory-gap
  - Perhaps, the new memory-gap would be useful
- Simulations have shown that worst fit is not a good idea either




23

## Base and limits translation lacks many of the features needed to support modern programs

- Base and limits translation supports only **coarse-grained** protection at the level of the *entire* process
  - It is not possible to prevent a program from overwriting its own code, for example
  - It is also **difficult to share** regions of memory between two processes
  - Since the memory for a process needs to be contiguous …
    - Supporting dynamic memory regions, such as for heaps, thread stacks, or memory mapped files, becomes difficult to impossible



24

## In our discussions so far …

- Logical/virtual memory is **one-dimensional**
  - Logical addresses go from 0 to some `max` value
- Many problems can benefit from having two or more **separate** logical address spaces



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

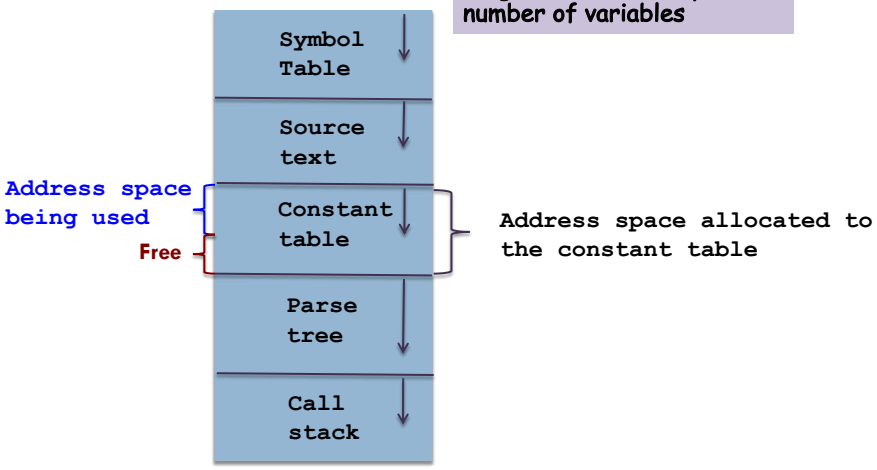
- Source Text
- Symbol table
  - Names and attributes of variables
- Constants Table
  - Integer and floating point constants
- Parse tree
  - Syntactic analysis of program
- Stack
  - Procedure calls within the compiler

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



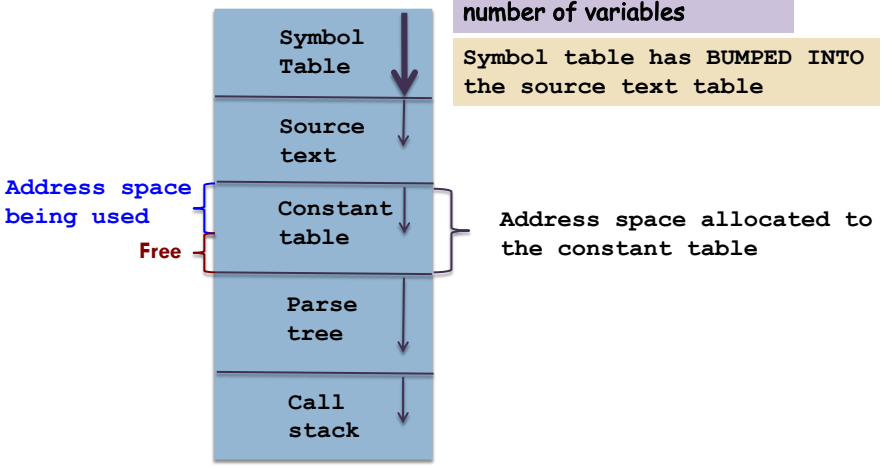



27

## One dimensional address space with growing tables

Program has an exceptional number of variables

Symbol table has BUMPED INTO the source text table






28

## 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**
- Each segment has linear sequence of addresses
  - 0 to max



## Segments and Base/Limit registers

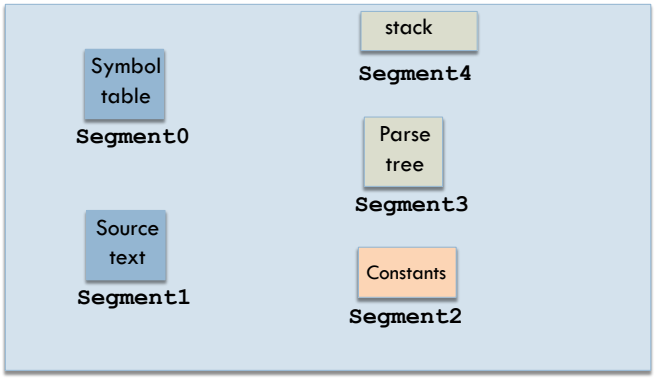
- The hardware supports an **array** of pairs of base and bounds registers, for *each process*
  - **Segment Table**
- Each entry in the array controls a portion, or **segment**, of the virtual address space
- The physical memory for **each segment is stored contiguously**, but different segments can be stored at different locations
  - For example, code and data segments are not immediately adjacent to each other in either the virtual or physical address space

## Other things about segments

- Different segments can and do have different lengths
- Segments grow and shrink independently without affecting each other; For example, consider a segment for the stack
  - Size increase: something pushed on stack segment
  - Size decrease: something popped off of stack segment



## Segmentation allows users to view memory as a collection of variable-sized segments

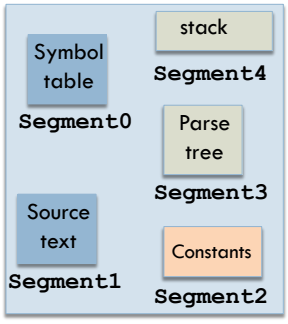


## Segmentation

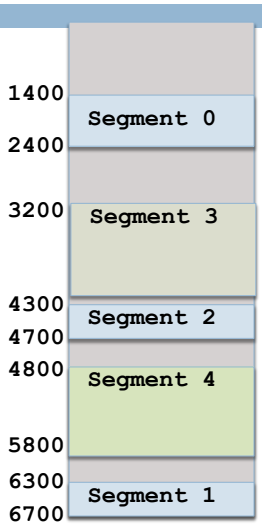
- Logical address space is a collection of segments
- Segments have name and length
- Addresses specify
  - Segment name
  - Offset within the segment
- Tuple: **<segment-number, offset>**

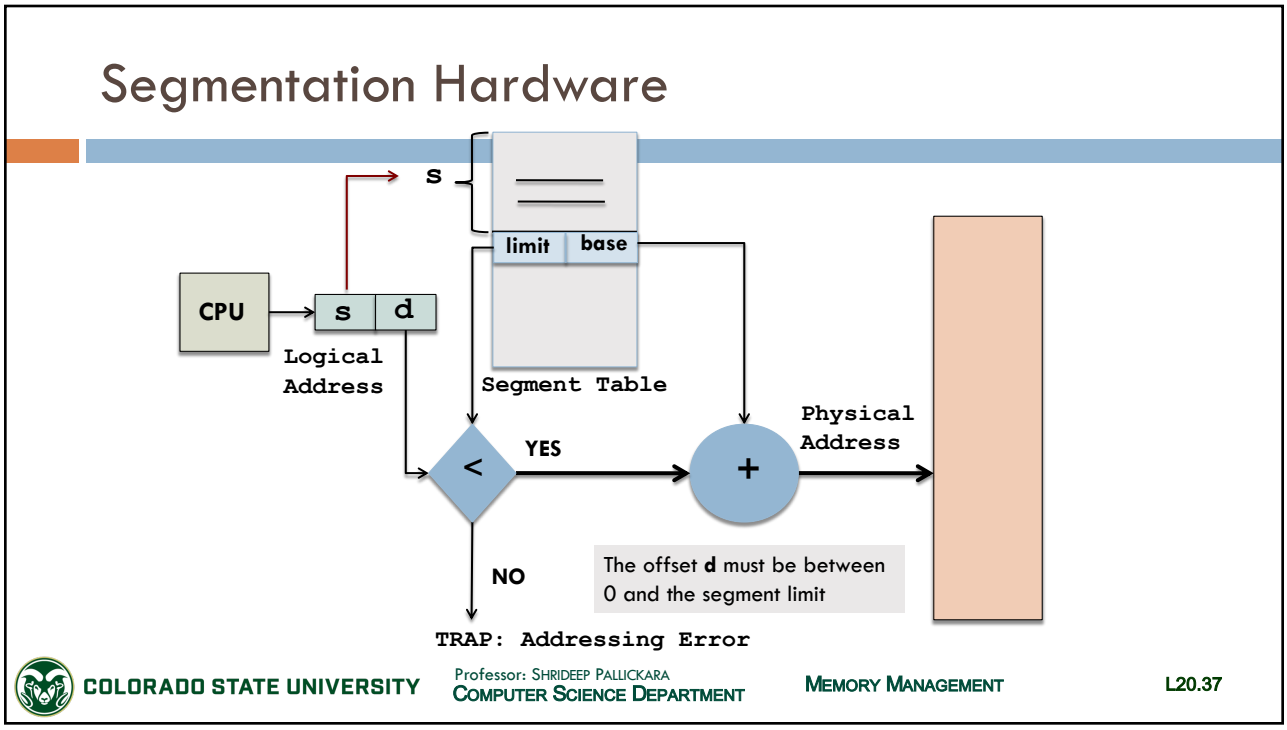


## Segmentation Addressing Example




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




Segmentation Hardware
s
limit
base
CPU
s
d
Logical
Address
Segment Table
<
YES
+
Physical
Address
NO
The offset d must be between
0 and the segment limit
TRAP: Addressing Error
COLORADO STATE UNIVERSITY
Professor: SHRIDEEP PALLICKARA
COMPUTER SCIENCE DEPARTMENT
MEMORY MANAGEMENT
L20.37

37


FRAGMENTATION

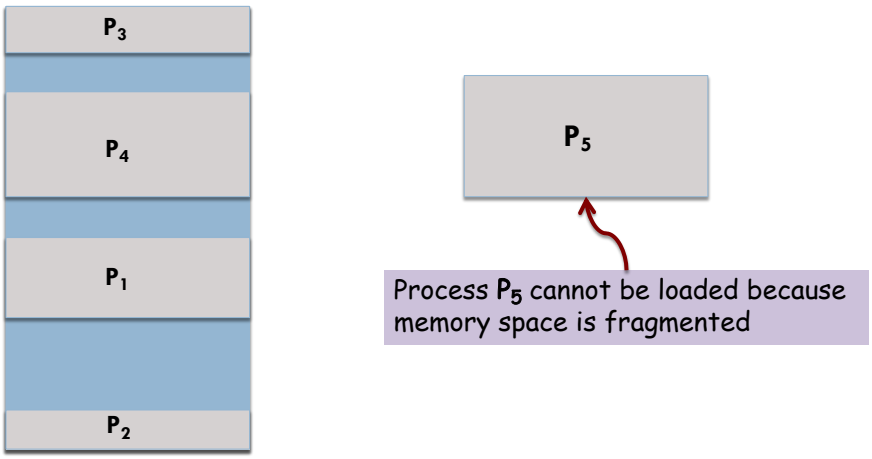
38

## Contiguous Memory Allocation: Fragmentation

- As processes (and segments) are loaded/removed from memory
  - Free memory space is **broken** into small pieces
- **External fragmentation**
  - Enough space to satisfy request; BUT
  - Available spaces are *not contiguous*



## Fragmentation: Example



## Fragmentation can be internal as well

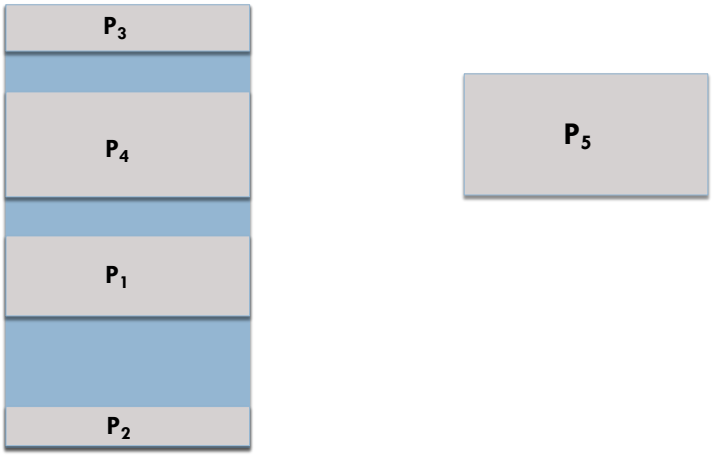
- Memory allocated to process may be *slightly larger* than requested
- **Internal fragmentation**
  - Unused memory is internal to blocks



## Compaction: Solution to external fragmentation

- **Shuffle** memory contents
  - Objective: Place free memory into large block
- Not possible if relocation is static
  - Load time
- Approach involves moving:
  1. Processes towards one end
  2. Gaps towards the other end

## Compaction: Example






## Memory compaction is time intensive and is usually not done

- Let's consider a machine with 1 GB of RAM
- The machine can copy 4 bytes in 20 nanoseconds
- Time to compact all the memory?

  $10^9 \times (20 \times 10^{-9}/4) = 5$ seconds (approximately)

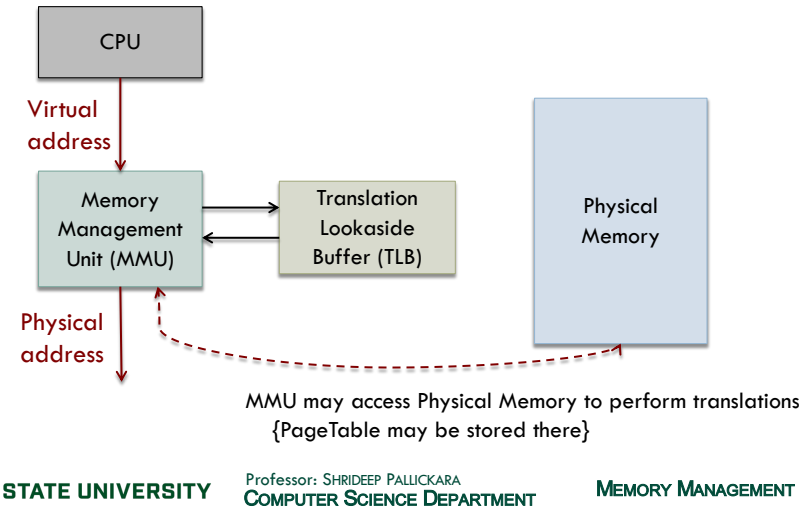
  Note: 1 GB is approximately $10^9$ bytes

# PAGING:
## OVERVIEW OF THE MAPPING PROCESS

COMPUTER SCIENCE DEPARTMENT
COLORADO STATE UNIVERSITY

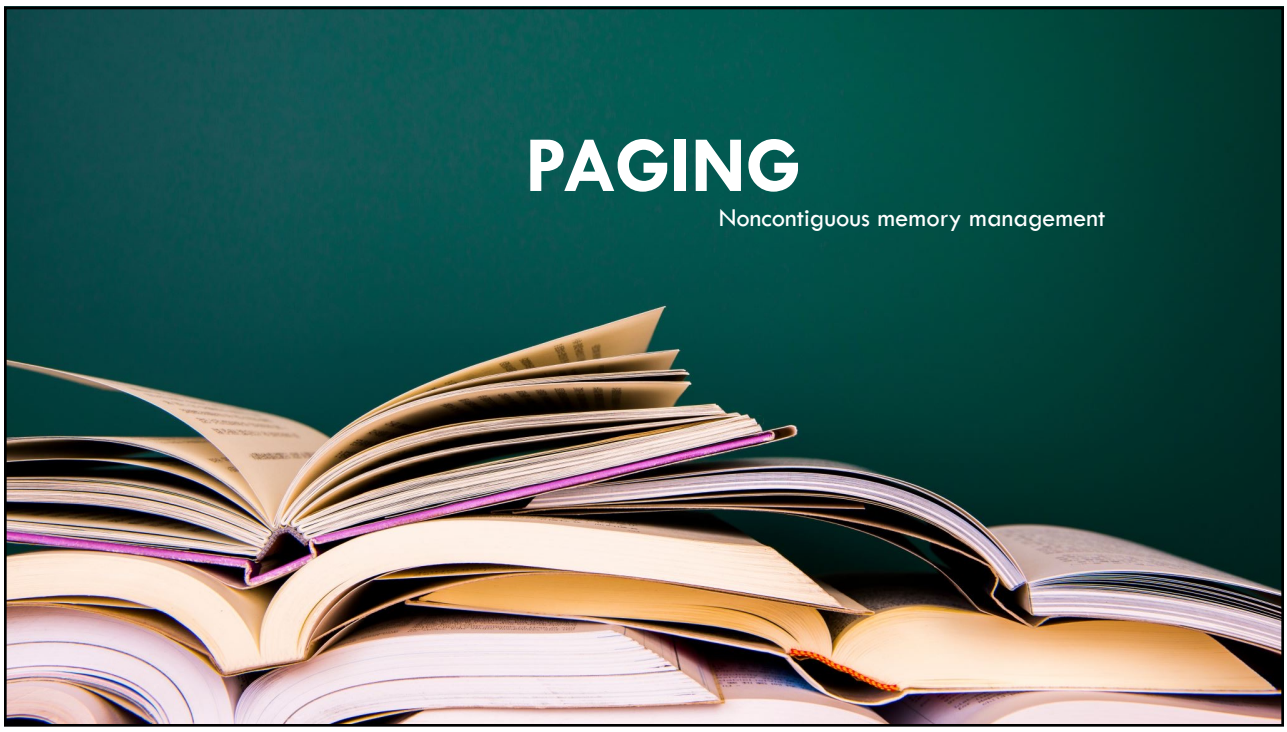
45

## Overview of how mapping of logical and physical addresses is performed




COLORADO STATE UNIVERSITY
Professor: SHRIDEEP PALLICKARA
COMPUTER SCIENCE DEPARTMENT
MEMORY MANAGEMENT
L20.46

46




47

## The Paging memory management scheme

- Physical address space of process can be **non-contiguous**
- Solves problem of fitting variable-sized memory chunks to backing store
  - Backing store has fragmentation problem
    - Compaction is impossible



48

## Basic method for implementing paging

- Break memory into **fixed-sized** blocks
  - Physical memory: **frames**
  - Logical memory: **pages**

  } **Same size**

- Backing store is also divided the same way



## 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 and Herbert Bos. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620. [Chapter 3]*
- *Thomas Anderson and Michael Dahlin. Operating Systems Principles and Practice. 2nd Edition. Recursive Books. ISBN: 978-0985673529.* [Chapter 8]
- *Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. Operating Systems: Three Easy Pieces. 1st edition. CreateSpace Independent Publishing Platform. ISBN-13: 978-1985086593. [Chapter 14]*