

# CS 370: OPERATING SYSTEMS

## [MEMORY MANAGEMENT]

Shrideep Pallickara  
Computer Science  
Colorado State University

COMPUTER SCIENCE DEPARTMENT



1

## Frequently asked questions from the previous class survey

- Where is the page table stored?
- Does the TLB contain page-to-frame mappings?
- Is there always an eviction on a TLB miss?



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.2

2

## Topics covered in this lecture

- Shared pages
- Page sizes
- Structure of Page tables
  - ▣ Hashed Page Tables
  - ▣ Inverted Page Tables



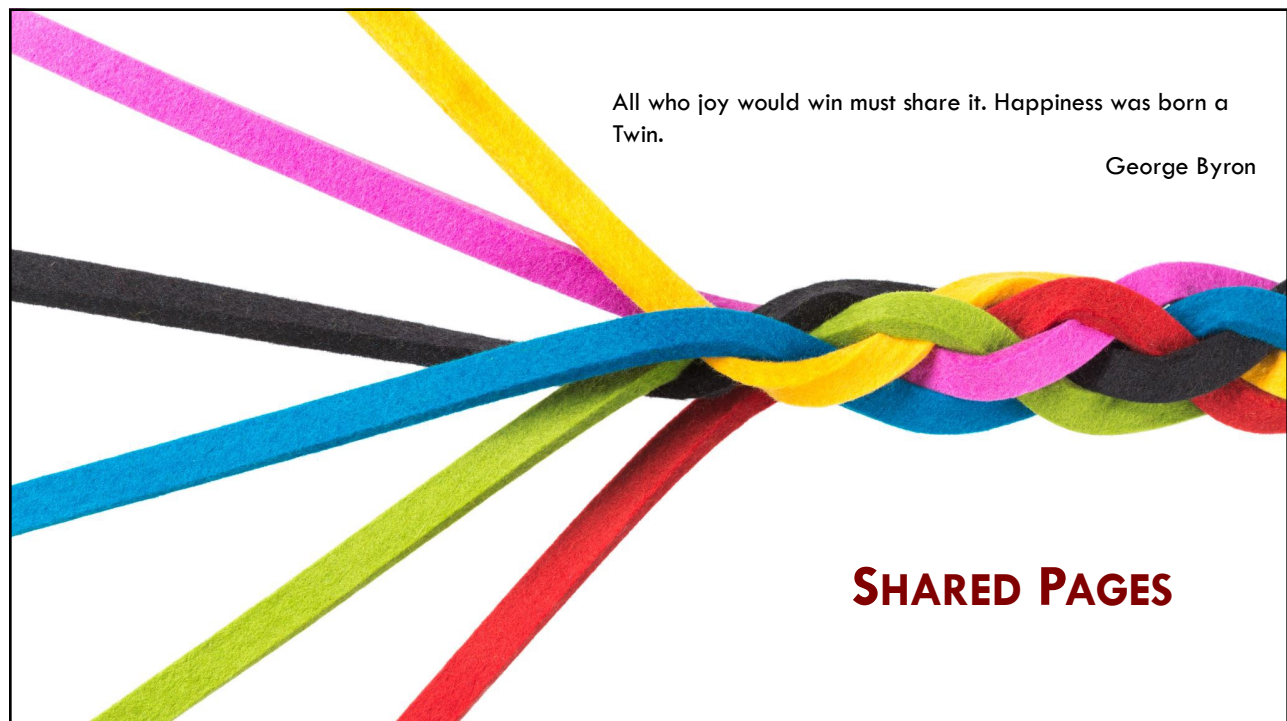
COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.3

3



4

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.5

5

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.6

6

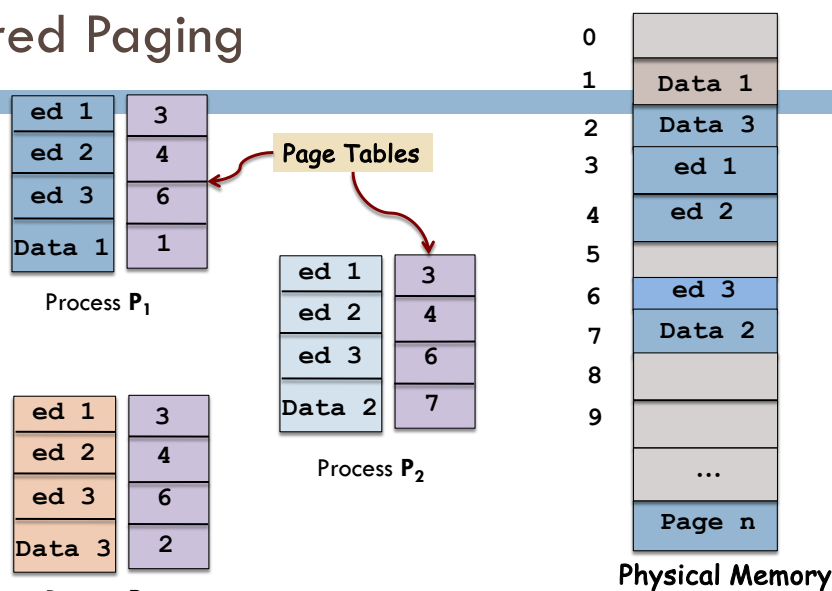
## Shared Pages

- 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 :  $150 + 40 \times 50 = 2150$  KB needed



7

## Shared Paging



8

## 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



9



10

## Paging and page sizes

- On average,  $\frac{1}{2}$  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**



11

## But having small pages is not necessarily efficient

- Small pages mean programs need more pages
  - **Larger** page tables
  - 32 KB program needs
    - 4 8-KB pages, but 64 512-byte pages
- **Context switches** can be *more expensive* with small pages
  - Need to reload the page table



12

## 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
  - $64 \times 15 = 960$  msec to load 64 512-bytes pages
  - $4 \times 25 = 100$  msec to load 4 8KB pages
- Here, **large** pages make sense



13

## 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 =  $se/p + p/2$ 
  - ← Page table overhead
  - ← Internal fragmentation loss



14

## 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  $p$

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

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



## Optimal page size: Considering only page size and internal fragmentation

- $p = \text{sqrt}(2se)$
- $s = 128\text{KB}$  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



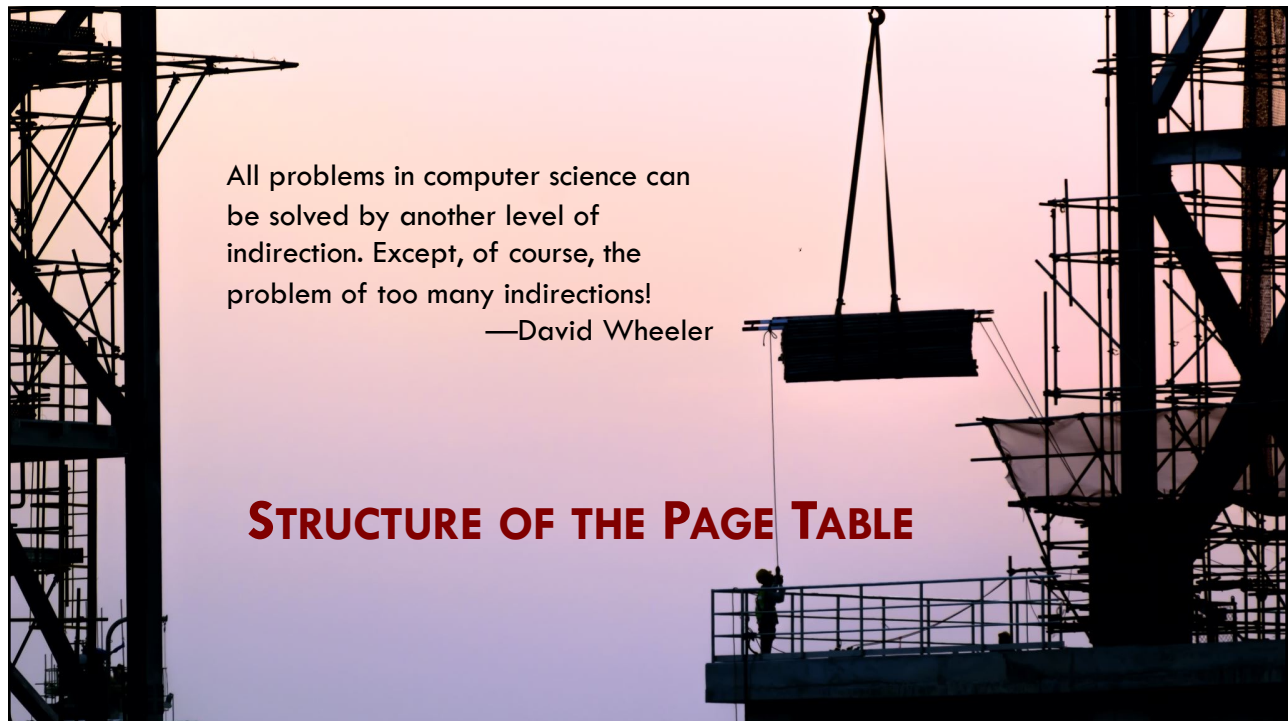


## Pages sizes and size of physical memory

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



17



All problems in computer science can  
be solved by another level of  
indirection. Except, of course, the  
problem of too many indirections!

—David Wheeler

## STRUCTURE OF THE PAGE TABLE

18

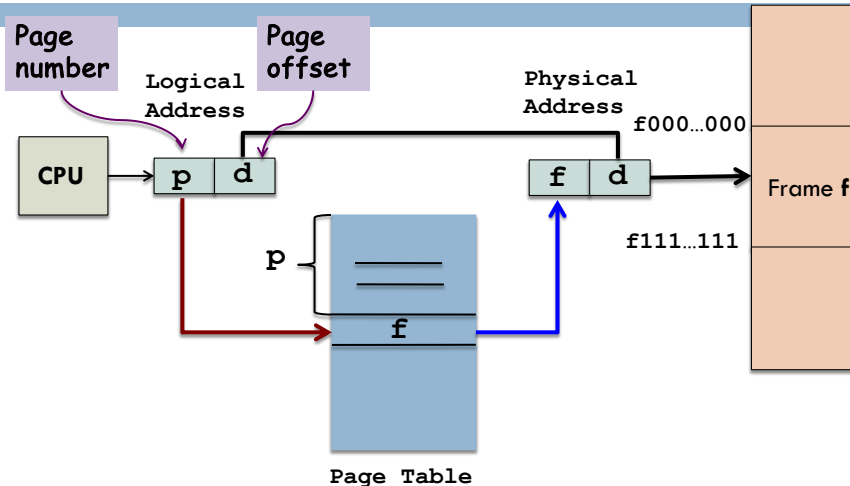
## Typical use of the page table

- 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



19

## Basic Paging Hardware



20

## Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.21

21

## Hierarchical Paging

- Logical address spaces:  $2^{32} \sim 2^{64}$
- Page size:  $4\text{KB} = 2^2 \times 2^{10} = 2^{12}$
- 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 =  $2^{20} \times 4 = 4$  MB



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.22

22

## 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

Page number	Page offset
20	12

32-bit logical address



## Two-level Paging

Outer Page	Inner Page	Page offset
10	10	12

32-bit logical address



COLORADO STATE UNIVERSITY

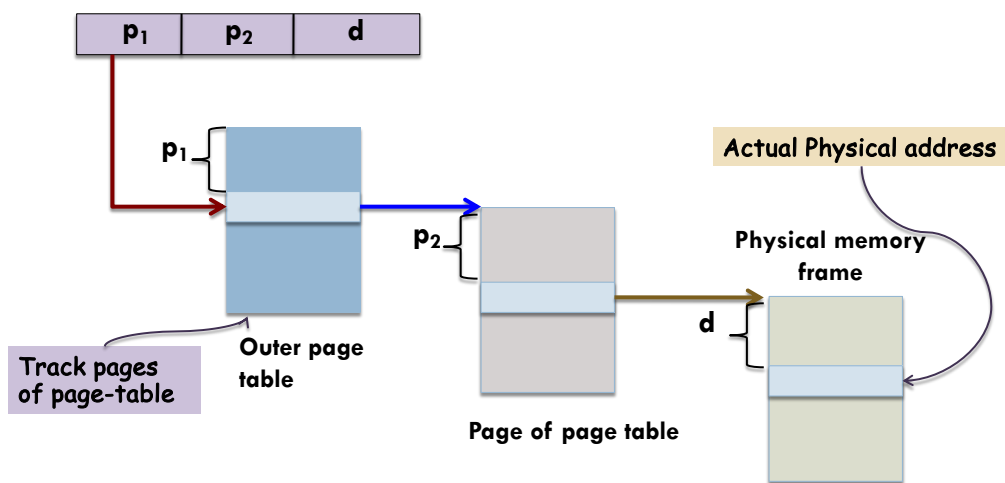
Professor: SHRIDEEP PALLICKARA  
 COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.25

25

## Address translation in two-level paging



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
 COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.26

26

## Two-level Page tables: The outer page table

4 GB address space split into 1024 chunks

Outer Page	Inner Page	Page offset
10	10	12

Each entry represents 4 MB

Page size is 4 KB

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALICKARA COMPUTER SCIENCE DEPARTMENT MEMORY MANAGEMENT L22.27

27

## Two-level Page tables: Case where only a few entries are needed

4 GB address space split into 1024 chunks

Outer Page	Inner Page	Page offset
10	10	12

Each entry represents 4 MB

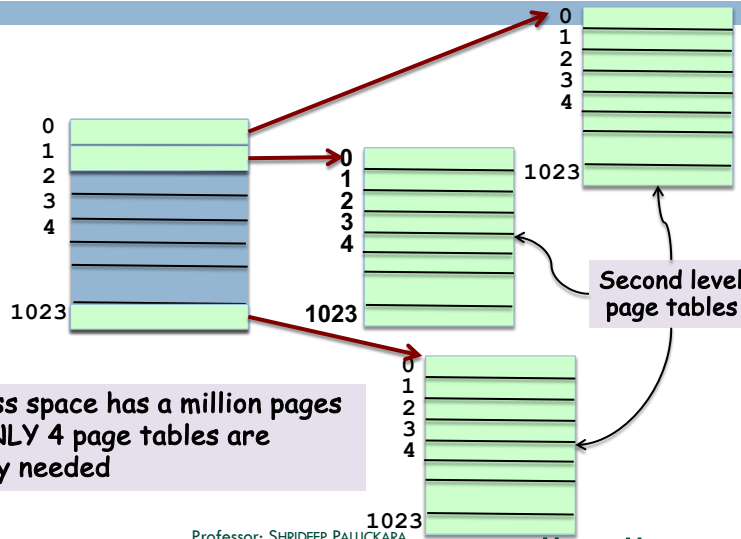
Unused by program

Page size is 4 KB

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALICKARA COMPUTER SCIENCE DEPARTMENT MEMORY MANAGEMENT L22.28

28

## Two-level Page tables



## Computing number of page tables in hierarchical paging

Outer Page	Inner Page	Page offset
11	11	10

- There is 1 outer table with  $2^{11}$  entries
- Each outer table entry points to an inner page table
  - So, there are  $2^{11}$  inner page tables
- Total number of page tables =  $1 + 2^{11}$
- Total number of entries =  $2^{11} + 2^{11} \times 2^{10}$



## Let's try 2-level paging for a 64-bit logical address space

Outer page	Inner page	Page offset
42	10	12

- Outer page table has  $2^{42}$  entries!
- **Divide the outer page table** into smaller pieces?

2 <sup>nd</sup> Outer page	Outer page	Inner page	Page offset
32	10	10	12



## Why hierarchical tables may strain 64-bit architectures

- In our previous example
  - ▣ There would be  $2^{32}$  entries in the outer page table
- We could keep going
  - ▣ 4-level page tables ...
- But all this results in a **prohibitive** number of memory accesses







33

## 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

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT MEMORY MANAGEMENT L22.34

34

## Searching through the hashed table for the frame number

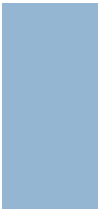
- Virtual page number is hashed
  - ▣ Hashed key has a corresponding value in table
    - Linked List of entries
  
- **Traverse** linked list to
  - ▣ Find a *matching virtual page* number




## Hash tables and 64-bit address spaces

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






# INVERTED PAGE TABLES

COMPUTER SCIENCE DEPARTMENT  COLORADO STATE UNIVERSITY

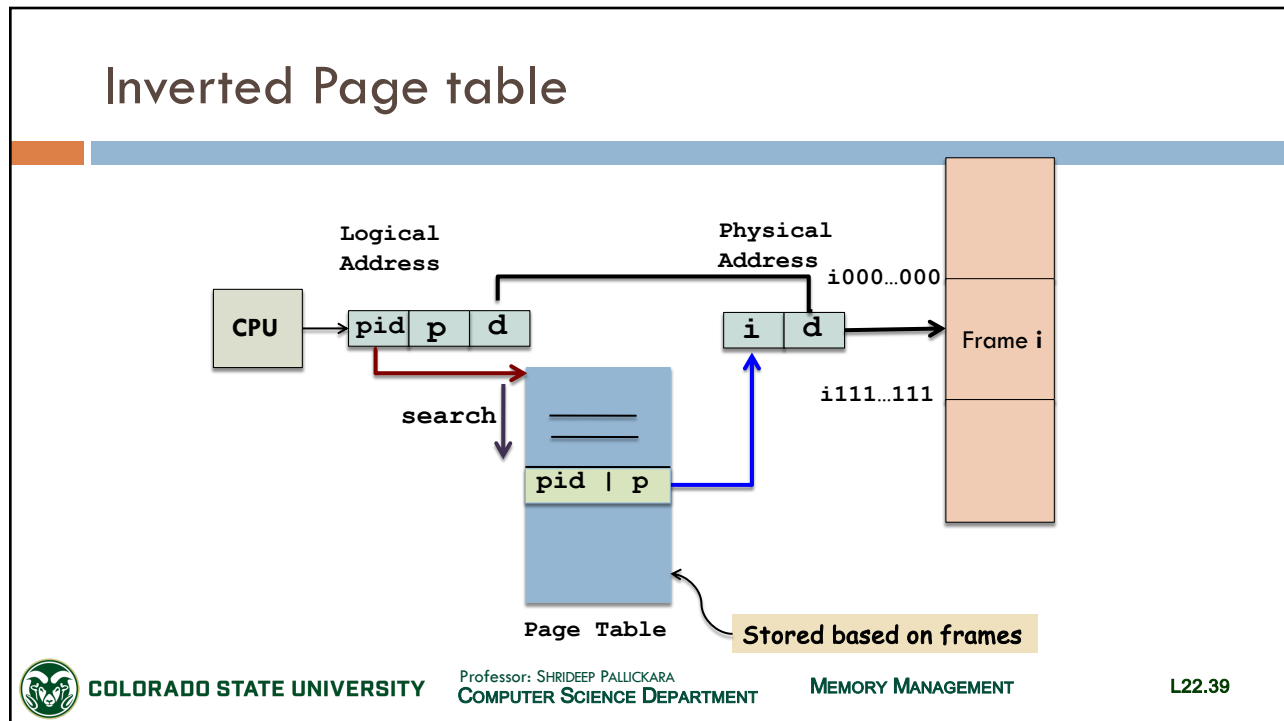
37

## 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**)

 COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT MEMORY MANAGEMENT L22.38

38



39

- ## 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!
- COLORADO STATE UNIVERSITY    Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT    MEMORY MANAGEMENT    L22.40

40

## 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



41

## PAGING IN REAL-WORLD SYSTEMS



42

## 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 [Sufficient for 256 TB]
  - Page sizes: 4 KB, 2 MB, and 1 GB
  - 4-level hierarchical paging



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.43

43

## A typical paging scheme in the x86-64



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.44

44

## Optimization to eliminate levels in the x86-64

- High-end servers routinely have 2 TB RAM
- With 48-bit addressing and 4-level page tables we can have some optimizations
- Each physical frame on the x86 is 4 KB
- Each page in the 4<sup>th</sup> level page table maps 2 MB
  - ▣ If the entire 2MB covered by that page table is allocated contiguously in physical memory?
    - Page table entry one layer up can be marked to point directly to this region instead of page table
- Also improves TLB efficiency



45

## ARM architectures

- iPhone and Android systems use this
  - 32-bit ARM
    - ▣ 4 KB and 16 KB pages
    - ▣ 1 MB and 16 MB pages
      - Termed sections
- ↙ 2-level paging
- ↙ 1-level paging

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



46

## SEGMENTATION WITH PAGING

COMPUTER SCIENCE DEPARTMENT



COLORADO STATE UNIVERSITY

47

## 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 (4GB)
  - ▣ Few programs need more than 1000 segments, but many programs need large segments



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.48

48



## Segmentation with Paging

- 32-bit x86
  - Virtual address space within a segment has a 2-level page table
    - First 10-bits top-level page table, next 10-bits second-level page-table, final 12-bits are the offsets within the page
- 64-bit x86
  - 48-bits of virtual addresses within a segment
  - 4-level page table
    - Includes optimizations to eliminate one or two levels of the page table



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.49

49

**VIRTUAL MEMORY**

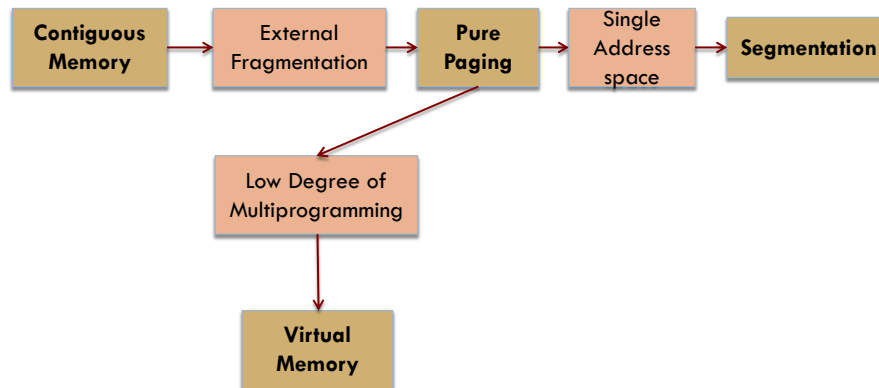
COMPUTER SCIENCE DEPARTMENT



COLORADO STATE UNIVERSITY

50

## How we got here ...



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.51

51

## 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**



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.52

52

## 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

- 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



COLORADO STATE UNIVERSITY

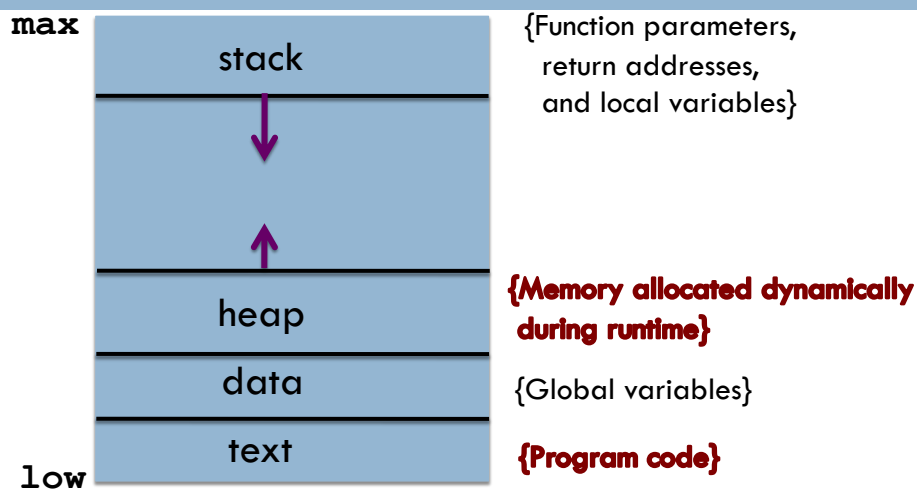
Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.55

55

## Logical view of a process in memory



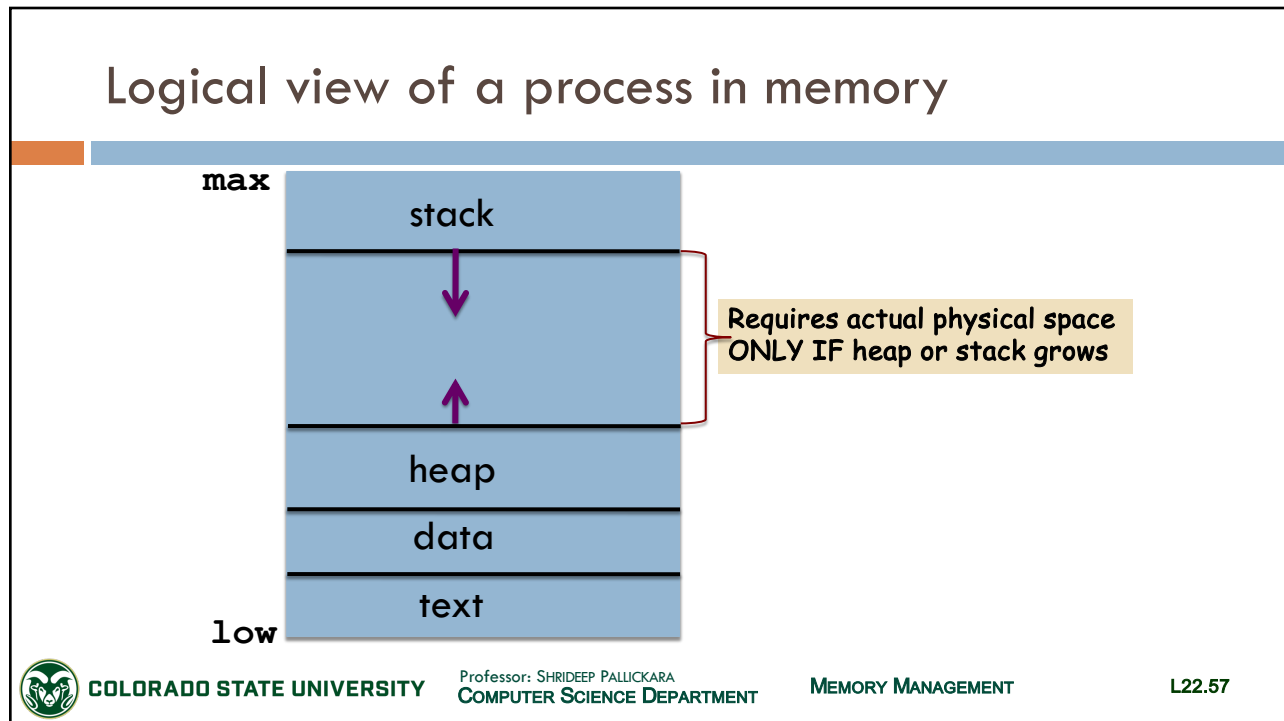
COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.56

56



57

## Sparse address spaces

- Virtual address spaces with holes
- Harnessed by
  - ▣ Heap or stack segments
  - ▣ Dynamically linked libraries

The footer includes the Colorado State University logo, the text 'COLORADO STATE UNIVERSITY', 'Professor: SHRIDEEP PALICKARA', 'COMPUTER SCIENCE DEPARTMENT', 'MEMORY MANAGEMENT', and 'L22.58'.

58

## DEMAND PAGING

COMPUTER SCIENCE DEPARTMENT



COLORADO STATE UNIVERSITY

59

## Loading an executable program into memory

- What if we load the entire program?
  - ▣ We may not need the entire program
  
- Load pages *only* when they are needed
  - ▣ **Demand Paging**



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

MEMORY MANAGEMENT

L22.60

60

## Differences between the swapper and pager

- Swapper
  - ▣ Swaps the *entire program* into memory
  
- **Pager**
  - ▣ Lazy swapper
  - ▣ Never swap a page into memory *unless* it is actually *needed*



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

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9<sup>th</sup> edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330. [Chapter 8]*
- *Andrew S Tanenbaum. Modern Operating Systems. 4<sup>th</sup> 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]*

