Operating Systems

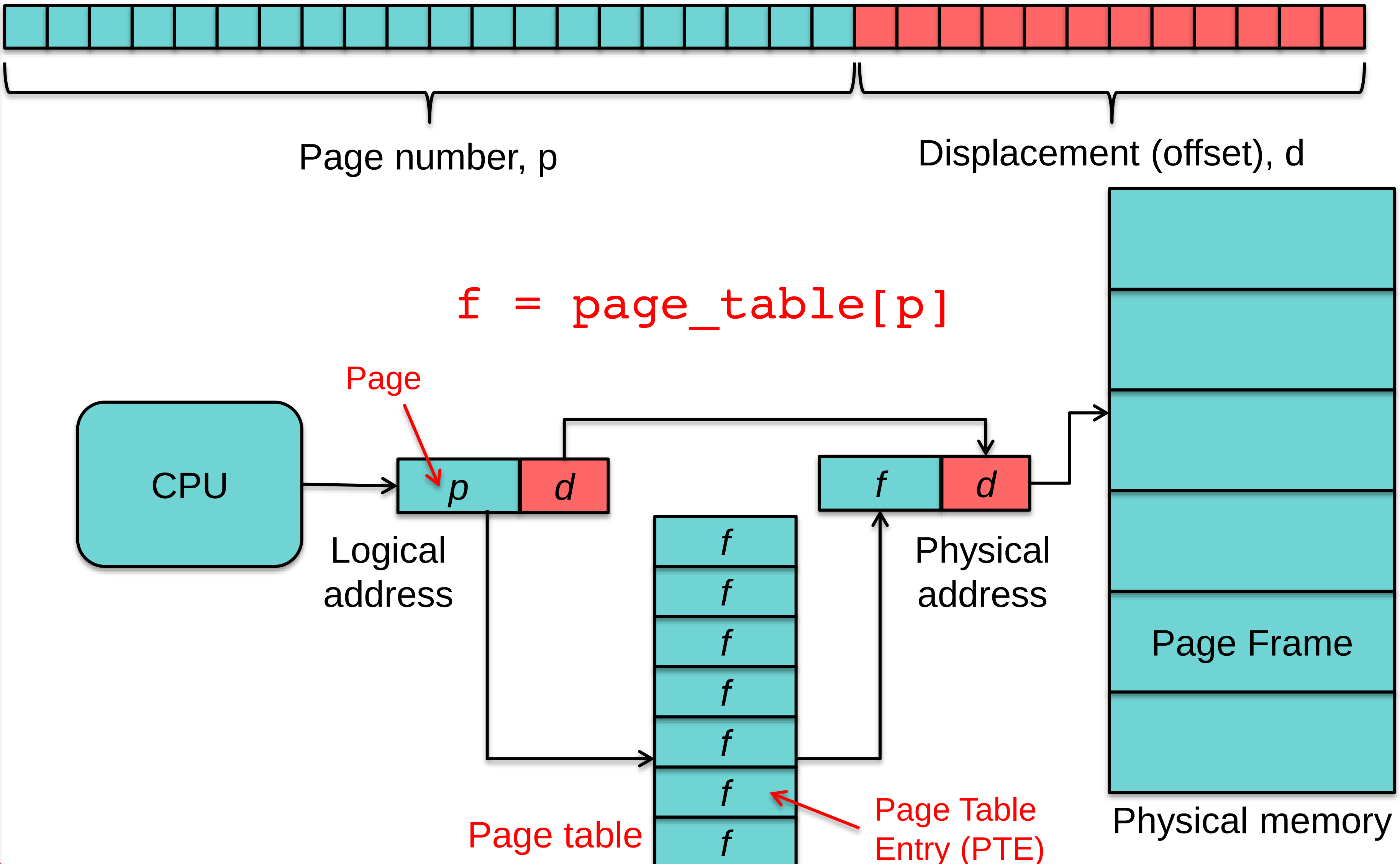
10. Memory Management – Part 2
Paging

Paul Krzyzanowski

Rutgers University

Spring 2015

Page translation



Page table

- One page table per process
  - Contains page table entries (PTEs)

- Each PTE contains
  - Corresponding page frame # for a page #
  - Permissions
    - Permissions (read-only, read-write, execute-only, privileged access only…)
  - Access flags
    - *Valid?* Is the page mapped?
    - *Modified?*
    - *Referenced?*

- Page table is selected by setting a page table base register with the address of the table

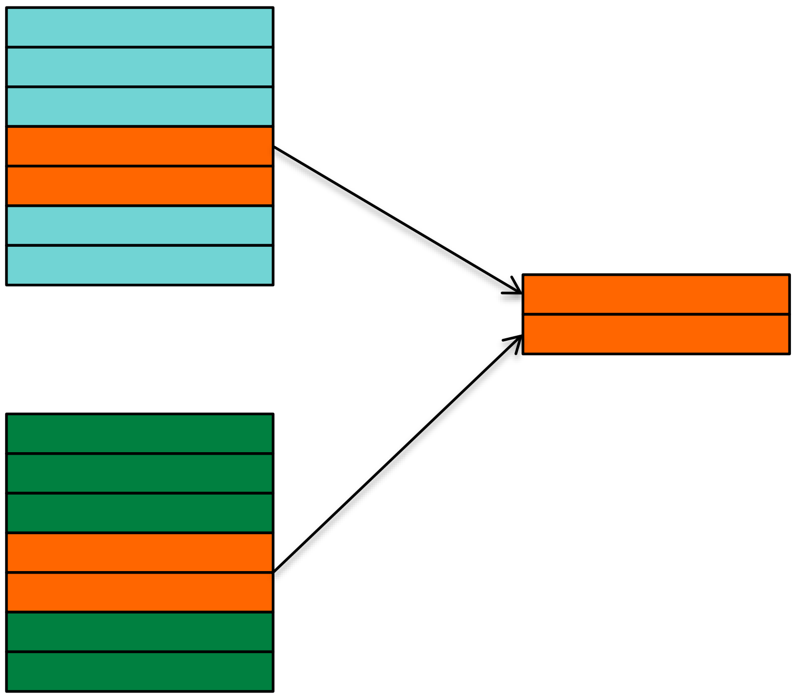
Page-Based Virtual Memory Benefits

- Allow discontiguous allocation
  - Simplify memory management for multiprogramming
  - MMU gives the illusion of contiguous allocation of memory

- Process can get memory anywhere in the address space
  - Allow a process to feel that it has more memory than it really has
  - Process can have greater address space than system memory

- Enforce memory Protection
  - Each process' address space is separate from others
  - MMU allows pages to be protected:
    - Writing, execution, kernel vs. user access

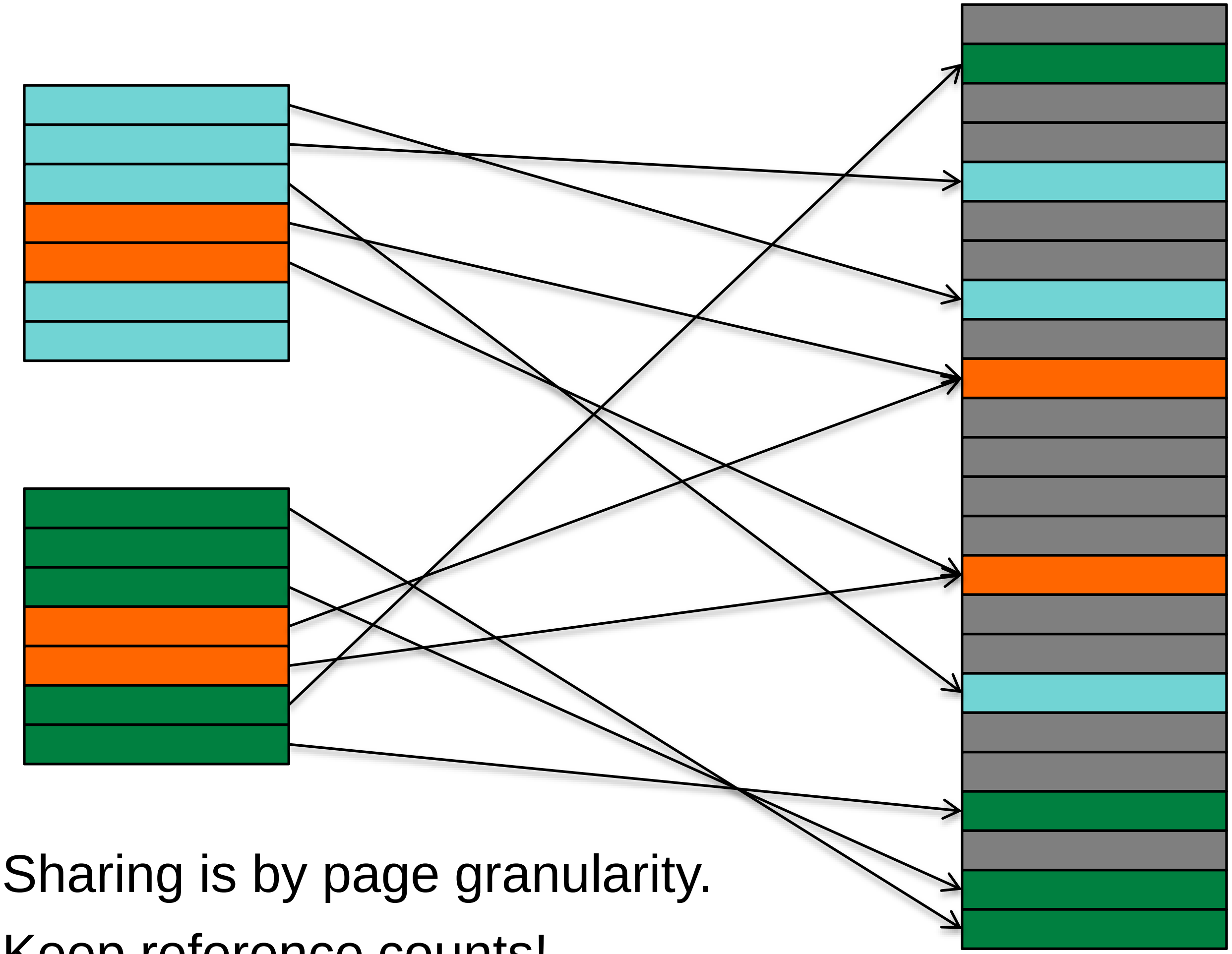
Virtual memory makes memory sharing easy



Shared library or
Shared memory

Sharing is by page granularity

Virtual memory makes memory sharing easy



Sharing is by page granularity.

Keep reference counts!

Copy on write

- Share until a page gets modified
- Example: fork()
  - Set all pages to read-only
  - Trap on write
  - If legitimate write
    - Allocate a new page and copy contents from the original

Demand Paging

Executing a program

- Allocate memory + stack
- Load the entire program from memory
(including any dynamically linked libraries)
- Then execute the loaded program

Executing a program

- Allocate memory + stack
- Load the entire program from memory (including any dynamically linked libraries)
- Then execute the loaded program

This can take a while!

There's a better way…

Demand Paging

- Load pages into memory only as needed
  - On first access
  - Pages that are never used never get loaded
- Use *valid* bit in page table entry
  - Valid: the page is in memory (“valid” mapping)
  - Invalid: out of bounds access or page is not in memory
    - Have to check the process’ memory map in the PCB to find out
- Invalid memory access generates a *page fault*

Demand Paging: At Process Start

- Open executable file
- Set up memory map (stack & text/data/bss)
  - But don't load anything!
- Load first page & allocate initial stack page
- Run it!

Memory Mapping

- Executable files & libraries must be brought into a process' virtual address space
  - File is *mapped* into the process' memory
  - As pages are referenced, page frames are allocated & pages are loaded into them

- `vm_area_struct`
  - Defines regions of virtual memory
  - Used in setting page table entries
  - Start of VM region, end of region, access rights
- Several of these are created for each mapped image
  - Executable code, initialized data, uninitialized data

Demand Paging: Page Fault Handling

- Eventually the process will access an address without a valid page
  - OS gets a page fault from the MMU

- What happens?
  - Kernel searches a tree structure of memory allocations for the process to see if the faulting address is valid
    - If not valid, send a SEGV signal to the process
  - Is the type of access valid for the page?
    - Send a signal if not
  - We have a valid page but it’s not in memory
    - Go get it from the file!

Page Replacement

- A process can run without having all of its memory allocated
  - It's allocated on demand
- If the
  {address space used by all processes + OS} ≤ physical memory
  then we're ok
- Otherwise:
  - Make room: discard or store a page onto the disk
  - If the page came from a file & was not modified
    - Discard … we can always get it
  - If the page is dirty, it must be saved in a page file (aka swap file)
  - Page file: a file (or disk partition) that holds excess pages
    - Windows: pagefile.sys
    - Linux: swap partition or swap file
    - OS X: multiple swap files in `/private/var/vm/swapfile*`

Page replacement

We need a good replacement policy for good performance

FIFO Replacement

First In, First Out

- Good
  - May get rid of initialization code or other code that’s no longer used
- Bad
  - May get rid of a page holding frequently used global variables

Least Recently Used (LRU)

- Timestamp a page when it is accessed
- When we need to remove a page, search for the one with the oldest timestamp
- Nice algorithm but…
  - Timestamping is a pain – we can’t do it with the MMU!

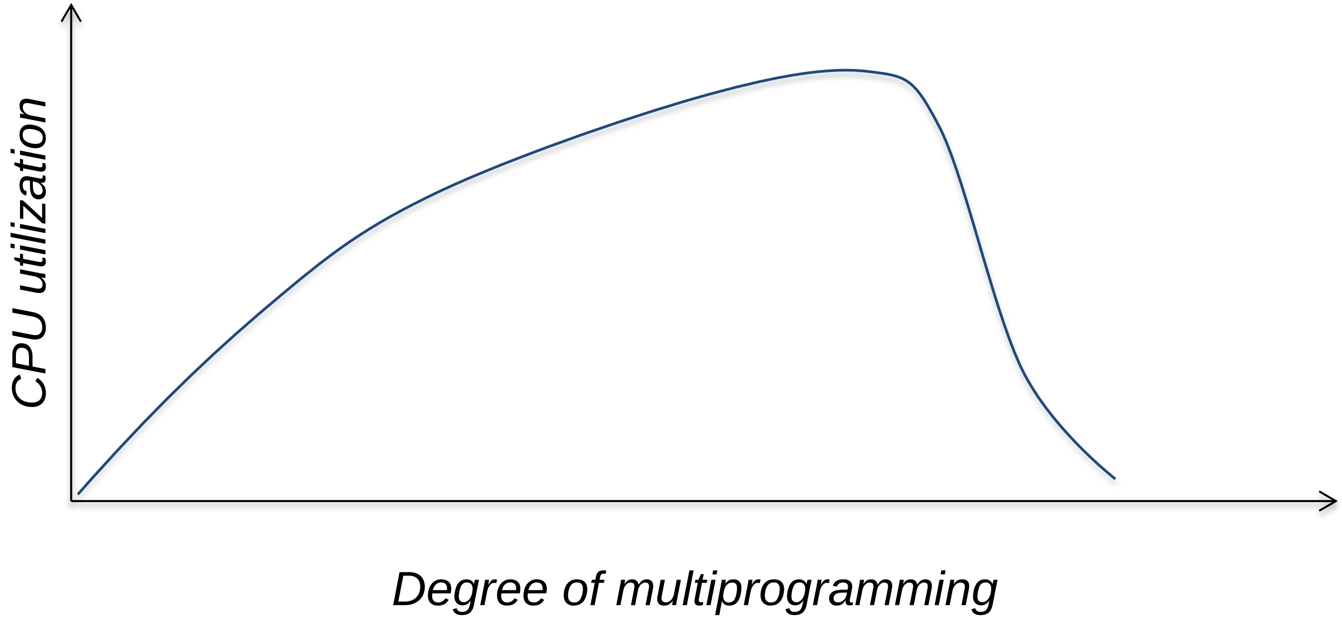
Not Frequently Used Replacement

Approximate LRU behavior

- Each PTE has a reference bit
- Keep a counter for each page frame
- At each clock interrupt:
  – Add the reference bit of each frame to its counter
  – Clear reference bit
- To evict a page, choose the frame with the lowest counter
- Problem
  – No sense of time: a page that was used a lot a long time ago may still have a high count
  – Updating counters is expensive

Thrashing

- Locality
  - Process migrates from one working set to another
- Thrashing
  - Occurs when sum of all working sets > total memory
  - There is not enough room to hold each process' working set



The End