## Lecture Topics: 11/24

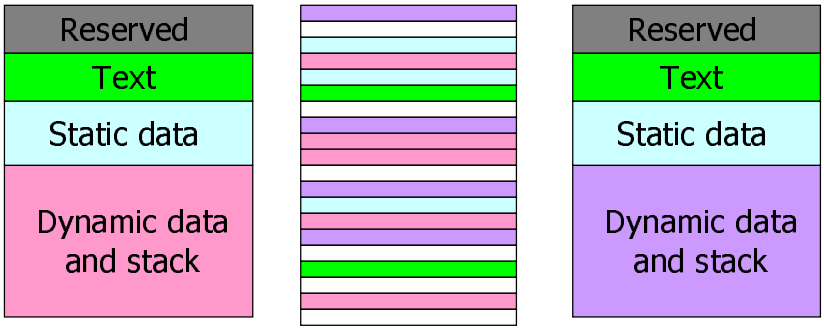
- Sharing Pages
- Demand Paging (and alternative)
- Page Replacement
  - optimal algorithm
  - implementable algorithms

## Redundancy in Memory

- A lot of times, we run more than one instance of the same program
  - more than one editor session
  - more than one server process
  - more than one of *anything* after a fork()
- The virtual address spaces will contain redundant data
- Physical memory may contain redundant data

# Shared Pages

- With paged virtual memory, an easy fix
- Just make both page tables point to the same page frame for shared data




# The Life Cycle of Process VM

- When a process starts, its page table contains no valid mappings
  - This is after exec() in unix
- The contents of virtual memory are known, but not resident in physical memory
- Data only comes from two places
  - Disk (program image, static data)
  - Zero (stack pages, heap)

## Demand Paging

- When the new process begins to run
  - The PC is set to the beginning of the first text page
  - Try to fetch the first instruction
  - The instruction's virtual address is sent to the MMU
  - No VPN->PPN mapping is found in the TLB
  - The page table entry is marked invalid
  - Trap to the OS - page fault!

## Page Fault Handling

- The OS must deal with invalid page table entries:
  - copy the code page from disk to memory
  - update the page table entry and mark it valid
  - insert the mapping into the TLB
  - return to the application

## The Process Keeps Running...

- As the process runs, it *touches* new pages
- Each new page is faulted in by the OS
- If it is a code or static data page, it comes from the program executable
- If it is a stack or heap page, it is filled with zeros

- Can also map a file into memory

## Alternative to Demand Paging

- Unsuprising that the first page faulted is always the first code page
- Why not just copy the executable into memory when you start the process?
  - Or at least a little bit of it
- Take it another step
  - Any time you have a good idea that a page will be needed soon, go ahead and get it
- This is called *prefetching*

## Page Replacement

- Eventually, memory will fill up
- Some page will have to be *evicted* to free up a page frame for another page
- We have to choose the *victim*
  - Should we choose globally or from this process alone?
  - Once the set of potential victims is selected, which page makes the best victim?

## Choosing a Victim

- The theoretical answer is clear:
  - The best victim is the page not needed for the longest time in the future
- One problem: we (usually) don't know the future
- We can use locality to get some ideas about the future, as in higher-level caches
  - this motivates LRU (least recently used)

## LRU and Approximations

- In higher level caches, we avoided LRU because it was too difficult/expensive to do in hardware
- Page replacement is executed in software; we have more resources
- Still, it's usually better just to approximate LRU
  - The results are often just as good
  - Less computation in the *critical path*
  - Too hard to get accurate "U" information

## FIFO with Second Chance

- FIFO = First in, First out
  - Not at all a good LRU approximation
  - No knowledge of use
- In FIFO with second chance, make two queues:
  - FIFO queue big, 2nd chance queue small
  - Pages move from FIFO queue to 2nd chance queue
  - Pages on 2nd chance queue move back to FIFO queue if used, otherwise evicted