Administrivia

• Lab time
  ♦ This week: Thu 4pm, Sat 2pm
  ♦ Next week: Tue, Wed

• At Washington University in St. Louis on Sun & Mon
  ♦ No Monday office hours
  ♦ Office hours Tue 3:30-4:30pm instead
Memory Management

Final lecture on memory management:

• Goals of memory management
  ♦ To provide a convenient abstraction for programming
  ♦ To allocate scarce memory resources among competing processes to maximize performance with minimal overhead

• Mechanisms
  ♦ Physical and virtual addressing (1)
  ♦ Techniques: Partitioning, paging, segmentation (1)
  ♦ Page table management, TLBs, VM tricks (2)

• Policies
  ♦ Page replacement algorithms (3)
Lecture Overview

- Review paging and page replacement
- Survey page replacement algorithms
- Discuss local vs. global replacement
- Discuss thrashing
Locality

- All paging schemes depend on locality
  - Processes reference pages in localized patterns
- Temporal locality
  - Locations referenced recently likely to be referenced again
- Spatial locality
  - Locations near recently referenced locations are likely to be referenced soon
- Although the cost of paging is high, if it is infrequent enough it is acceptable
  - Processes usually exhibit both kinds of locality during their execution, making paging practical
Demand Paging (OS)

• Recall demand paging from the OS perspective:
  ♦ Pages are evicted to disk when memory is full
  ♦ Pages loaded from disk when referenced again
  ♦ References to evicted pages cause a TLB miss
    » PTE was invalid, causes fault
  ♦ OS allocates a page frame, reads page from disk
  ♦ When I/O completes, the OS fills in PTE, marks it valid, and
    restarts faulting process

• Dirty vs. clean pages
  ♦ Actually, only dirty pages (modified) need to be written to disk
  ♦ Clean pages do not – but you need to know where on disk to
    read them from again
Demand Paging (Process)

- Demand paging is also used when a process first starts up
- When a process is created, it has
  - A brand new page table with all valid bits off
  - No pages in physical memory
- When the process starts executing
  - Instructions fault on code and data pages
  - Faulting stops when all necessary code and data pages are in memory
  - Only code and data needed by a process needs to be loaded
  - This, of course, changes over time…
Page Replacement

• When a page fault occurs, the OS loads the faulted page from disk into a page frame of memory
• At some point, the process has used all of the page frames it is allowed to use
  ♦ This is likely (much) less than all of available memory
• When this happens, the OS must replace a page for each page faulted in
  ♦ It must evict a page to free up a page frame
• The page replacement algorithm determines how this is done
  ♦ And they come in all shapes and sizes
Evicting the Best Page

• The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove
• The best page to evict is the one never touched again
  ♦ Will never fault on it
• Never is a long time, so picking the page closest to “never” is the next best thing
  ♦ Evicting the page that won’t be used for the longest period of time minimizes the number of page faults
  ♦ Proved by Belady
• We’re now going to survey various replacement algorithms, starting with Belady’s
Belady’s Algorithm

• Belady’s algorithm is known as the optimal page replacement algorithm because it has the lowest fault rate for any page reference stream
  ♦ Idea: Replace the page that will not be used for the longest time in the future
  ♦ Problem: Have to predict the future

• Why is Belady’s useful then? Use it as a yardstick
  ♦ Compare implementations of page replacement algorithms with the optimal to gauge room for improvement
  ♦ If optimal is not much better, then algorithm is pretty good
  ♦ If optimal is much better, then algorithm could use some work
    » Random replacement is often the lower bound
First-In First-Out (FIFO)

- FIFO is an obvious algorithm and simple to implement
  - Maintain a list of pages in order in which they were paged in
  - On replacement, evict the one brought in longest time ago

- Why might this be good?
  - Maybe the one brought in the longest ago is not being used

- Why might this be bad?
  - Then again, maybe it’s not
  - We don’t have any info to say one way or the other

- FIFO suffers from “Belady’s Anomaly”
  - The fault rate might actually increase when the algorithm is given more memory (very bad)
Least Recently Used (LRU)

• LRU uses reference information to make a more informed replacement decision
  ♦ Idea: We can’t predict the future, but we can make a guess based upon past experience
  ♦ On replacement, evict the page that has not been used for the longest time in the past (Belady’s: future)
  ♦ When does LRU do well? When does LRU do poorly?

• Implementation
  ♦ To be perfect, need to time stamp every reference (or maintain a stack) – much too costly
  ♦ So we need to approximate it
Approximating LRU

• LRU approximations use the PTE reference bit
  ♦ Keep a counter for each page
  ♦ At regular intervals, for every page do:
    » If ref bit = 0, increment counter
    » If ref bit = 1, zero the counter
    » Zero the reference bit
  ♦ The counter will contain the number of intervals since the last reference to the page
  ♦ The page with the largest counter is the least recently used

• Some architectures don’t have a reference bit
  ♦ Can simulate reference bit using the valid bit to induce faults
  ♦ What happens when we make a page invalid?
LRU Clock
(Not Recently Used)

• Not Recently Used (NRU) – Used by Unix
  ♦ Replace page that is “old enough”
  ♦ Arrange all of physical page frames in a big circle (clock)
  ♦ A clock hand is used to select a good LRU candidate
    » Sweep through the pages in circular order like a clock
    » If the ref bit is off, it hasn’t been used recently
      ▪ What is the minimum “age” if ref bit is off?
    » If the ref bit is on, turn it off and go to next page
  ♦ Arm moves quickly when pages are needed
  ♦ Low overhead when plenty of memory
  ♦ If memory is large, “accuracy” of information degrades
    » What does it degrade to?
    » One fix: use two hands (leading erase hand, trailing select hand)
Example: gcc Page Replace

![Graph showing page replacement strategies]

- Optimal
- LRU
- Clock
- FIFO
- LIFO
- LFU
- Random
Example: Belady’s Anomaly
Fixed vs. Variable Space

• In a multiprogramming system, we need a way to allocate memory to competing processes
• Problem: How to determine how much memory to give to each process?
  ♦ Fixed space algorithms
    » Each process is given a limit of pages it can use
    » When it reaches the limit, it replaces from its own pages
    » Local replacement
      ▪ Some processes may do well while others suffer
  ♦ Variable space algorithms
    » Process’ set of pages grows and shrinks dynamically
    » Global replacement
      ▪ One process can ruin it for the rest
Working Set Model

- A working set of a process is used to model the dynamic locality of its memory usage
  - Defined by Peter Denning in 60s
- Definition
  - \( WS(t,w) = \{\text{pages } P \text{ such that } P \text{ was referenced in the time interval } (t, t-w)\} \)
  - \( t \) – time, \( w \) – working set window (measured in page refs)
- A page is in the working set (WS) only if it was referenced in the last \( w \) references
Working Set Size

- The working set size is the number of unique pages in the working set
  - The number of pages referenced in the interval $(t, t-w)$
- The working set size changes with program locality
  - During periods of poor locality, you reference more pages
  - Within that period of time, the working set size is larger
- Intuitively, want the working set to be the set of pages a process needs in memory to prevent heavy faulting
  - Each process has a parameter $w$ that determines a working set with few faults
  - Denning: Don’t run a process unless working set is in memory
Example: gcc Working Set

[Graph showing the variation of Window Size (Pages) over Time (Page Faults)]

WS(0,T)
Working Set Problems

• Problems
  ♦ How do we determine w?
  ♦ How do we know when the working set changes?

• Too hard to answer
  ♦ So, working set is not used in practice as a page replacement algorithm

• However, it is still used as an abstraction
  ♦ The intuition is still valid
  ♦ When people ask, “How much memory does Firefox need?”, they are in effect asking for the size of Firefox’s working set
Page Fault Frequency (PFF)

- Page Fault Frequency (PFF) is a variable space algorithm that uses a more ad-hoc approach
  - Monitor the fault rate for each process
  - If the fault rate is above a high threshold, give it more memory
    » So that it faults less
    » But not always (FIFO, Belady’s Anomaly)
  - If the fault rate is below a low threshold, take away memory
    » Should fault more
    » But not always

- Hard to use PFF to distinguish between changes in locality and changes in size of working set
Thrashing

- Page replacement algorithms avoid thrashing
  - When most of the time is spent by the OS in paging data back and forth from disk
  - Little time spent doing useful work (making progress)
  - In this situation, the system is overcommitted
    - No idea which pages should be in memory to reduce faults
    - Could just be that there isn’t enough physical memory for all of the processes in the system
    - Ex: Running Windows95 with 4 MB of memory…
  - Possible solutions
    - Swapping – write out all pages of a process
    - Buy more memory
Summary

• Page replacement algorithms
  ♦ Belady’s – optimal replacement (minimum # of faults)
  ♦ FIFO – replace page loaded furthest in past
  ♦ LRU – replace page referenced furthest in past
    » Approximate using PTE reference bit
  ♦ LRU Clock – replace page that is “old enough”
  ♦ Working Set – keep the set of pages in memory that has minimal fault rate (the “working set”)
  ♦ Page Fault Frequency – grow/shrink page set as a function of fault rate

• Multiprogramming
  ♦ Should a process replace its own page, or that of another?
Next time...

• Read Chapters 37, 39, 40