Recap: cache

- Well-written programs exhibit a property called locality, this is the foundation of caching

- Basic concepts of cache
  - Cache hit
  - Cache miss

- Cache replacement policy is important
Today

- Virtual memory
  - Memory as cache

- Page/cache replacement policy
**VM as a Tool for Caching**

- *Virtual memory* is an array of $N$ contiguous bytes stored on disk.
- The contents of the array on disk are cached in *physical memory* (*DRAM cache*)
  - These cache blocks are called *pages* (size is $P = 2^p$ bytes)

![Diagram of virtual memory and physical memory](attachment:image.png)

- Virtual pages (VPs) stored on disk
- Physical pages (PPs) cached in DRAM
Page Table Setup

- Valid PTEs map virtual pages to physical pages.
- Invalid PTEs map virtual pages to disk blocks.
Page hit: reference to VM word that is in physical memory (DRAM cache hit)
**Page Fault (Cache miss)**

- **Page fault**: Reference to VM word that is not in physical memory (DRAM cache miss)

![Diagram showing page fault and memory layout](image)

- Virtual address
- Physical page number or disk address
- Valid
- Memory resident page table (DRAM)
- Physical memory (DRAM)
- Virtual memory (disk)
Handling Page Fault (1)

- Page miss causes page fault (an exception)
Handling Page Fault (2)

- Page fault handler selects a victim to be evicted (here VP 4)
Handling Page Fault (3)

- Evict the content of VP4
Handling Page Fault (4)

- Update page table

![Diagram of page table and memory layout](image)
Handling Page Fault (5)

- Load content of VP3 to DRAM

![Diagram of memory resident page table and virtual memory]

- Memory resident page table (DRAM)
- Physical memory (DRAM)
- Virtual memory (disk)
Handling Page Fault (6)

- Update page table
Handling Page Fault (7)

- Restart the instruction: page hit!
Today

- Virtual memory
  - Memory as cache

- Page/cache replacement policy
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.
  - Written back only if it is has been modified (i.e., “dirty”)!
Page replacement policy

- What we discussed so far (page faults, swap, page table structures, etc…) is mechanisms

- Page replacement policy: determine which page to remove when we need a victim

- Does it matter?
  - Yes! Page faults are super expensive
  - Getting the number down, can improve the performance of the system significantly
Considerations

- Page replacement support has to be simple during memory accesses
  - They happen all the time, we cannot make that part slow

- But it can be complicated/expensive when a page fault occurs – why?
  - Reason 1: if we are successful, this will be rare
  - Reason 2: when it happens we are paying the cost of I/O
    » I/O is very slow: can afford to do some extra computation
    » Worth it if we can save some future page faults

- What makes a good page replacement policy?
Locality to the Rescue

- Recall that virtual memory works because of locality
  - Temporal and spatial
  - Work at different scales: for cache, at a line level, for VM, at page level, and even at larger scales

- All paging schemes depend on locality
  - What happens if a program does not have locality?
  - High cost of paging is acceptable, if infrequent
  - Processes usually reference pages in localized patterns, making paging practical
Evicting the Best Page

- Goal is to reduce the page fault rate
- 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
  - Idea: Replace the page that will not be used for the longest time in the future
  - Optimal? How would you show?
  - Problem: Have to predict the future

- Why is Belady’s useful then?
  - Use it as a yardstick/upper bound
  - 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
  - What’s a good lower bound?
    - 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
## 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)
LRU Clock

P1: 1
P2: 1
P3: 1
P4: 0
P5: 1
P6: 0
P7: 0
P8: 0

P1: 0
P2: 0
P3: 0
P4: 0
P5: 1
P6: 0
P7: 0
P8: 1
Example: gcc Page Replace

![Graph showing page replacement policies]

- Optimal
- LRU
- Clock
- FIFO
- LIFO
- LFU
- Random

CS 153 – Lecture 20 – Virtual Memory
Example: Belady’s Anomaly

- Number of Page Frames
- Number of Page Faults (log)

FIFO
Other ideas

- Victim buffer
  - Add a buffer (death row!) we put a page on when we decide to replace it
  - Buffer is FIFO
  - If you get accessed while on death row – clemency!
  - If you are the oldest page on death row – replacement!
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
  - $WS(t,w) = \{\text{set of pages } P, \text{ such that every page in } P \text{ was referenced in the time interval } (t, t-w)\}$
  - $t$ – 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 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
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
  - No 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...

- Disk Drives
- Preparation
  - Read Module 35, 36, 37, 44