Lec 16

Page Replacement

Demand Paging Steps

1. TLB miss
2. Hardware page table walk
3. Page fault (invalid in table)
4. Trap to kernel
5. Locate page on disk
6. Allocate frame
   - evict if necessary
7. Initiate disk read into frame
8. Interrupt when read complete
9. Mark page valid in table
10. Resume process at faulting instruction
11. TLB miss
12. Hardware page table walk
13. Page valid, execute instruction.

Locating a page on disk

- option 1: Store a disk sector in the virtual addressing data structure

- option 2: Use the file system
  - for each region of memory, associate a file
    - code: ELF file
    - data/heap/stack: temp file

Allocating a frame

- select frame to evict
- find all pages that map to that frame
  - more than 1, if frame is shared! (use coremap)
- set all such page table entries to invalid
- Remove any TLB entries (on any core)
  - Warning: would be wrong to do this before setting PTEs to invalid!
- Write any changes to old frame back to disk
  - Warning: would be wrong to do this before TLB shootdown!

Has page been modified/recently used?

- x86 tracks:
  - (D) Dirty bit — has it been modified
    set by hardware on store
  - (A) Access bit — has it been accessed
    set by hardware on every TLB miss
- OS Kernel periodically resets these bits
  - D bit: flush changes to disk
  - A bit: just set to 0

Cache Replacement Policy

- On a cache miss, how do we choose which entry to replace?

- goals:
  - reduce cache misses
  - improve expected performance
  - make very bad performance unlikely
  - handle large number of entries ($\sim 2^{25} = 10^7 - 10^8$)

Simple policies

Random: - choose a random entry to evict
- unlikely to be pessimal!
- simple to implement

FIFO: - replace oldest page
- When will this work poorly?

Linear Scan Workload

Accesses

A B C D E A B C D E A B C D E

Frames

1

2

3

4

Optimal policy: MIN

- replace entry that will not be used for longest time

- proof of optimality is similar to SJF
  - any alternate policy would have more misses

- requires knowledge of future
  - can only implement approximations

Approximating MIN

Least Recently Used (LRU)

- replace entry not used longest time in past
- approximates MIN by using the past to predict the future

Least Frequently Used (LFU)

- replace entry used the least often
- uses popularity to predict the future (approximately)
- better than LRU if usage pattern is based on popularity
  (eg Zipf distribution)
- real systems can combine LRU and LFU

Temporal Locality Workload

Accesses

A B A C B D A D E D A E B A C

Frames

1

2

3

4

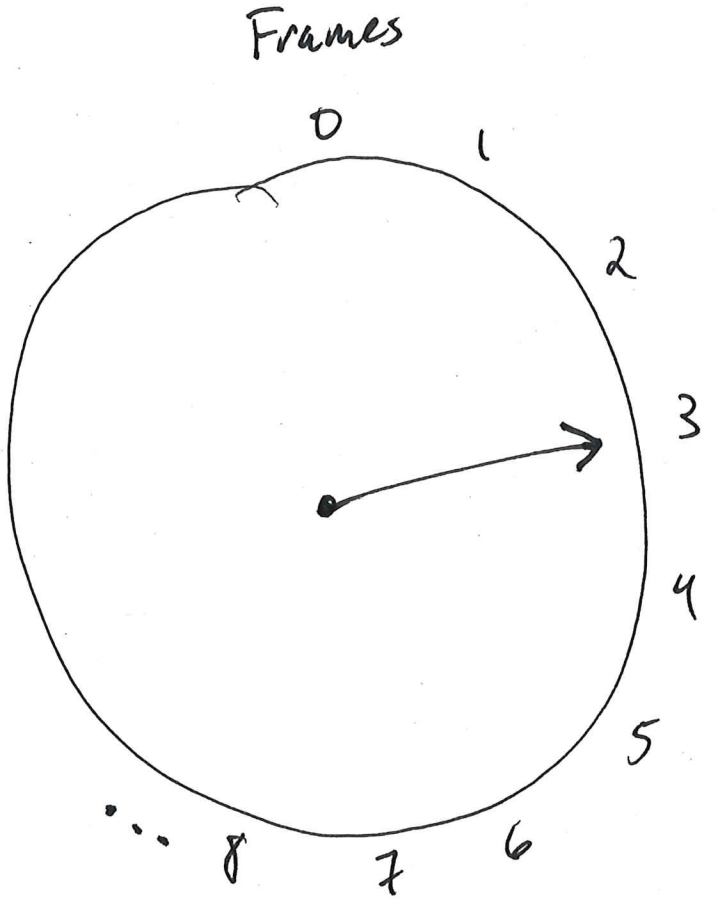
Implementing LRU

- Would need to know when page was accessed
  - but hardware only gives us A bit

- How can we use A bit to approximate time?
  - how good of an approximation do we need
    - If miss cost is low, any approx will work
    - If miss cost is high but # pages is large
      any not recently used page will work
    - If miss cost is high but # pages is small,
      then need to be accurate

Clock Algorithm

- hardware sets A bit
- periodically OS sweeps through frames
  - if page is unused since last sweep reclaim it (flush if dirty)
  - if page is used, reset A bit



16.12