CSE 120
Principles of Operating Systems

Spring 2023

Lecture 17: Page Replacement
Yiying Zhang
Announcements

• Project 2 due this Friday, no more extension!

• Project 3 announced
  ♦ You should start as soon as you can if you want to work on it
  ♦ You can change team for PR3
  ♦ Fair warning that PR3 will be time consuming!

• Midterm regrading requests due today
Memory Management

The real 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
  - Techniques: Partitioning, paging, segmentation
  - Page table management, TLBs
  - Memory allocation

- **Policies**
  - Page replacement algorithms
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 that it is acceptable
  ♦ Processes usually exhibit both kinds of locality during their execution, making paging practical
The BIG picture: Running at Memory Capacity

- Expect to run with all phy. pages in use
- Every demand paging request (e.g., swap-in, new phys page allocation) requires an eviction
- Goal of page replacement
  - Maximize hit rate → kick out the page that’s least useful
- Challenge: how do we determine utility?
  - Kick out pages that aren’t likely to be used again

- Page replacement is a difficult policy problem
Performance metric for page replacement policies

• Give a sequence of memory accesses, minimize the # of page faults
  ♦ Similar to cache miss rate
  ♦ What about hit latency and miss latency?
• 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
What makes finding the least useful page hard?

- Don’t know future!

- Past behavior is a good indication of future behavior! (e.g. LRU)
  » temporal locality \(\rightarrow\) kick out pages that have not been used recently

- Perfect (past) reference stream hard to get
  ♦ Every memory access would need bookkeeping
  ♦ Is this feasible (in software? In hardware?)

- Minimize overhead
  ♦ If no memory pressure, ideally no bookkeeping
  ♦ In other words, make the common case fast (page hit)

\(\Rightarrow\) Get imperfect information, while guaranteeing foreground perf
  ♦ What is minimum hardware support that need to added?
What can we do without extra hardware support?
First-In-First-Out (FIFO)

- Algorithm
  - 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
  - Low-overhead implementation

- Cons
  - No frequency/no recency → may replace the heavily used pages

- FIFO suffers from “Belady’s Anomaly”
  - The fault rate might actually increase when the algorithm is given more memory (very bad)
Predicting future based on past

• “Principle of locality”
  ♦ Recency:
    » Page recently used are likely to be used again in the near future
  ♦ Frequency:
    » Pages frequently used before are likely to be used frequently again in the near future

• Is this temporal or spatial locality?
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
  - 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
Exploiting locality needs some hardware support

• Reference bit
  ♦ A hardware bit that is set whenever the page is referenced (read or written)

• Why not in software?
# x86 Page Table Entry

<table>
<thead>
<tr>
<th>Page frame number</th>
<th>U</th>
<th>P</th>
<th>Cw</th>
<th>Gl</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>Cd</th>
<th>Wt</th>
<th>O</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Reserved**
- **Valid (present)**
- **Read/write**
- **Owner (user/kernel)**
- **Write-through**
- **Cache disabled**
- **Accessed (referenced)**
- **Dirty**
- **PDE maps 4MB**
- **Global**
LRU Clock (Not Recently Used)

- Clock algorithm – Used by Unix
- Idea: 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
    - Pick it for page replacement (victim page)
    - What is the minimum “age” if ref bit is off?
  - If the ref bit is on, turn it off and go to next page. (why turn off?)
- Low overhead when plenty of memory
Clock (cont.)

- What happens if all reference bits are 1?

- If memory is large, “accuracy” of information degrades
  - What does it degrade to?

- What does it suggest if observing clock hand is sweeping very fast?

- What does it suggest if clock hand is sweeping very slow?
We’ve focused on miss rate. What about miss latency?

- Key observation: it is cheaper to pick a “clean” page over a “dirty” page
  - Clean page does not need to be swapped to disk (after it has been previously swapped out)

- Challenge:
  - How to get this info?
Refinement by adding extra hardware support

• **Reference bit**
  ♦ A hardware bit that is set whenever the page is referenced (read or written)

• **Modified bit (dirty bit)**
  ♦ A hardware bit that is set whenever the page is written into
# [lec13] x86 Page Table Entry

<table>
<thead>
<tr>
<th>Page frame number</th>
<th>U</th>
<th>P</th>
<th>Cw</th>
<th>Gl</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>Cd</th>
<th>Wt</th>
<th>O</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

- **Valid (present)**
- **Read/write**
- **Owner (user/kernel)**
- **Write-through**
- **Cache disabled**
- **Accessed (referenced)**
- **Dirty**
- **PDE maps 4MB**
- **Global**

- **Reserved**
Enhanced Clock

- Same as the basic Clock, except that it considers both (reference bit, modified bit)
  - (0,0): neither recently used nor modified (good)
  - (0,1): not recently used but dirty (not as good)
  - (1,0): recently used but clean (not good)
  - (1,1): recently used and dirty (bad)

- On page fault, follow hand to inspect pages:
  - Round 1:
    » If bits are (0,0), take it and stops
    » if bits are (0,1), record 1\textsuperscript{st} instance
    » Clear ref bit for (1,0) and (1,1), if (0,1)/(0,0) not found yet
  - At end of round 1, if (0,1) was found, take it
  - If round 1 does not succeed, try 1 more round
Enhanced Clock

• Pros
  ♦ Avoid write back

• Cons
  ♦ More complicated, worse case scans multiple rounds
Key observation

- Locality in memory references
  - Spatial and temporal

- Want to keep a set of pages in memory that would avoid a lot of page faults
  - “Hot” pages

- Can we formalize it?
Working Set Model

- A working set of a process is used to model the dynamic locality of its memory usage
  - $WS(t,w) = \{\text{all pages referenced in the time interval (t, t-w)}\}$
- Working set size is the number of unique pages in the WS
- The working set size changes with program execution
Working Sets in the Real World

Working set size

transition, stable
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
When there are not enough page frames

• Suppose many processes are making frequent references to 50 pages; there are 49 physical pages.

• Assuming LRU
  ‣ Each time one page is brought in, another page, whose content will soon be referenced, is thrown out.

• What is the average memory access time?

• The system is spending most of its time paging!

• The progress of programs makes it look like “memory access is as slow as disk”, rather than “disk being as fast as memory”.

• Btw, what is the optimal strategy here?
  ‣ MRU
Thrashing

- 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
Thrashing can lead to vicious cycle

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
  - page fault rate goes even higher
What causes thrashing?

- The system does not know it has taken more work than it can handle

- What do humans do when thrashing?
  - Dropping or degrading a course if taking too many than you can handle 😊
Intuitively, what to do about thrashing?

• If a single process’s locality too large for memory, what can OS do?
  ♦ e.g., pin most data (hotter data) in memory, sacrifice the rest

• If the problem arises from the sum of several processes?
  ♦ Figure out how much memory each process needs – “locality”
  ♦ What can we do?
    » Can limit effects of thrashing using local replacement
    » Or, bring a process’ working set before running it
    » Or, wait till there is enough memory for a process’s need
Summary

• Page replacement algorithms
  ♦ FIFO – replace page loaded furthest in past
  ♦ LRU – replace page referenced furthest in past
    » Approximate using PTE reference bit
  ♦ Clock – replace page that is “old enough”
  ♦ Enhanced Clock – pick clean pages first (for lower miss latency)
  ♦ Working Set – keep the set of pages in memory that has minimal fault rate (the “working set”)

• We are finally done with memory management!
Next time...

- Chapters 39, 40, 41
What else can we do to improve miss latency?
Page out on critical path?

• If no free page in physical memory, swap in has to wait till a current page in physical memory is swapped out
  ♦ Page fault handling time = proc. overhead + 2 * I/Os

• There is a chance of swapped out page being referenced soon
Page buffering techniques

OS maintains a pool of free pages

- When a page fault occurs, victim page chosen as before
- But desired page swapped into a free page (a slot in the free page pool) right away before victim page paged out
- OS swaps out dirty victim pages in the background, off the page fault critical path (to make more room in the free page pool)
Page buffering techniques

- Maintaining a list of free physical pages enables another important optimization
- Recall that the page replacement algorithm is a rough approximation of LRU
  - Can certainly make mistakes
  - LRU does not necessarily work well for all program behaviors
- Idea: If a page is on the free list, and it is accessed by a process before being reallocated, rescue it from the free list and give it back to the process
  - Recovers from poor choices made by replacement algorithm