Page Replacement
Address Translation Redux

- We started this topic with the high-level problem of translating virtual addresses into physical addresses.
- We’ve covered all of the pieces:
  - Virtual and physical addresses
  - Virtual pages and physical page frames
  - Page tables and page table entries (PTEs), protection
  - TLBs
  - Demand paging
- Now let’s put it together, bottom to top
The Common Case

- Situation: Process is executing on the CPU, and it issues a read to an address
  - What kind of address is it? Virtual or physical?
- The read goes to the TLB in the MMU
  1. TLB does a lookup using the page number of the address
  2. Common case is that the page number matches, returning a page table entry (PTE) for the mapping for this address
  3. TLB validates that the PTE protection allows reads (in this example)
  4. PTE specifies which physical frame holds the page
  5. MMU combines the physical frame and offset into a physical address
  6. MMU then reads from that physical address, returns value to CPU
- Note: This is all done by the hardware
TLB Misses

- At this point, two other things may happen
  1. TLB does not have a PTE mapping this virtual address
  2. PTE exists, but memory access violates PTE protection bits
- We’ll consider each in turn
Reloading the TLB

- If the TLB does not have mapping, two possibilities:
  1. MMU loads PTE from page table in memory
     - Hardware managed TLB, OS not involved in this step
     - OS has already set up the page tables so that the hardware can access it directly
  2. Trap to the OS
     - Software managed TLB, OS intervenes at this point
     - OS does lookup in page table, loads PTE into TLB
     - OS returns from exception, TLB continues

- A machine will only support one method or the other
- At this point, there is a PTE for the address in the TLB
TLB Misses (2)

Note that:

- Page table lookup (by HW or OS) can cause a recursive fault if page table is paged out

- When TLB has PTE, it restarts translation
  - Common case is that the PTE refers to a valid page in memory
    - These faults are handled quickly, just read PTE from the page table in memory and load into TLB
  - Uncommon case is that TLB faults again on PTE because of PTE protection bits (e.g., page is invalid)
    - Becomes a page fault...
Protection Faults

- PTE can indicate a protection/page fault
  - Read/write/execute – operation not permitted on page
  - Invalid – virtual page not allocated, or page not in physical memory
- TLB traps to the OS (software takes over)
  - R/W/E – OS usually will send fault back up to process, or might be playing games (e.g., copy on write, mapped files)
  - Invalid
    - Virtual page not allocated in address space
      - OS sends fault to process (e.g., segmentation fault)
    - Page not in physical memory
      - OS allocates frame, reads from disk, maps PTE to physical frame
Summary before we move on

- Multi-level page table to save space
- TLB for efficient virtual to physical translation
- Swapping, Demand paging & page fault
- Put everything together for a memory access

**Next:** if physical memory is full, which virtual page’s data do we evict (swap out) to make space for a page loaded from disk (swap partition) that is needed by a process
Page Replacement

1. Find location of page on disk
2. Find a free page frame
   1. If free page frame use it
   2. Otherwise, select a page frame using the page replacement algorithm
   3. Write the selected page to the disk and update any necessary tables
3. Read the requested page from the disk.
4. Restart the user process.
5. It is necessary to be careful of synchronization problems. For example, page faults may occur for pages being paged out.
Issue: Eviction

- Hopefully, kick out a less-useful page
  - Dirty pages require writing, clean pages don’t
    - Hardware has a dirty bit for each page frame indicating this page has been updated or not
  - Where do you write?
    - To “swap space”

- Goal: kick out the page that’s least useful

- Problem: how do you determine utility?
  - Kick out pages that aren’t likely to be used again
  - Heuristic: temporal locality exists
Page Replacement Strategies

- **The Principle of Optimality**
  - Replace the page that will not be used again the farthest time in the future.
- **Random page replacement**
  - Choose a page randomly
- **FIFO - First in First Out**
  - Replace the page that has been in primary memory the longest
- **LRU - Least Recently Used**
  - Replace the page that has not been used for the longest time
- **LFU - Least Frequently Used**
  - Replace the page that is used least often
- **NRU - Not Recently Used**
  - An approximation to LRU.
- **Working Set**
  - Keep in memory those pages that the process is actively using.
Belady’s Algorithm

- Known as the optimal page replacement algorithm because it has the lowest fault rate for any page reference stream/sequence
  - 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 your algorithm is pretty good
  - If optimal is much better, then algorithm could use some work
    - Random replacement is often the lower bound
Optimal Example

12 references, 7 faults

Hit Ratio: 5/12
Miss Ratio: 7/12

<table>
<thead>
<tr>
<th>Page Refs</th>
<th>3 Page Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault?</td>
<td>Page Contents</td>
</tr>
<tr>
<td>A</td>
<td>yes</td>
</tr>
<tr>
<td>B</td>
<td>yes</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>no</td>
</tr>
<tr>
<td>A</td>
<td>no</td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
</tr>
<tr>
<td>A</td>
<td>no</td>
</tr>
<tr>
<td>B</td>
<td>no</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>no</td>
</tr>
</tbody>
</table>

Evict c (farthest in the future)
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)
12 references, 9 faults

Miss rate: 9/12
Hit rate: 3/12

<table>
<thead>
<tr>
<th>Page Refs</th>
<th>3 Page Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fault?</td>
</tr>
<tr>
<td>A</td>
<td>yes</td>
</tr>
<tr>
<td>B</td>
<td>yes</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
</tr>
<tr>
<td>A</td>
<td>yes</td>
</tr>
<tr>
<td>B</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
</tr>
<tr>
<td>A</td>
<td>no</td>
</tr>
<tr>
<td>B</td>
<td>no</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>no</td>
</tr>
</tbody>
</table>

Evict A (oldest)
Evict B (oldest)
Intuitive Paging Behavior with Increasing Number of Page Frames
Belady's Anomaly (for FIFO)

As the number of page frames increase, so does the fault rate.

12 references, 10 faults
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?
### LRU

**12 references, 10 faults**

<table>
<thead>
<tr>
<th>Page Refs</th>
<th>3 Page Frames</th>
<th>Fault?</th>
<th>Page Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>yes</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>yes</td>
<td>B A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
<td>C B A</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
<td>D C B</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>yes</td>
<td>A D C</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>yes</td>
<td>B A D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
<td>E B A</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>no</td>
<td>A E B</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>no</td>
<td>B A E</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
<td>C B A</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
<td>D C B</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
<td>E D C</td>
<td></td>
</tr>
</tbody>
</table>

*Evict A (least recent)*
*Evict B (least recent)*
Least Recently Used Issues

- Does not suffer from Belady's anomaly
LRU and Anomalies

Anomalies cannot occur, why?

<table>
<thead>
<tr>
<th>Page Refs</th>
<th>4 Page Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fault?</td>
</tr>
<tr>
<td>A</td>
<td>yes</td>
</tr>
<tr>
<td>B</td>
<td>yes</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
</tr>
<tr>
<td>A</td>
<td>no</td>
</tr>
<tr>
<td>B</td>
<td>no</td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
</tr>
<tr>
<td>A</td>
<td>no</td>
</tr>
<tr>
<td>B</td>
<td>no</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
</tr>
</tbody>
</table>
LRU Implementation

- How to track “recency”?
  - use time
    - record time of reference with page table entry
    - use counter as clock
    - search for smallest time.
  - use stack
    - remove reference of page from stack (linked list)
    - push it on top of stack

- both approaches require large processing overhead, more space, and hardware support

- Maybe We can approximate it
NRU: A LRU Approximation

- NRU: Evict a page that is NOT recently used; LRU: evict a page that is LEAST recently used;
- NRU Implementation: simpler than LRU
  - additional reference bits
  - a register is kept per page
  - a one bit is set in the register if the page is referenced
  - the register is shifted by one after some time interval
  - 00110011 would be accessed more recently than 00010111
  - the page with register holding the lowest number is the least recently used.
  - the value may not be unique. use FIFO to resolve conflicts.
LRU Clock
(A Simple Not Recently Used)

- A Simple Not Recently Used (NRU) – Used by Unix
  - Replace page that is “cold 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?
Switching Gear

- So far, all we have talked about is memory management for a single process
- What about multiple applications/processes?
Thrashing and CPU Utilization

- As the page fault rate goes up, processes get suspended on page out queues for the disk.
- the system may try to optimize performance by starting new jobs.
  - But is it always good?
- starting new jobs will reduce the number of page frames available to each process, increasing the page fault requests.
- system throughput plunges.
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
the working set model assumes locality.

the principle of locality states that a program clusters its access to data and text temporally.

As the number of page frames increases above some threshold, the page fault rate will drop dramatically.
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 in Action

- **Algorithm**
  - if # free page frames > working set of some suspended process, then activate process and map in all its working set
  - if working set size of some process increases and no page frame is free, suspend process and release all its pages

- moving window over reference string used for determination.
- keeping track of working set.
Working Set Example

Window size is $\Delta$

12 references, 8 faults

<table>
<thead>
<tr>
<th>Page Contents</th>
<th>Fault?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>yes</td>
</tr>
<tr>
<td>A</td>
<td>no</td>
</tr>
<tr>
<td>A</td>
<td>no</td>
</tr>
<tr>
<td>A</td>
<td>yes</td>
</tr>
<tr>
<td>B</td>
<td>yes</td>
</tr>
<tr>
<td>B</td>
<td>no</td>
</tr>
<tr>
<td>B</td>
<td>no</td>
</tr>
<tr>
<td>B</td>
<td>yes</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
</tr>
</tbody>
</table>
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
Page Fault Frequency Version of Working Set

Page Fault Rate

Working set too small
Page Rate for Single Process
Reasonable Working Set
Working Set too large

Number of Page Frames
Working Set Solution

- Approximate working set model using timer and reference bit.
- Set timer to interrupt after approximately $x$ references, $\tau$.
- Remove pages that have not been referenced and reset reference bit.
Locality

- Most 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
Page Size Considerations

- **Small pages**
  - **Pros:**
    - Locality of reference tends to be small (256)
    - Less fragmentation
  - **Cons:**
    - Require large page tables

- **Large pages**
  - **Pros**
    - Small page table
    - I/O transfers have high seek time, so better to transfer more data per seek
  - **Cons:**
    - Internal fragmentation
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?