Lecture 12:
Dirty Tricks & Demand Paging

CSE 120: Principles of Operating Systems

Guest Starring:
Geoff Voelker

HW 3 Due 5/20
Advanced Functionality

- Last lecture covered the mechanics of paging
  - Address space broken up into pages
  - Pages stored in physical page frames
  - Mapping recorded in PTEs stored in page tables
  - Frequently used PTEs cached in the TLB

- Now we’re going to look at some advanced functionality that the OS can provide applications using virtual memory tricks
  - Shared memory
  - Copy on Write
  - Mapped files
Private virtual address spaces protect applications from each other
- Usually exactly what we want

But this makes it difficult to share data (have to copy)
- Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying

We can use shared memory to allow processes to share data using direct memory references
- Both processes see updates to the shared memory segment
  » Process B can immediately read an update by process A
- How are we going to coordinate access to shared data?
Page-Level Sharing

- How can we implement sharing using page tables?
  - Have PTEs in both tables map to the same physical frame
  - Each PTE can have different protection values
  - Must update both PTEs when page becomes invalid

- Can map shared memory at same or different virtual addresses in each process’ address space
  - Different: Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid (Why?)
  - Same: Less flexible, but shared pointers are valid (Why?)

- What happens if a pointer inside the shared segment references an address outside the segment?
Copy on Write

- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()
- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create shared mappings of parent pages in child virtual address space
  - Shared pages are protected as read-only in child
    - Reads happen as usual
    - Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
- How does this help fork()? (Implemented as Unix vfork())
CoW: Read Sharing to Start

Virtual Address

Page X

Offset

Frame A

Offset

Physical Address

Frame A

Offset

Frame A

Offset

Physical Memory

Page Y

Offset

Virtual Address

P0 Page Table

Frame A 1 0

P1 Page Table

Frame A 1 0
CoW: Exclusive Page on Write

Virtual Address

Physical Address

Page X
Offset

Frame B
Offset

Frame A
Offset

Page Y
Offset

Frame A
Offset

Frame B
Offset

Frame A
Offset

Frame B
Offset

P0 Page Table

Physical Memory

P1 Page Table
Mapped Files

- Mapped files enable processes to do file I/O using loads and stores
  - Instead of “open, read into buffer, operate on buffer, …”
- Bind a file to a virtual memory region (mmap() in Unix)
  - PTEs map virtual addresses to physical frames holding file data
  - Virtual address base + N refers to offset N in file
- Initially, all pages mapped to file are invalid
  - OS reads a page from file when invalid page is accessed
  - OS writes a page to file when evicted, or region unmapped
  - If page is not dirty (has not been written to), no write needed
    » Another use of the dirty bit in PTE
Mapped Files (2)

- File is essentially backing store for that region of the virtual address space (instead of using the swap file)
  - Virtual address space not backed by “real” files also called **Anonymous VM**

- Advantages
  - Uniform access for files and memory (just use pointers)
  - Less copying

- Drawbacks
  - Process has less control over data movement
    - OS handles faults transparently
  - Does not generalize to streamed I/O (pipes, sockets, etc.)
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 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 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.
  - And they come in all shapes and sizes.
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 going to survey various replacement algorithms:
- But first, why can they work?
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
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
    » But random is not necessarily the worst…what would be?
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)
## Belady’s Anomaly w/FIFO

<table>
<thead>
<tr>
<th>Page References</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>3</th>
<th>1</th>
<th>1</th>
<th>5</th>
<th>5</th>
<th>2</th>
<th>2</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>2</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>3</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>4</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>1</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>2</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>5</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>1</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>2</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>3</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>4</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>5</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>

- **Page References**
  - 1: 1, 1, 3, 3, 1, 1, 5, 5, 2, 2, 4, 4
  - 2: 2, 2, 4, 4, 2, 2, 1, 1, 3, 3, 5, 5
  - 3: 1, 1, 1, 4, 4, 4, 5, 5, 5, 5
  - 4: 2, 2, 2, 1, 1, 1, 1, 1, 2, 2
  - 5: 3, 3, 3, 3, 2, 2, 2, 2, 3, 3

The number of page faults for each method:
- FIFO: 10
- LRU: 9
- Belady’s Anomaly: 12
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 (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
    - Use additional hands
Next time…

- New topic: File systems
- Read Chapters 9, 10
- Homework 3 is due!