Vnodes

- **Every open file has an associated vnode struct**
- **All file system types (FFS, NFS, etc.) have vnodes**
  - `v_data` points to FS-specific data
  - Function pointers for operations (open/read/write/...)
- **When refcount → 0, `inactive()` called**
  - Does not "deallocate" vnode—can be quickly reopened
  - `reclaim()` revokes vnode for another use

Name cache

- **Caches $\langle$dir, name$\rangle$ $\rightarrow$ `vnode` translations**
  - Both positive and negative lookups cached
- **Need to invalidate all names for a `vnode` (mv dir)**
  - Each vnode has a "capability," also contained in cache
  - Bump 32-bit capability to flush cache efficiently
  - When counter wraps, invalidate all of name cache
- **Need to invalidate negative lookups when directory changed**
  - Bump capability

Buffer cache

- **Caches file blocks in memory**
  - Hash table maps $\langle$vnode, offset$\rangle \to$ buffer
  - Freelists keep buffers not in use
- **Operations on buffers:**
  - `bread()` – fill buffer from underlying file
  - `breadn()` – like `bread()`, but start read ahead
  - `brelse()` – relinquish unmodified buffer
  - `bwrite()` – synchronously write data to disk
  - `bawrite()` – asynchronously write data to disk
  - `bdwrite()` – schedule delayed write

Write policies

- **When to use synchronous `bwrite()`?**
  - `fsync()` system call
  - Network file systems with synchronous writes
  - When cleaning reclaimed buffer
  - When order of disk writes matters
- **When to use `bdwrite()`?**
  - Buffer may be modified again
- **When to use `bawrite()`?**
  - Buffer full, might as well clean it

Buffer free lists

- **Locked – unused (for superblock?)**
- **LRU**
- **Age**
  - Deleted files pushed onto front (reuse immediately)
  - Read-ahead blocks placed at end
- **Empty**
  - No physical memory

Algorithm: Optimal

- **Definition: Maximize #hits/#references**
  - Evict block that will be referenced furthest in the future
- **How to implement**
  - Gather trace of all references
  - Retroactively figure out what you should have done
- **Useful only as a point of comparison**
- **LRU used to approximate algorithm**
  - Most recently touched most likely to be touched soon
- **If you could implement Optimal, is it best?**
  - Best hit rate, but what about fairness?

Algorithm: FBR

- **Idea: Weight blocks by frequency of reference**
  - Count # of references
  - Evict blocks with lowest counts
- **Problem: Many short-spaced references, then none**
  - Don't bump count in "new" section of LRU queue
- **Problem: Evicted right as blocks leave new**
  - Only evict in "old" section of queue
- **Problem: Never evict blocks with high count**
  - Decay by half when total of counts reach some max

Algorithm: LRU-k

- **LRU based on kth most recent access**
- **Regular LRU is LRU-1**
- **LRU-2 works well in practice**
  - Great for walking indexed data structures
- **Computationally expensive**
  - Costs $\log N$ to manipulate buffer (with cache size N)

Algorithm: 2Q

- **Goal: Cheaper algorithm with benefits of LRU-2**
- **Idea: Keep 2 queues:**
  - A_1 for buffers accessed only once – FIFO
  - A_m for buffers accessed multiple times – LRU
- **Problem: Sizing A_1 vs. A_m is hard**
- **Solution: Ghost buffers**
  - Break A_1 into $A_{1\text{in}}$ and $A_{1\text{out}}$
  - $A_{1\text{out}}$ doesn't actually contain buffered data

Algorithm: SEQ

- Detect sequential accesses
- Apply MRU to pages fetched by sequential access
- Does not detect looping behavior

EELRU

- **Idea: Ordinarily use simple LRU**
  - If many recently fetched pages being evicted, move to fallback algorithm.
- **Divide LRU queue into three regions**
  - LRU region – most recently accessed pages
  - early region – less recently accessed pages
  - late region – even less recently accessed pages
  - Use ghost buffers to track more buffers than memory size
- **Evict from head of early or head of late point, based on mathematical predictions**