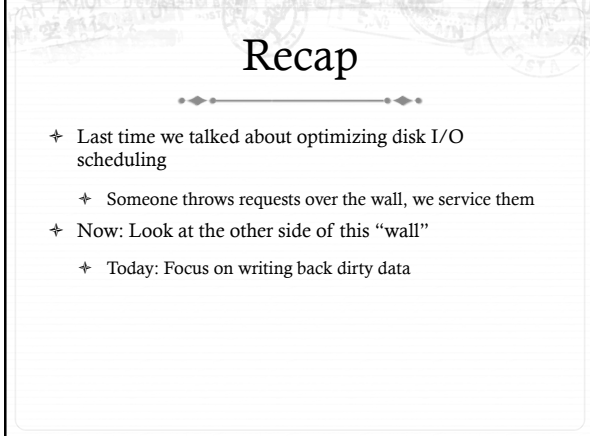


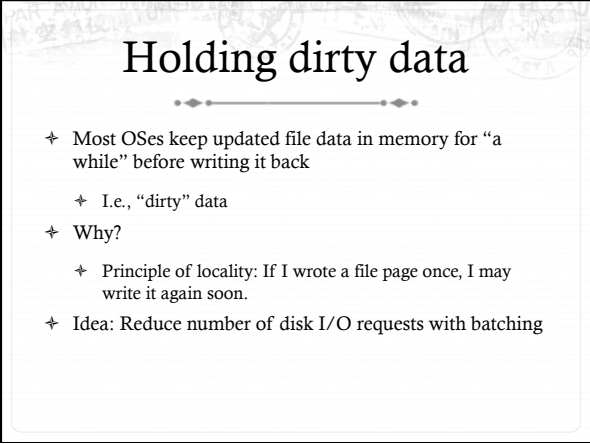
# The Page Cache

Don Porter  
CSE 506



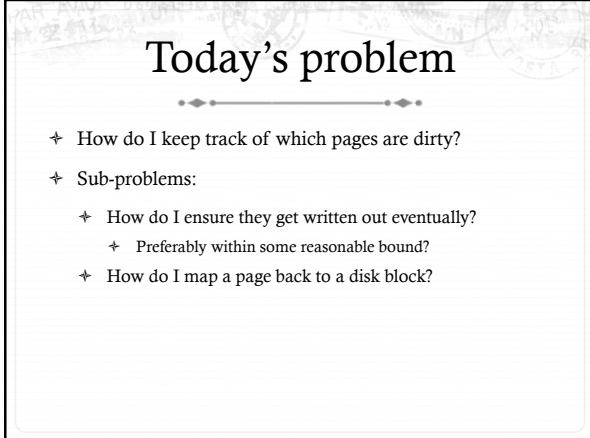
# Recap

- ✦ Last time we talked about optimizing disk I/O scheduling
  - ✦ Someone throws requests over the wall, we service them
- ✦ Now: Look at the other side of this “wall”
  - ✦ Today: Focus on writing back dirty data



# Holding dirty data

- ✦ Most OSes keep updated file data in memory for “a while” before writing it back
  - ✦ I.e., “dirty” data
- ✦ Why?
  - ✦ Principle of locality: If I wrote a file page once, I may write it again soon.
- ✦ Idea: Reduce number of disk I/O requests with batching



# Today’s problem

- ✦ How do I keep track of which pages are dirty?
- ✦ Sub-problems:
  - ✦ How do I ensure they get written out eventually?
    - ✦ Preferably within some reasonable bound?
  - ✦ How do I map a page back to a disk block?

## Starting point

- ✦ Just like JOS, Linux represents physical memory with an array of page structs
  - ✦ Obviously, not the exact same contents, but same idea
- ✦ Some memory used for I/O mapping, device buffers, etc.
  - ✦ Other memory associated with processes, files
- ✦ How to represent these associations?
  - ✦ For today, interested in "What pages go with this process/file/etc?"
  - ✦ Tomorrow: What file does this page go to?

## Simple model

- ✦ Each page needs:
  - ✦ A reference to the file/process/etc. it belongs to
    - ✦ Assume for simplicity no page sharing
  - ✦ An offset within the file/process/etc
- ✦ Unifying abstraction: the address space
  - ✦ Each file inode has an address space (0—file size)
  - ✦ So do block devices that cache data in RAM (0—dev size)
  - ✦ The (anonymous) virtual memory of a process has an address space (0—4GB on x86)

## Address space representation

- ✦ We saw before that a process uses a list and tree of VM area structs (VMAs) to represent its address space
- ✦ A VMA can be anonymous (no file backing)
  - ✦ Or it can map (part of) a file
- ✦ Page table stores association with physical page
- ✦ Good solution:
  - ✦ Sparse, like most process address spaces
  - ✦ Scalable: can efficiently represent large address spaces

## Tracking file pages

- ✦ What data structure to use for a file?
  - ✦ No page tables for files
- ✦ For example: What page stores the first 4k of file "foo"
  - ✦ What data structure to use?
    - ✦ Hint: Files can be small, or very, very large

## The Radix Tree

- ✦ A space-optimized trie
- ✦ Trie: Rather than store entire key in each node, traversal of parent(s) builds a prefix, node just stores suffix
  - ✦ Especially useful for strings
- ✦ Prefix less important for file offsets, but does bound key storage space
- ✦ More important: A tree with a branching factor  $k > 2$ 
  - ✦ Faster lookup for large files (esp. with tricks)
- ✦ Note: Linux's use of the Radix tree is constrained

## A bit more detail

- ✦ Assume an upper bound on file size when building the radix tree
  - ✦ Can rebuild later if we are wrong
- ✦ Specifically: Max size is 256k, branching factor ( $k$ ) = 64
- ✦  $256k / 4k$  pages = 64 pages
  - ✦ So we need a radix tree of height 1 to represent these pages

## Tree of height 1

- ✦ Root has 64 slots, can be null, or a pointer to a page
- ✦ Lookup address X:
  - ✦ Shift off low 12 bits (offset within page)
  - ✦ Use next 6 bits as an index into these slots ( $2^6 = 64$ )
  - ✦ If pointer non-null, go to the child node (page)
  - ✦ If null, page doesn't exist

## Tree of height n

- ✦ Similar story:
  - ✦ Shift off low 12 bits
- ✦ At each child shift off 6 bits from middle (starting at  $6 * (\text{distance to the bottom} - 1)$  bits) to find which of the 64 potential children to go to
  - ✦ Use fixed height to figure out where to stop, which bits to use for offset
- ✦ Observations:
  - ✦ "Key" at each node implicit based on position in tree
  - ✦ Lookup time constant in height of tree
    - ✦ In a general-purpose radix tree, may have to check all  $k$  children, for higher lookup cost

## Fixed heights

- ✦ If the file size grows beyond max height, must grow the tree
- ✦ Relatively simple: Add another root, previous tree becomes first child
- ✦ Scaling in height:
  - ✦ 1:  $2^6(6*1) + 12 = 256$  KB
  - ✦ 2:  $2^6(6*2) + 12 = 16$  MB
  - ✦ 3:  $2^6(6*3) + 12 = 1$  GB
  - ✦ 4:  $2^6(6*4) + 12 = 16$  GB
  - ✦ 5:  $2^6(6*5) + 12 = 4$  TB

## Back to address spaces

- ✦ Each address space for a file cached in memory includes a radix tree
  - ✦ Radix tree is sparse: pages not in memory are missing
- ✦ Radix tree also supports tags: such as dirty
  - ✦ A tree node is tagged if at least one child also has the tag
- ✦ Example: I tag a file page dirty
  - ✦ Must tag each parent in the radix tree as dirty
  - ✦ When I am finished writing page back, I must check all siblings; if none dirty, clear the parent's dirty tag

## When does Linux write pages back?

- ✦ Synchronously: When a program calls a sync system call
- ✦ Asynchronously:
  - ✦ Periodically writes pages back
  - ✦ Ensures that they don't stay in memory too long

## Sync system calls

- ✦ `sync()` – Flush all dirty buffers to disk
- ✦ `fsync(fd)` – Flush all dirty buffers associated with this file to disk (including changes to the inode)
- ✦ `fdatasync(fd)` – Flush only dirty data pages for this file to disk
  - ✦ Don't bother with the inode

## How to implement sync?

- ✦ Goal: keep overheads of finding dirty blocks low
  - ✦ A naïve scan of all pages would work, but expensive
  - ✦ Lots of clean pages
- ✦ Idea: keep track of dirty data to minimize overheads
  - ✦ A bit of extra work on the write path, of course

## How to implement sync?

- ✦ Background: Each file system has a super block
  - ✦ All super blocks in a list
- ✦ Each super block keeps a list of dirty inodes
- ✦ Inodes and superblocks both marked dirty upon use

## Simple traversal

```

for each s in superblock list:
    if (s->dirty) writeback s
    for i in inode list:
        if (i->dirty) writeback i
        if (i->radix_root->dirty) :
            // Recursively traverse tree writing
            // dirty pages and clearing dirty flag
  
```

## Asynchronous flushing

- ✦ Kernel thread(s): pdflush
  - ✦ Recall: a kernel thread is a task that only runs in the kernel's address space
  - ✦ 2-8 threads, depending on how busy/idle threads are
- ✦ When pdflush runs, it is given a target number of pages to write back
  - ✦ Kernel maintains a total number of dirty pages
  - ✦ Administrator configures a target dirty ratio (say 10%)

## pdflush

- ✦ When pdflush is scheduled, it figures out how many dirty pages are above the target ratio
- ✦ Writes back pages until it meets its goal or can't write more back
  - ✦ (Some pages may be locked, just skip those)
- ✦ Same traversal as sync() + a count of written pages
  - ✦ Usually quits earlier

## How long dirty?

- ✦ Linux has some inode-specific bookkeeping about when things were dirtied
- ✦ pdflush also checks for any inodes that have been dirty longer than 30 seconds
  - ✦ Writes these back even if quota was met
- ✦ Not the strongest guarantee I've ever seen...

## Mapping pages to disk blocks

- ✦ Most disks have 512 byte blocks; pages are generally 4K
  - ✦ Some new "green" disks have 4K blocks
  - ✦ Per page in cache – usually 8 disk blocks
- ✦ When blocks don't match, what do we do?
  - ✦ Simple answer: Just write all 8!
  - ✦ But this is expensive – if only one block changed, we only want to write one block back

## Buffer head

- ✦ Simple idea: for every page backed by disk, store an extra data structure for each disk block, called a buffer\_head
- ✦ If a page stores 8 disk blocks, it has 8 buffer heads
- ✦ Example: write() system call for first 5 bytes
  - ✦ Look up first page in radix tree
  - ✦ Modify page, mark dirty
  - ✦ Only mark first buffer head dirty

## More on buffer heads

- ✦ On write-back (sync, pdflush, etc), only write dirty buffer heads
- ✦ To look up a given disk block for a file, must divide by buffer heads per page
  - ✦ Ex: disk block 25 of a file is in page 3 in the radix tree
- ✦ Note: memory mapped files mark all 8 buffer\_heads dirty. Why?
  - ✦ Can only detect write regions via page faults

## Raw device caching

- ✦ For simplicity, we've focused on file data
- ✦ The page cache can also cache raw device blocks
  - ✦ Disks can have an address space + radix tree too!
- ✦ Why?
  - ✦ On-disk metadata (inodes, directory entries, etc)
  - ✦ File data may not be stored in block-aligned chunks
    - ✦ Think extreme storage optimizations
  - ✦ Other block-level transformations between FS and disk (e.g., encryption, compression, deduplication)

## Summary

- ✦ Seen how mappings of files/disks to cache pages are tracked
  - ✦ And how dirty pages are tagged
  - ✦ Radix tree basics
- ✦ When and how dirty data is written back to disk
- ✦ How difference between disk sector and page sizes are handled