Page Cache

Nima Honarmand

The Address Space Abstraction

- Unifying abstraction:
  - Each file 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 32-bit x86)
- In other words, all page mappings can be thought of as an (object, offset) tuple

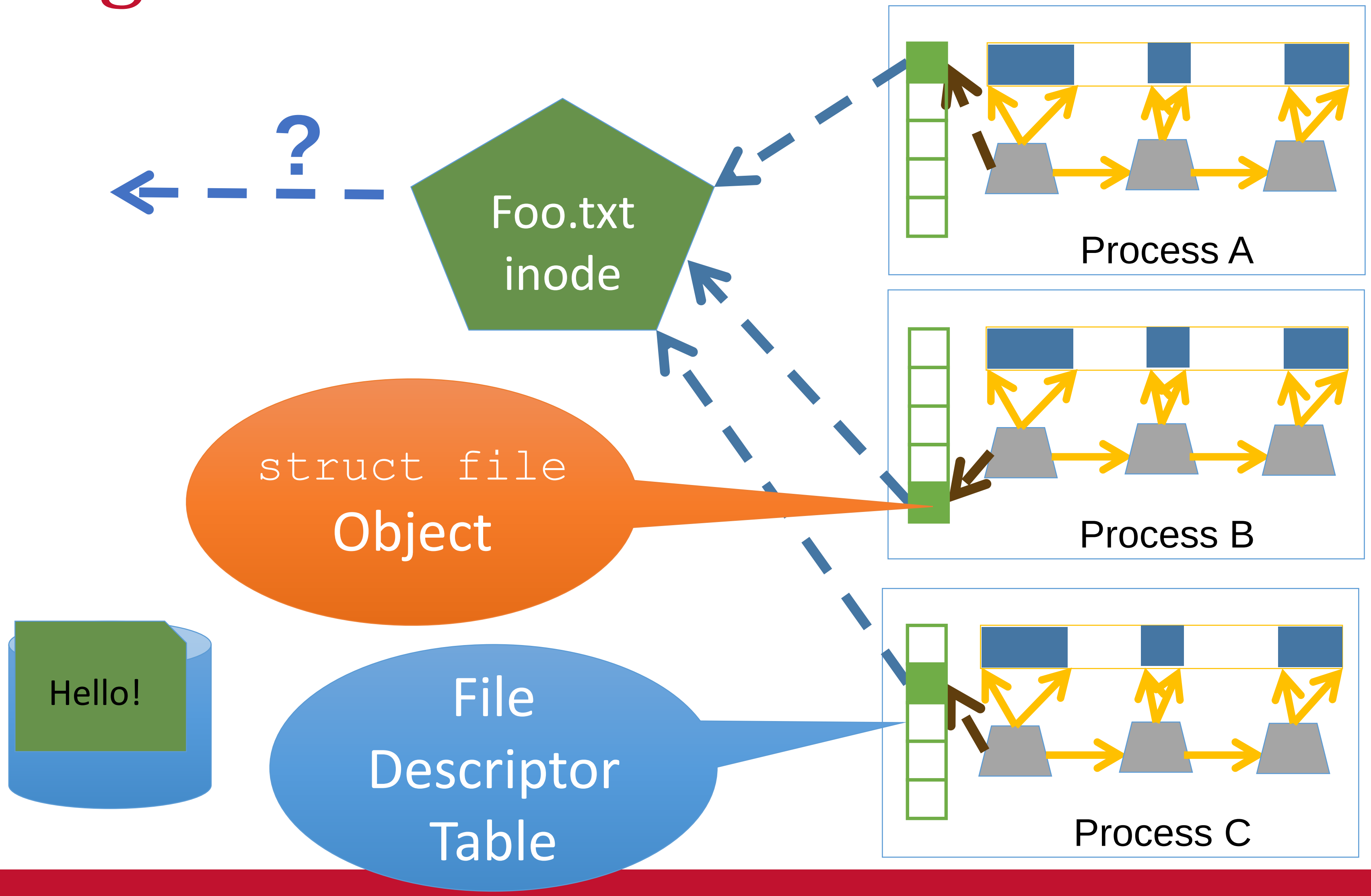
Recap: Anonymous Mapping

- “Anonymous” memory — no file backing it
  - E.g., the stack or heap of a process
  - Can be shared between processes
- How do we figure out virtual to physical mapping?
  - Just walk the page tables!
- Linux doesn’t do anything outside of the page tables to track this mapping

File Mappings

- A VMA can also represent a memory mapped file
  - A VMA may map only part of the file
  - VMA includes a ***struct file*** pointer and an ***offset*** into file
    - Offset must be at page granularity
- The kernel can also map file pages to service **`read()`** or **`write()`** system calls
- Goal: We only want to load a file into memory once!

Logical View



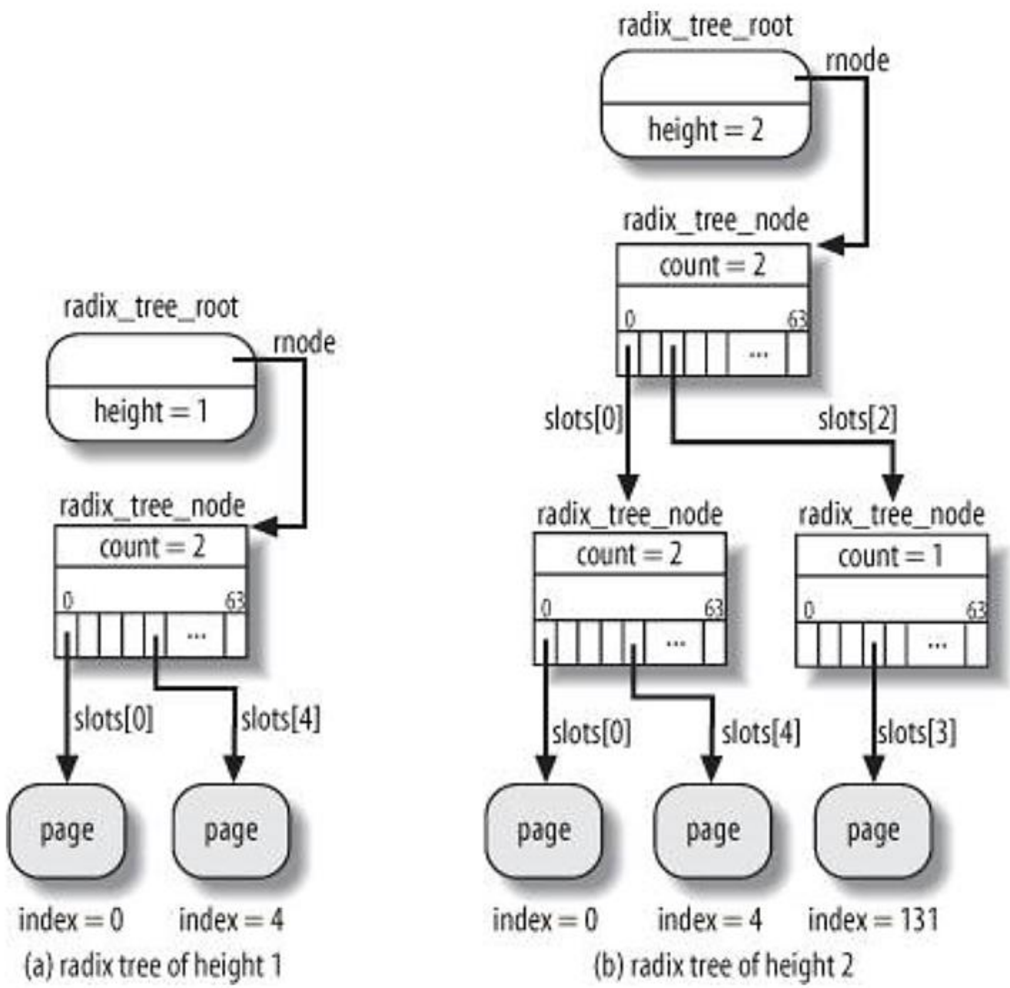
Tracking In-memory File Pages

- What data structure to use for a file?
  - E.g., what page stores the first 4k of file “foo”?
  - No page tables for files
- What data structure to use?
  - Hint: Files can be small, or very, very large

The Radix Tree

- A prefix tree
  - Rather than store entire key in each node, traversal of parent(s) builds a prefix, node just stores suffix
  - "Key" at each node implicit based on position in tree
- More important: A tree with a branching factor k > 2
  - Faster lookup for large files (esp. with tricks)
- Does this remind you of another data structure we have already seen?
  - Yes, page table itself
    - Similar problem: supporting both small and big address spaces
    - Similar solution: A radix tree

Radix Tree Structure



Source: Understanding Linux kernel, 3rd Ed

Using Radix Tree for File Address Space

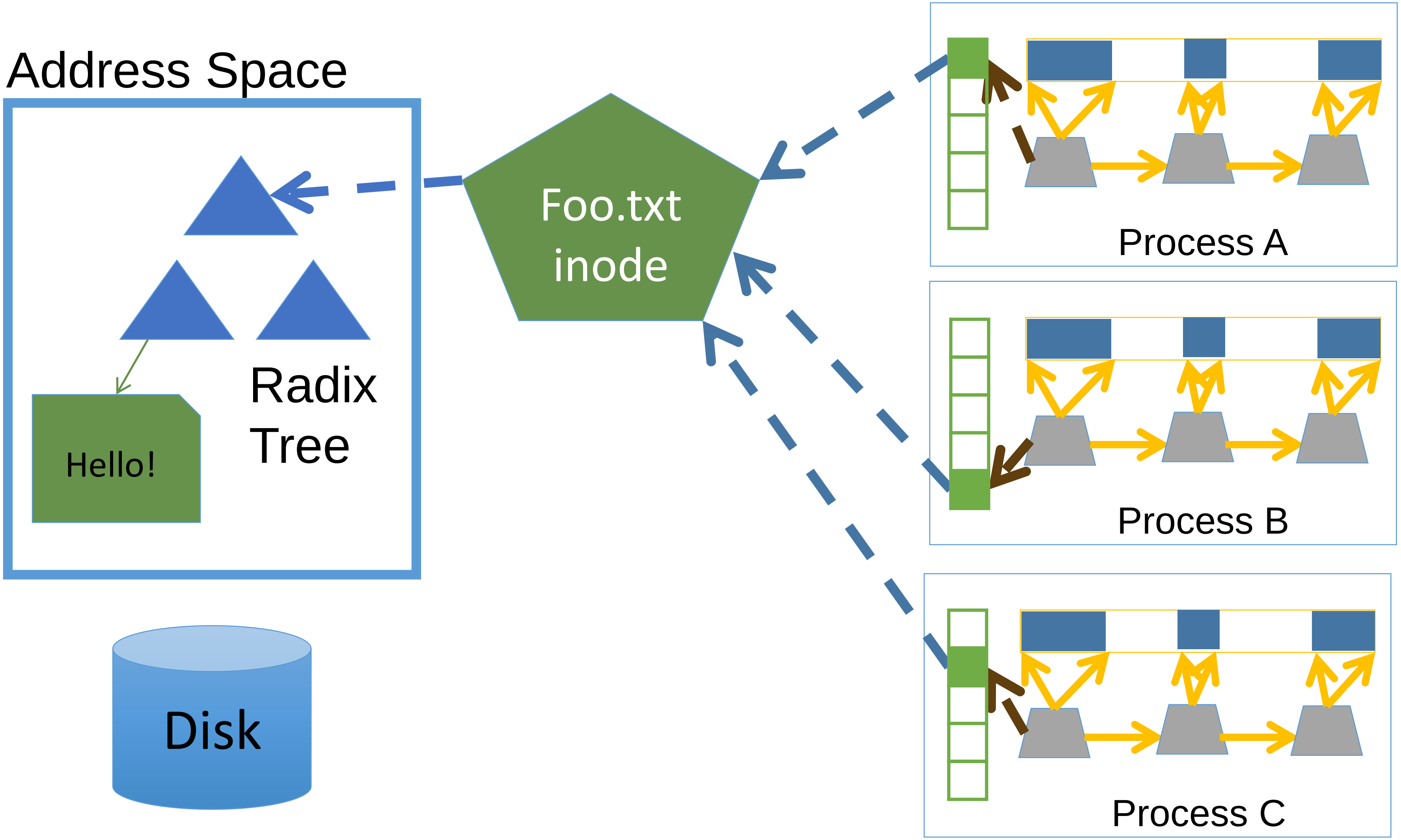
- Each address space for a file cached in memory includes a radix tree
  - Radix tree is sparse: pages not in memory are missing
- What's the key?
  - Offset of the file
- What's the value stored in a leaf?
  - Pointer to physical page descriptor

- Assume an upper bound on file size when building the radix tree (rebuild later if wrong)
- Example: 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

Increasing Tree Height

- 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*1) + 12) = 256 KB
  - 2: 2^((6*2) + 12) = 16 MB
  - 3: 2^((6*3) + 12) = 1 GB
  - 4: 2^((6*4) + 12) = 16 GB
  - 5: 2^((6*5) + 12) = 4 TB

Logical View



Tracking Dirty Pages

- 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

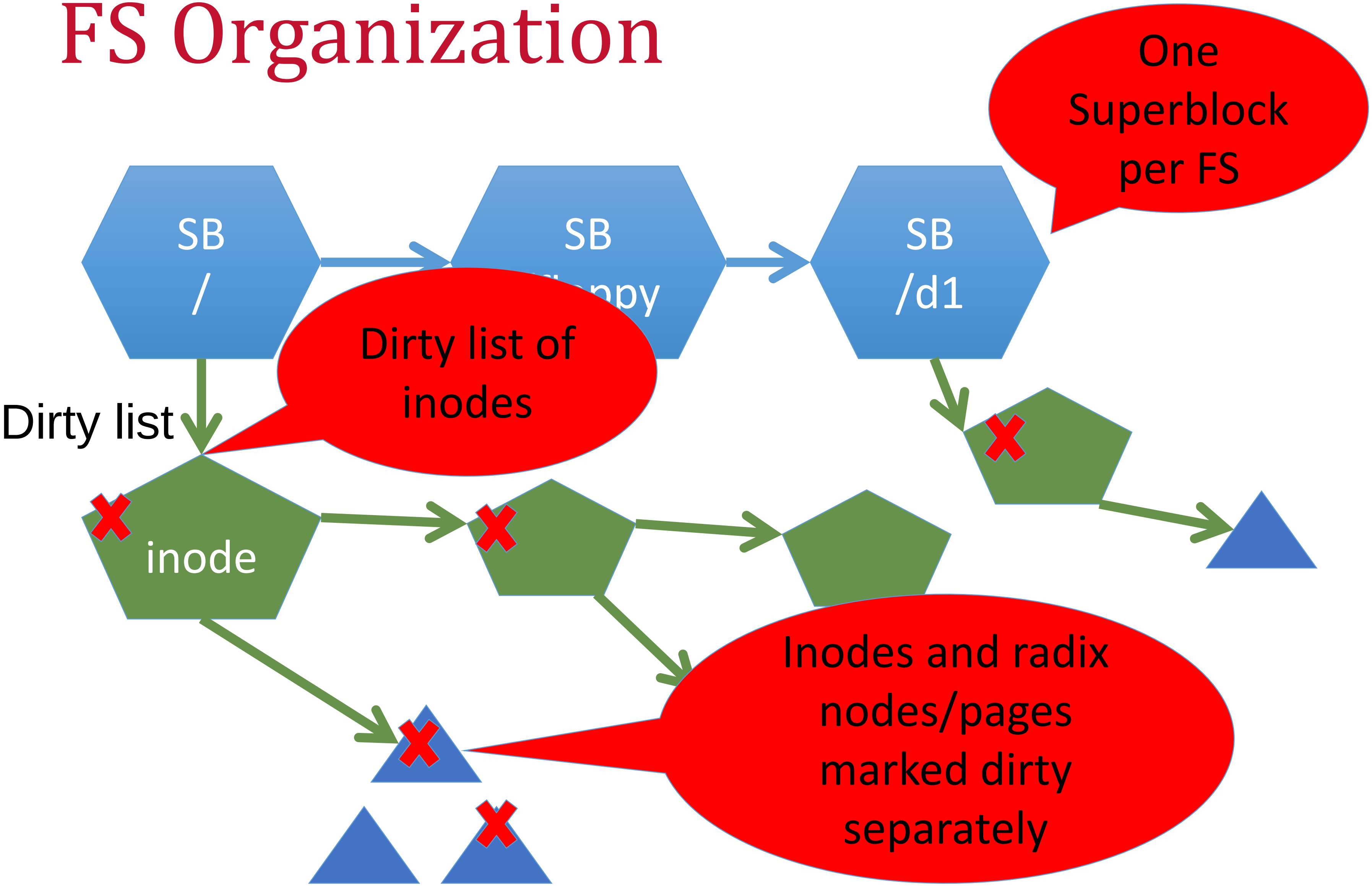
`sync()` System Calls

- Most OS don't write file updates to disk immediately
  - OS tries to optimize disk arm movement
  - Application can force write back using ***sync*** system calls
- `sync()` – Flush all dirty buffers to disk
- `syncfs(fd)` – Flush all dirty buffers to disk for FS containing `fd`
- `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 inode, unless critical metadata changed

How to implement `sync()`?

- Each file system has a ***superblock***
  - Superblock keeps global meta data about the file system such as size, list of inodes, list of free and used blocks, etc.
  - All superblocks in a list in the kernel
- Each superblock keeps a list of dirty inodes
- inode has a pointer to the address space (including the radix tree)
- Radix tree tracks dirty pages

FS Organization



Asynchronous Flushing

- Kernel thread(s): ***pdflush***
  - Kernel thread: task that only runs in kernel's address space
  - 2 – 8 pdflush 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%)
- Same traversal as `sync()` + a count of written pages
  - Until the target is met

Synthesis: `read()` syscall

```
int read(int fd, void *buf, size_t bytes);
```

- `fd`: File descriptor index
- `buf`: Buffer kernel writes the read data into
- `bytes`: Number of bytes requested
- Returns: bytes read (if >= 0), or -errno

Simple steps

- Translate `fd` to a `struct file` (if valid)
  - Increase reference count
- Validate that `sizeof(buf)` >= bytes requested
  - and that `buf` is a valid address in user space
- Search the radix tree for the appropriate page of data
- If not found, or **PG_uptodate** flag not set, re-read from disk
  - `read_cache_page()`
- Copy into the user buffer
  - up to `inode->i_size` (i.e., the file size)

Requesting a page read

- Allocate a physical page to hold the file content
- First, the physical page must be locked
  - Atomically set a lock bit in the page descriptor
  - If this fails, the process sleeps until page is unlocked
- Once the page is locked, double-check that no one else has re-read from disk before locking the page
- Invoke `address_space->readpage()` (set by FS)

Generic readpage()

- Recall that most disk blocks are 512 bytes, yet pages are 4k
- If the blocks are contiguous on disk, read entire page as a batch
- If not, read each block one at a time
- These block requests are sent to the backing device I/O scheduler

After readpage()

- Mark the page accessed (for LRU reclaiming)
- Unlock the page
- Then copy the data, update file access time, advance file offset, etc.

Copying data to user

- Kernel needs to be sure that `buf` is a valid address
  - Remember: `buf` is a pointer in user space
- How to do it?
  - Can walk appropriate page table entries
- What could go wrong?
  - Concurrent munmap from another thread
  - Page might be lazy allocated by kernel

Trick

- What if we don't do all of this validation?
  - Looks like kernel had a page fault
  - Usually REALLY BAD
- Idea: set a kernel flag that says we are in `copy_to_user()`
  - If a page fault happens for a user address, don't panic
    - Just handle demand faults
  - If the page is really bad, write an error code into a register so that it breaks the write loop; check after return

Benefits

- This trick actually speeds up the common case (where `buf` is ok)
- Avoids complexity of handling weird race conditions
- Still need to be sure that `buf` address isn't in the kernel