CSE 506:
Operating Systems

Block Cache

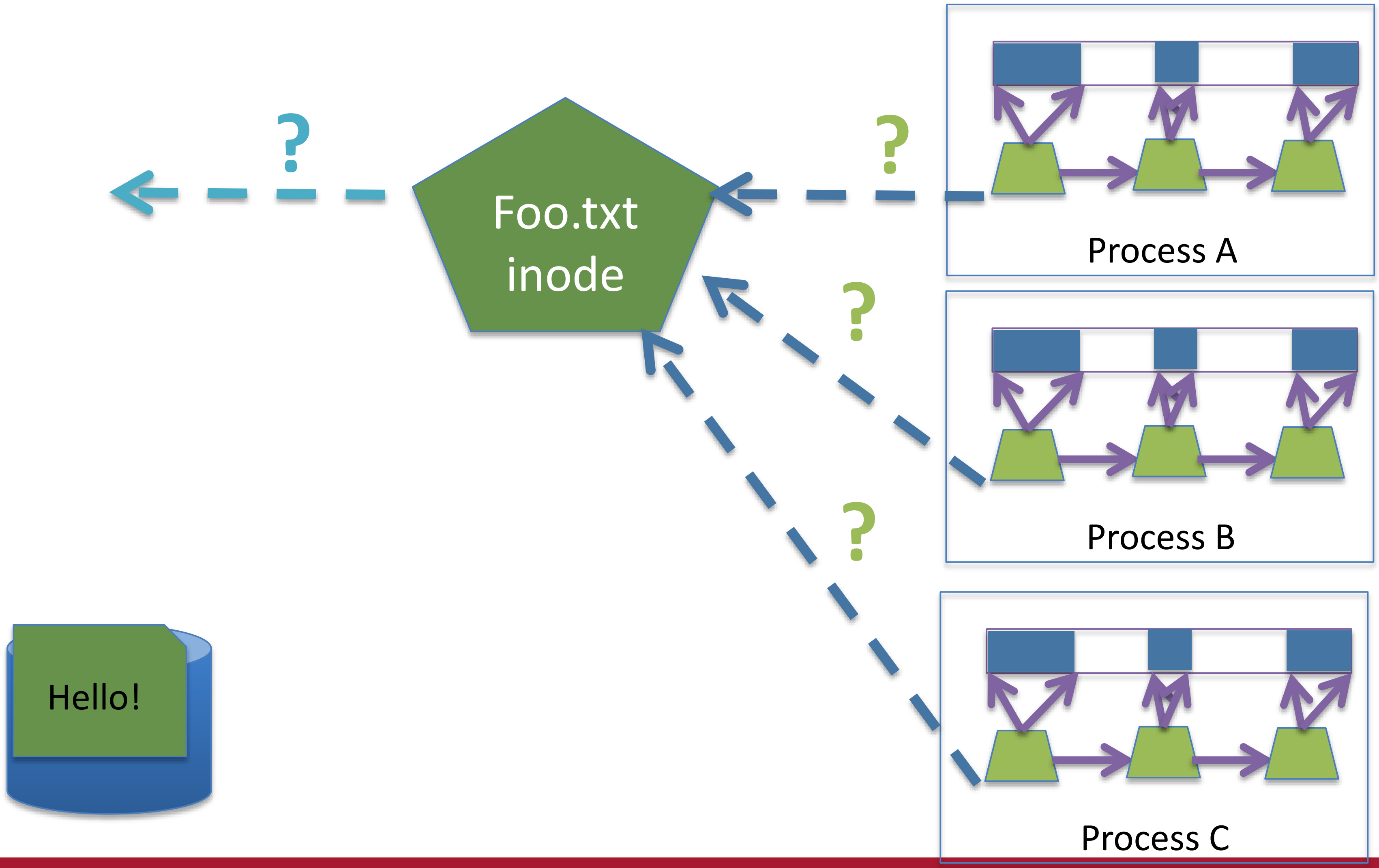
Address Space Abstraction

- Given a file, which physical pages store its data?
- Each file inode has an ***address space*** (0—file size)
  - So do block devices that cache data in RAM (0—dev size)
  - So does virtual memory of a process (0—16EB in 64-bit)
- All page mappings are (object, offset) tuple

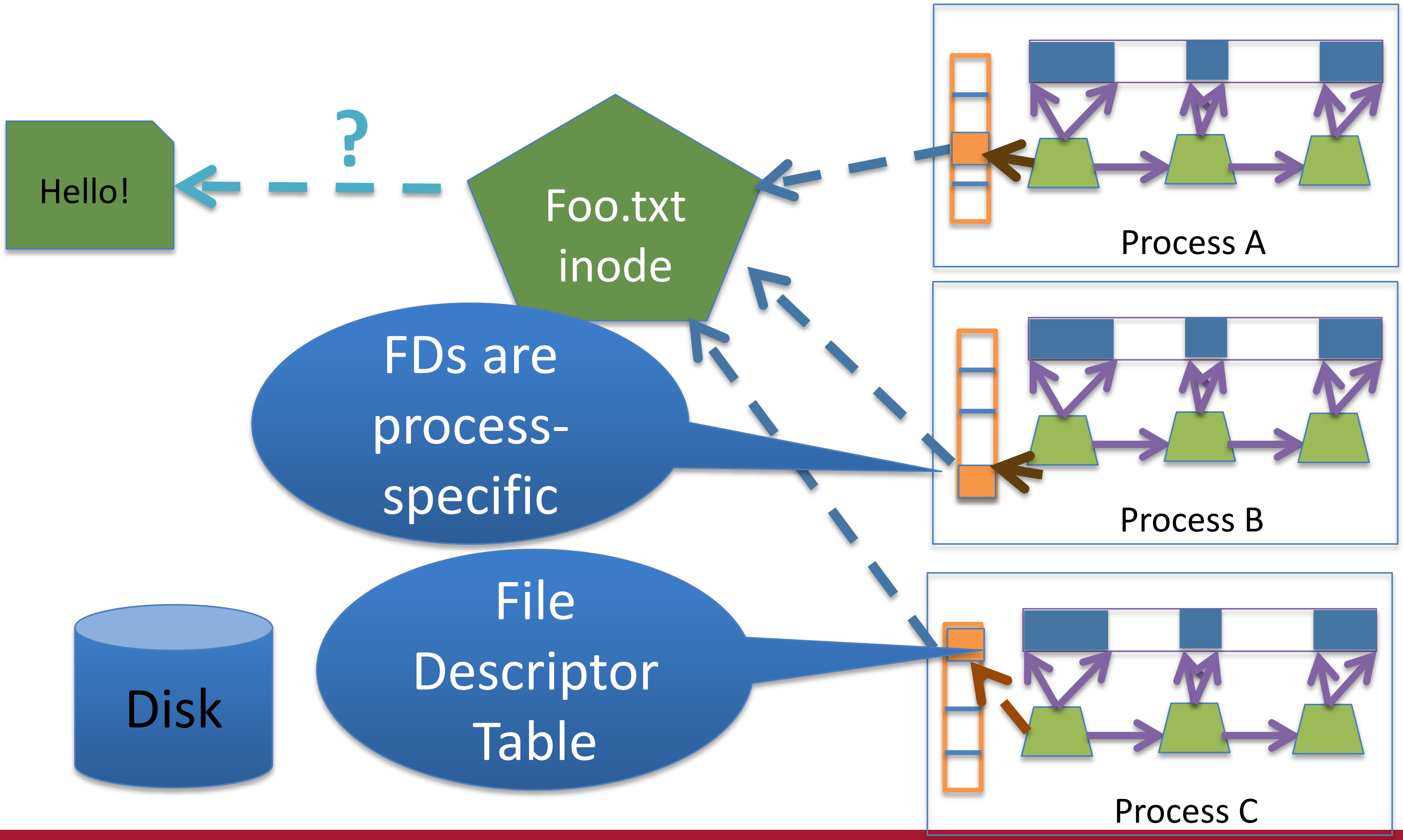
Types of Address Spaces

- “Anonymous” memory – no file backing it
  - e.g., the stack for a process
  - Not shared between processes
    - Will discuss sharing and swapping later
  - How do we figure out virtual to physical mapping?
    - Data structure to map virtual to physical addresses
    - Some OSes (e.g, Linux) just walk the page tables
- File contents – backed by storage device
  - Kernel can map file pages into application

Logical View



Logical View



FDs are process-specific

File Descriptor Table

Tracking Inode Pages

- What data structure to use for an inode?
  - No page tables for files
- Ex: 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 without key in each node
    - Traversal of parent(s) builds a prefix
- Tree with branching factor k > 2 is fast
  - Faster lookup for large files (esp. with tricks)
- Assume upper bound file size when building
  - Can rebuild later if we are wrong
- Ex: Max size is 256k, branching factor (k) = 64
- 256k / 4k pages = 64 pages
  - 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

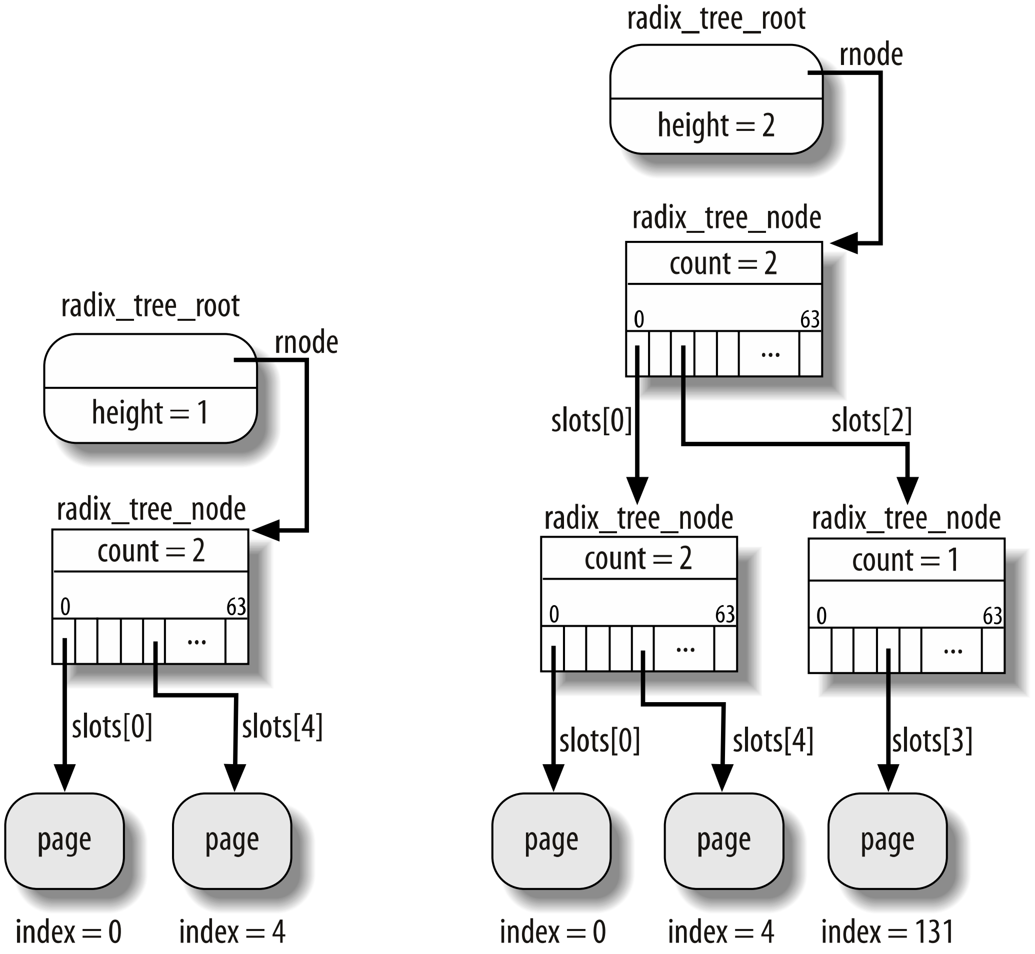
Tree of height n

- Shift off low 12 bits
- At each child, shift off 6 bits from middle
  … (starting at 6 * (distance to the bottom – 1) bits)
  - To find which of the 64 potential children to go to
  - Fixed height to figure out where to stop (use bits 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
      - Higher lookup cost

Fixed heights

- If the file size grows beyond max height
  - Grow the tree
    - Add another root, previous tree becomes first child
- Scaling in height:
  - 1: $2^{(6 \cdot 1)+12}$ = 256 KB
  - 2: $2^{(6 \cdot 2)+12}$ = 16 MB
  - 3: $2^{(6 \cdot 3)+12}$ = 1 GB
  - 4: $2^{(6 \cdot 4)+12}$ = 16 GB
  - 5: $2^{(6 \cdot 5)+12}$ = 4 TB

From "Understanding the Linux Kernel"



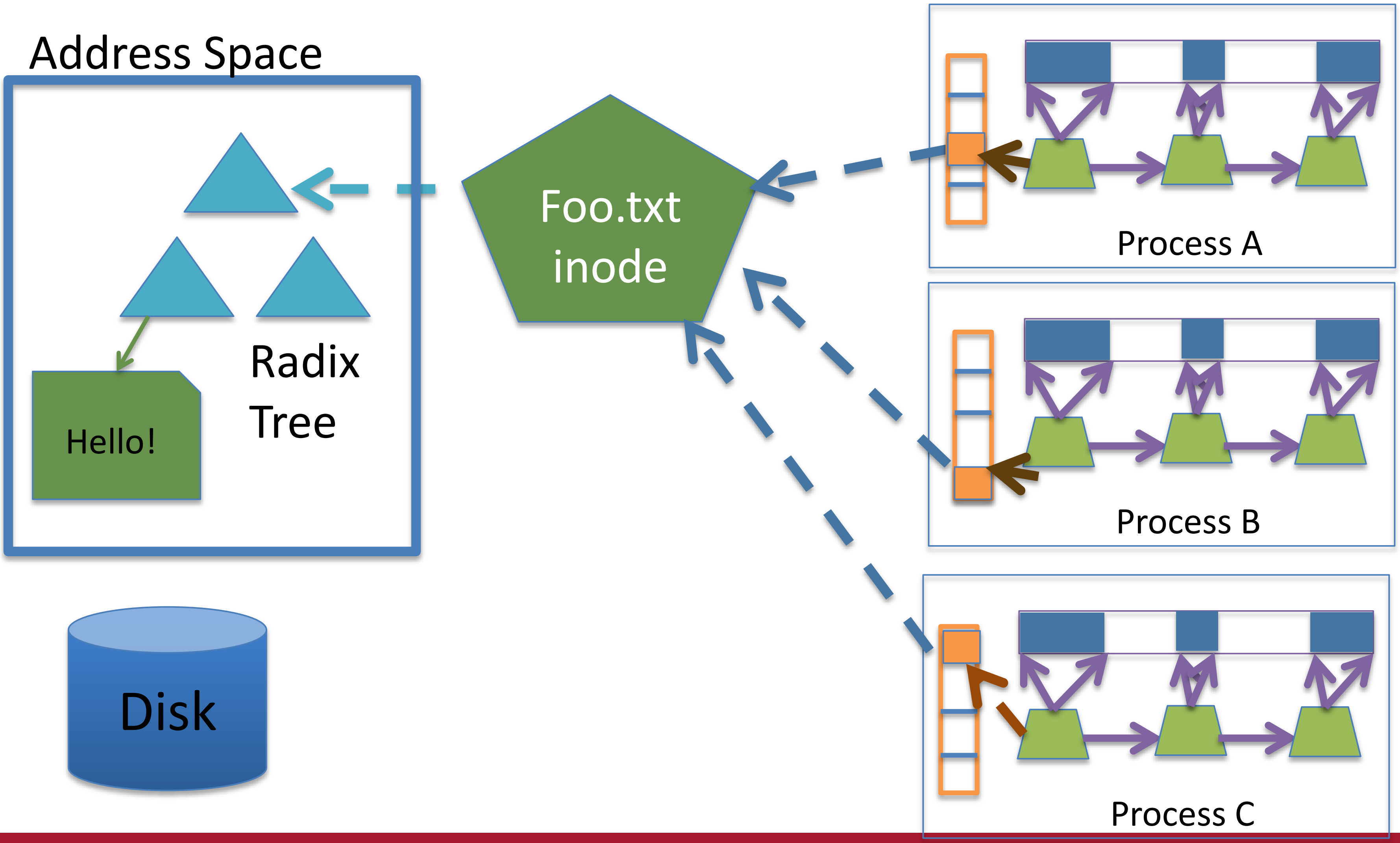
(a) radix tree of height 1

(b) radix tree of height 2

Block Cache (File Address Spaces)

- Cached inodes (files) have a radix tree
  - Points to physical pages
  - Tree is sparse: pages not in memory are missing
- Radix tree also supports tags
  - A tree node is tagged if at least one child also has the tag
  - Example: Tag a file’s page ‘dirty’
    - Must tag each parent in the radix tree as dirty
    - When finished writing page back
      - Check all siblings
        - If none dirty, clear the parent’s dirty tag

Logical View



Reading Block Cache

- VFS does a `read()`
  - Looks up in inode’s radix tree
  - If found in block cache, can return data immediately
- If data not in block cache
  - Call’s FS-specific `read()` operation
    - Schedules getting data from disk
      - Puts process to sleep until disk interrupt
    - When disk read is done
      - Populate radix tree with pointer to page
      - Wake up process
  - Repeat VFS read attempt

Dirty Pages

- OSes do not write file updates to disk immediately
  - Pages might be written again soon
  - Writes can be done later, “when convenient”
- OS instead tracks “dirty” pages
  - Ensures that write back isn’t delayed too long
    - Lest data be lost in a crash
- Application can force immediate write back
  - `sync()` system calls (and some open/mmap options)

Sync System Calls

- `sync()` – Flush all dirty buffers to disk
- `fsync(fd)` – Flush fd’s dirty buffers to disk
  - Including inode
- `fdatasync(fd)` – Flush fd’s dirty buffers to disk
  - Don’t bother with the inode

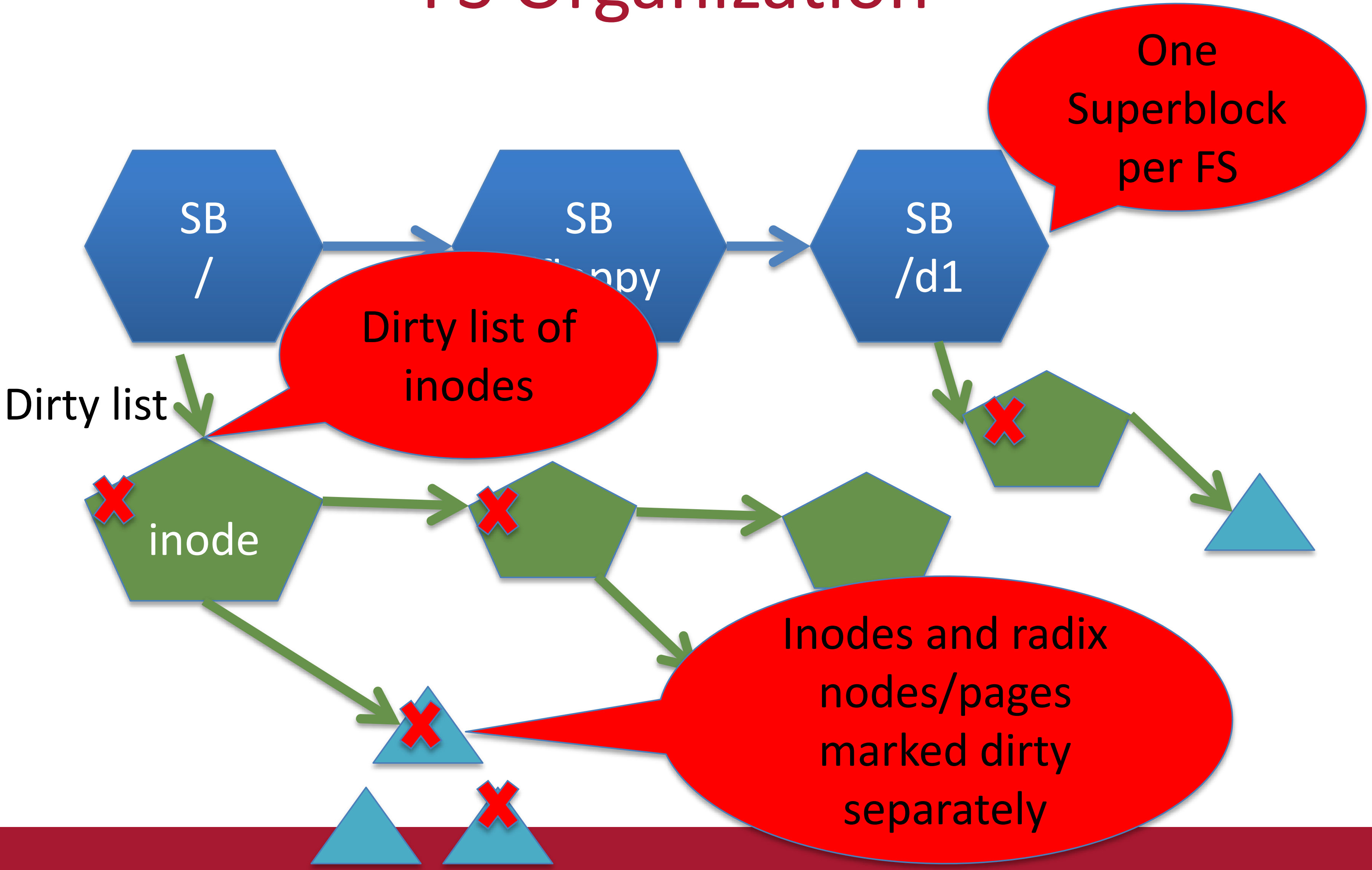
How to implement sync?

- 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 in-memory super block keeps list of dirty inodes
- Inodes and superblocks both marked dirty upon use

FS Organization



One Superblock per FS

Dirty list of inodes

Inodes and radix nodes/pages marked dirty separately

Simple Dirty 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
  - Task that runs in the kernel's address space
  - 2-8 threads, depending on how busy/idle threads are
- When pdflush runs
  - Kernel maintains a total number of dirty pages
  - Heuristics/admin configures a target dirty ratio (say 10%)
  - When pdflush wakes up
    - Figures out how many dirty pages are above target ratio
    - Determines target number of pages to write back
    - Writes back pages until it meets its goal or can't write more back
      - (Some pages may be locked, just skip those)

Writeback to Stable Storage

- We can find dirty pages in physical memory
- How does kernel know where on disk to write them?
  - And which disk for that matter?
- Superblock tracks device
- Inode tracks mapping from file offset to LBA
  - Note: this is FS’s inode, not VFS’s inode
  - Probably uses something like a radix tree