cs6465: Advanced Operating System Implementation
Lecture 04 – Linux Virtual Memory (one more time)

Anton Burtsev

September, 2024

- Physical memory
  - Tracking with an array of PageInfo structs
  - One element per physical page
- Virtual memory of a process
  - Track with virtual memory areas (VMAs)
- Physical memory in the kernel is mapped with a shift
  - V2P and P2V is simply a shift
  - For user memory one has to walk the page table to get the physical address of a page
  - Often hardware does it for you (on a page fault)

Address spaces

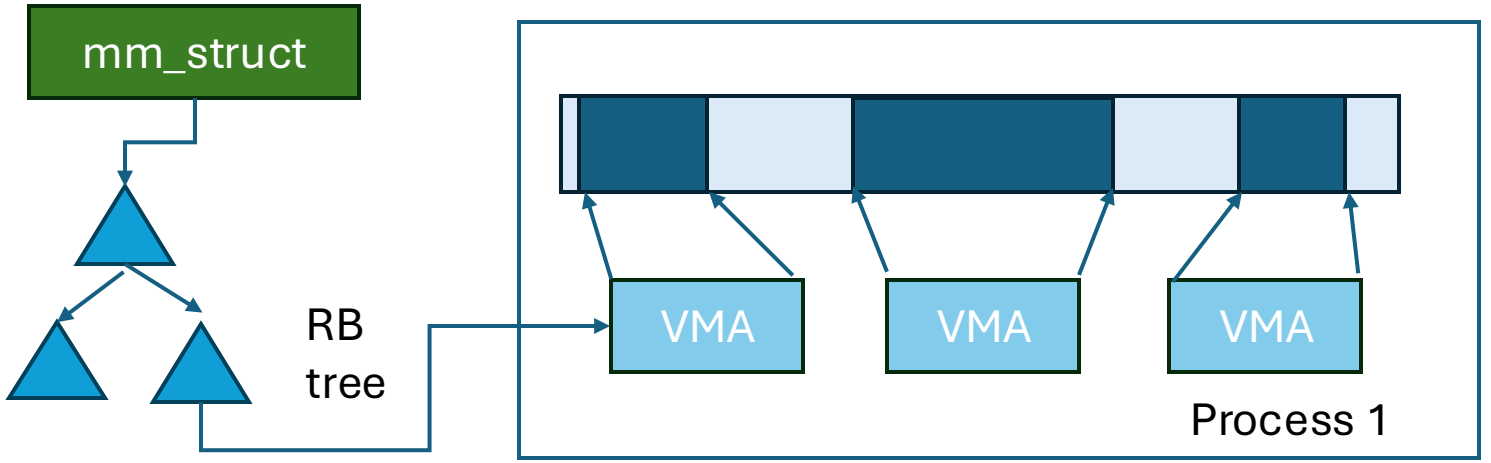
- Unifying abstraction:
  - Each file inode has an address space (0 - file size)
  - So do block devices that cache data in RAM (0---dev size)
- The (anonymous) virtual memory of a process has an
  - address space (0 - … on x86)
- In other words, all page mappings can be thought of
  - as and (object, offset) tuple
  - Make sense?

Address spaces for

- VMAs
- Files

Anonymous memory

- “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?
  - Just walk the page tables!
- Linux doesn’t do anything outside of the page tables to track this mapping

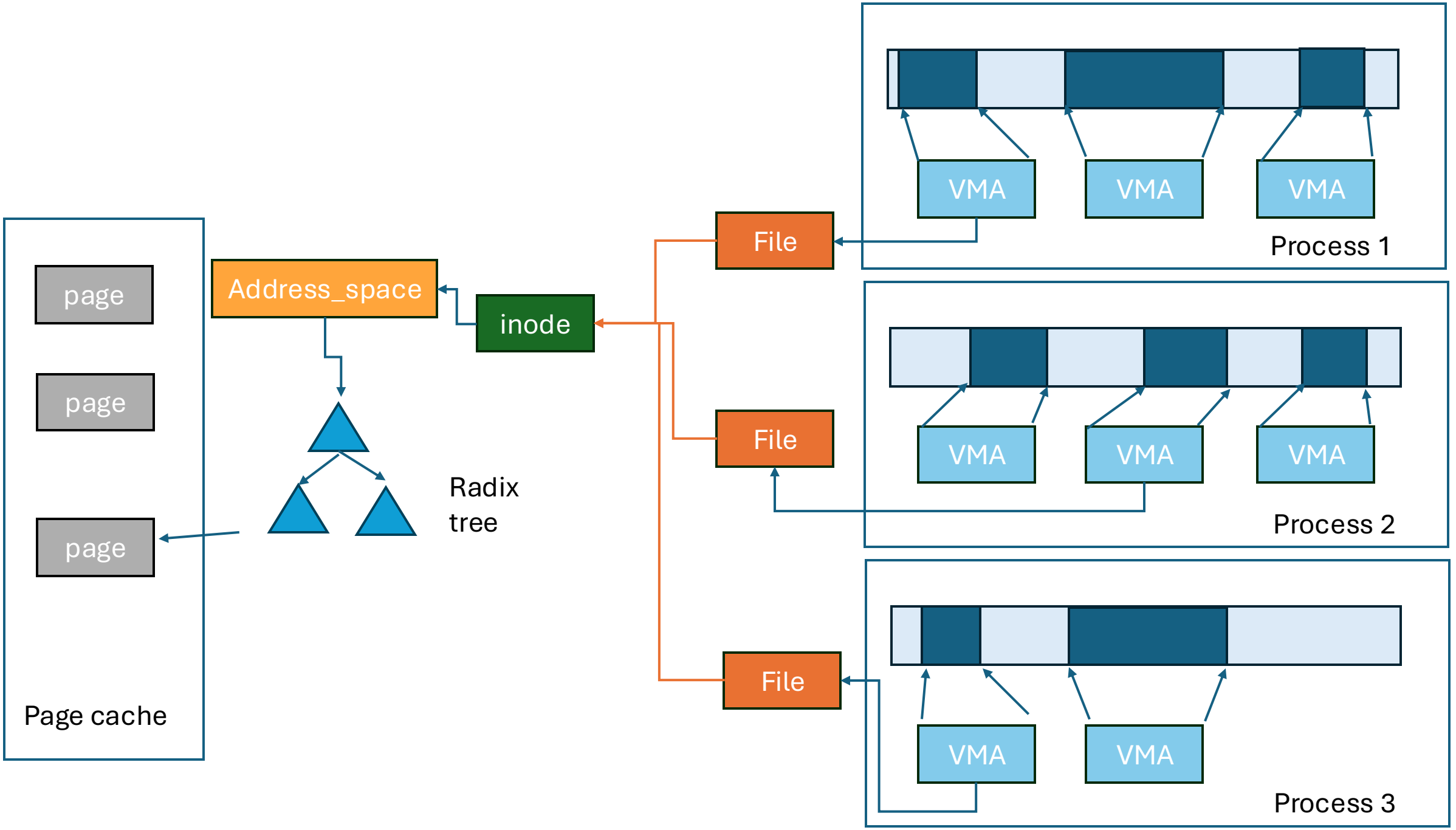

mm_struct
RB
tree
VMA
VMA
VMA
Process 1

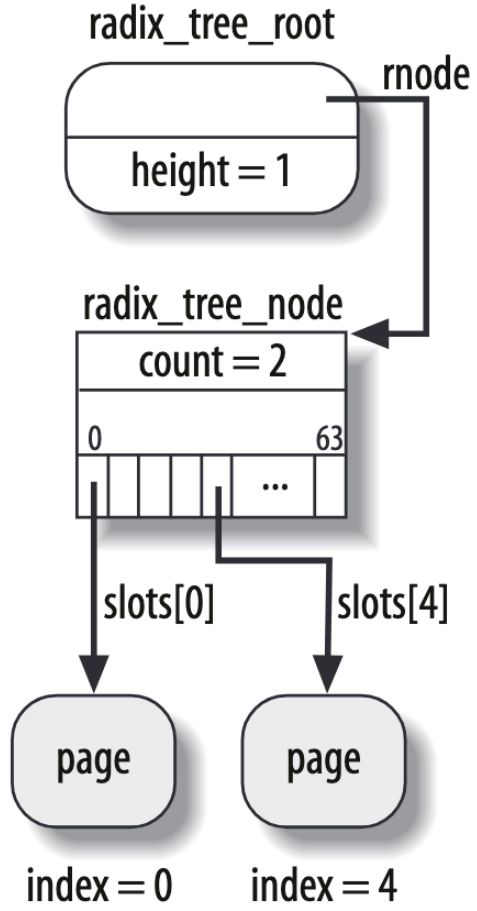
Files

- High-level goal:
  - Never load file in memory more than once
  - For read(), write() from multiple processes, as well as memory-mapped files

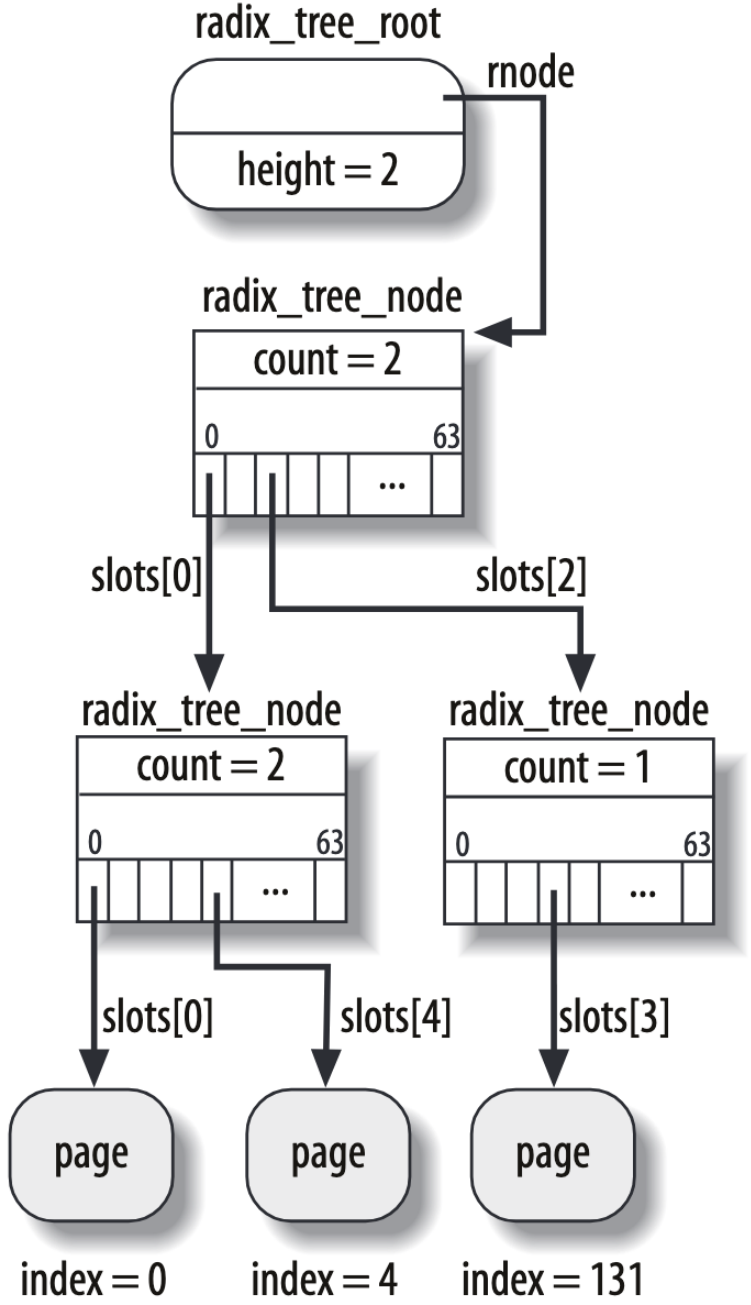
VMA to page

- VMA includes a file pointer and an offset into file
  - A VMA may map only part of the file
  - Offset must be at page granularity
  - Anonymous mapping: file pointer is null
- File pointer is an open file descriptor in the process file descriptor table


page
page
page
Page cache
Address_space
inode
Radix
tree
File
File
File
VMA
VMA
VMA
Process 1
VMA
VMA
VMA
Process 2
VMA
VMA
Process 3



(a) radix tree of height 1



(b) radix tree of height 2

- 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) = 64 GB
  - 5: 2^((6*5) + 12) = 4 TB

Dirty pages

Dirty pages

- Most OSes do not write file updates to disk immediately
  - OS tries to optimize disk arm movement
- OS instead tracks “dirty” pages
  - Ensures that write back isn’t delayed too long
  - Lest data can be lost in a crash
- Application can force immediate write back with sync system calls (and some open/mmap options)

Sync system calls

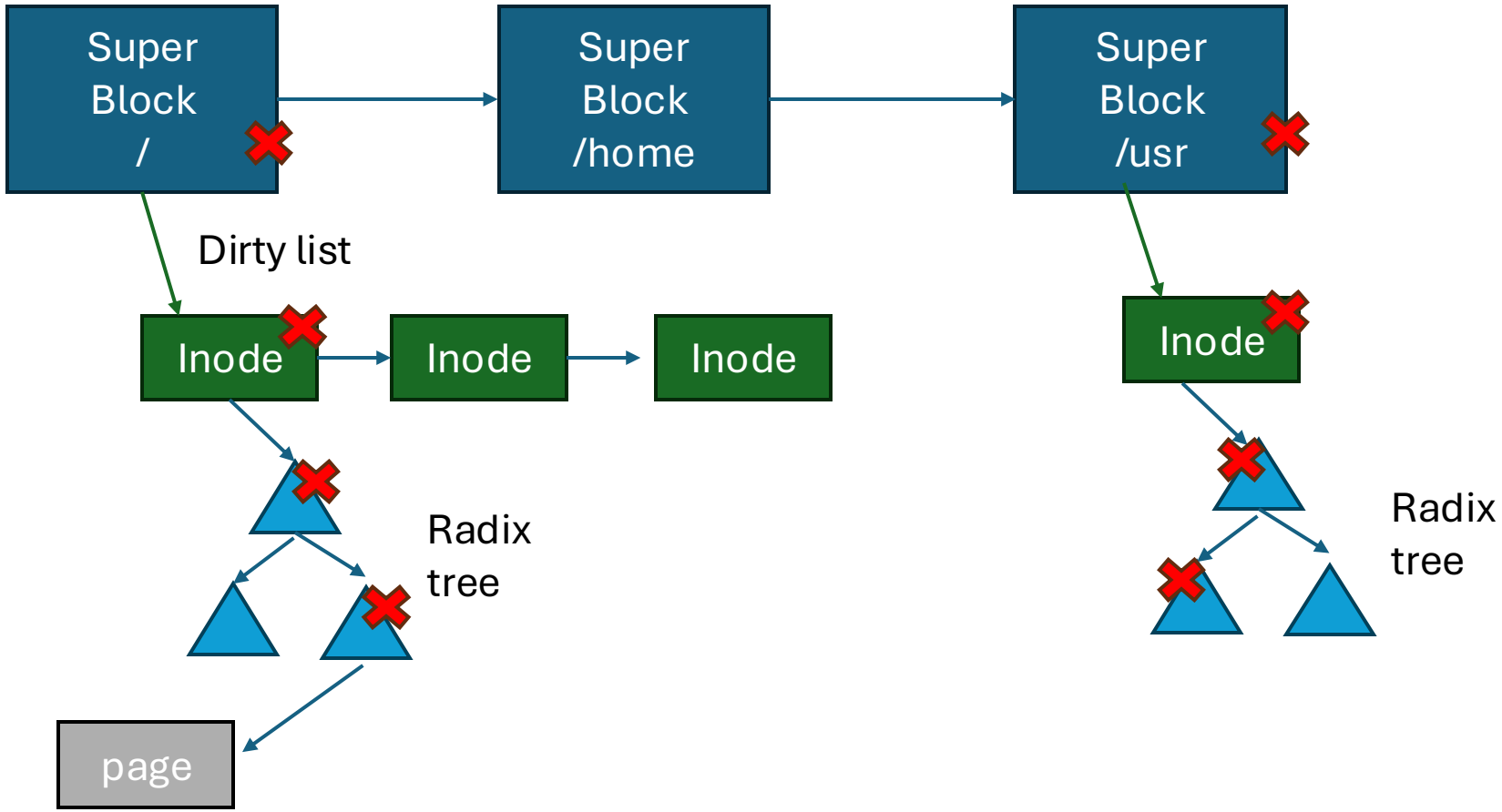
- sync() – Flush all dirty buffers to disk
- fsync(fd) – Flush all dirty buffers associated with this file to disk (including changes to the inode)
- fdatasync(fd) – Flush only dirty data pages for this file to disk
  - Don’t bother with the inode

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


Super Block /
Super Block /home
Super Block /usr
Dirty list
Inode
Inode
Inode
Inode
Radix tree
Radix tree
page

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
  - 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…

But where to write?

- Ok, so I see how to find the dirty pages
- How does the kernel know where on disk to write them?
  - And which disk for that matter?
- Superblock tracks device
  - Inode tracks mapping from file offset to sector

Block size mismatch

- Many disks have 512 byte blocks; pages are generally 4K
  - Many newer 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

Buffer head

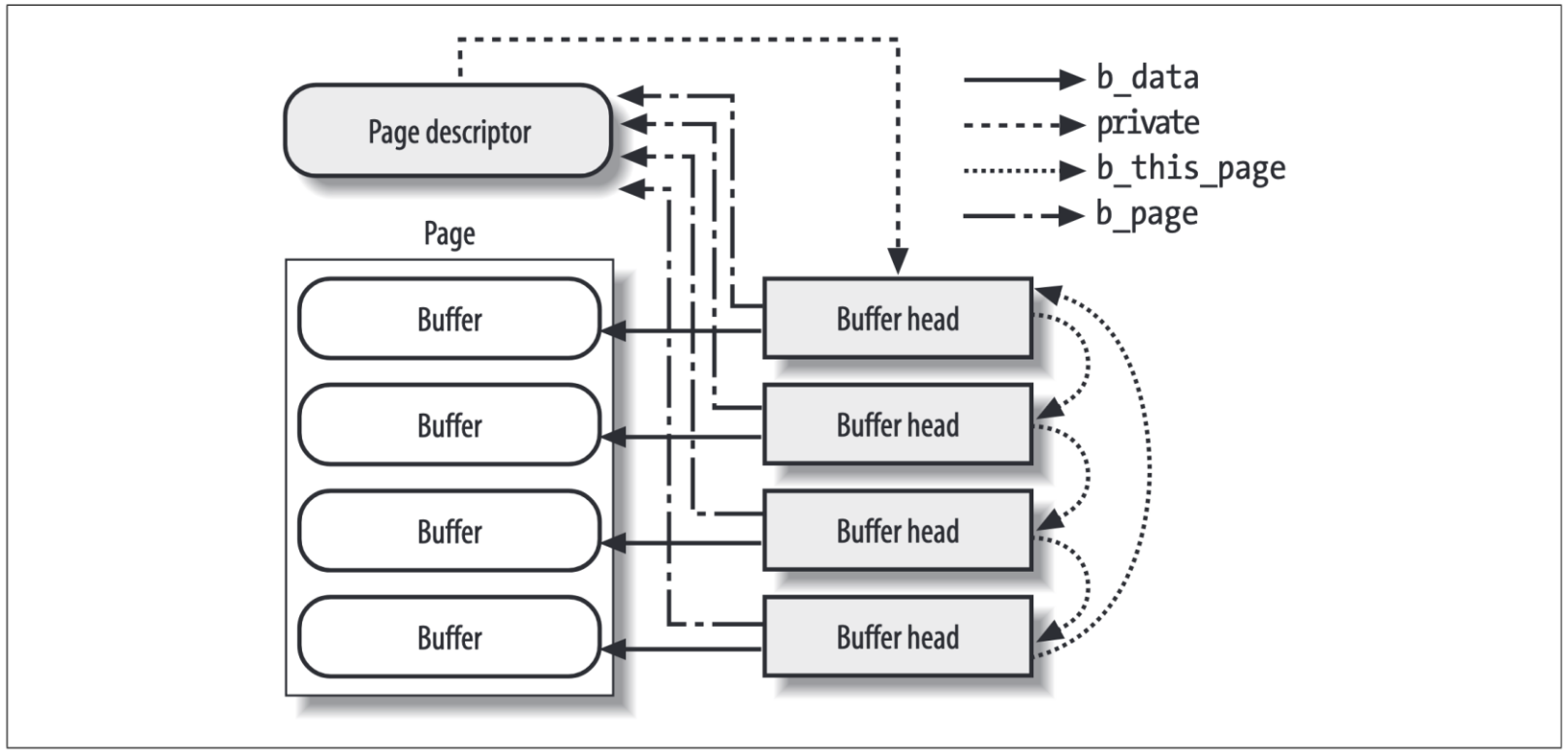


Figure 15-2. A buffer page including four buffers and their buffer heads

- Also used to implement legacy buffer-cache

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