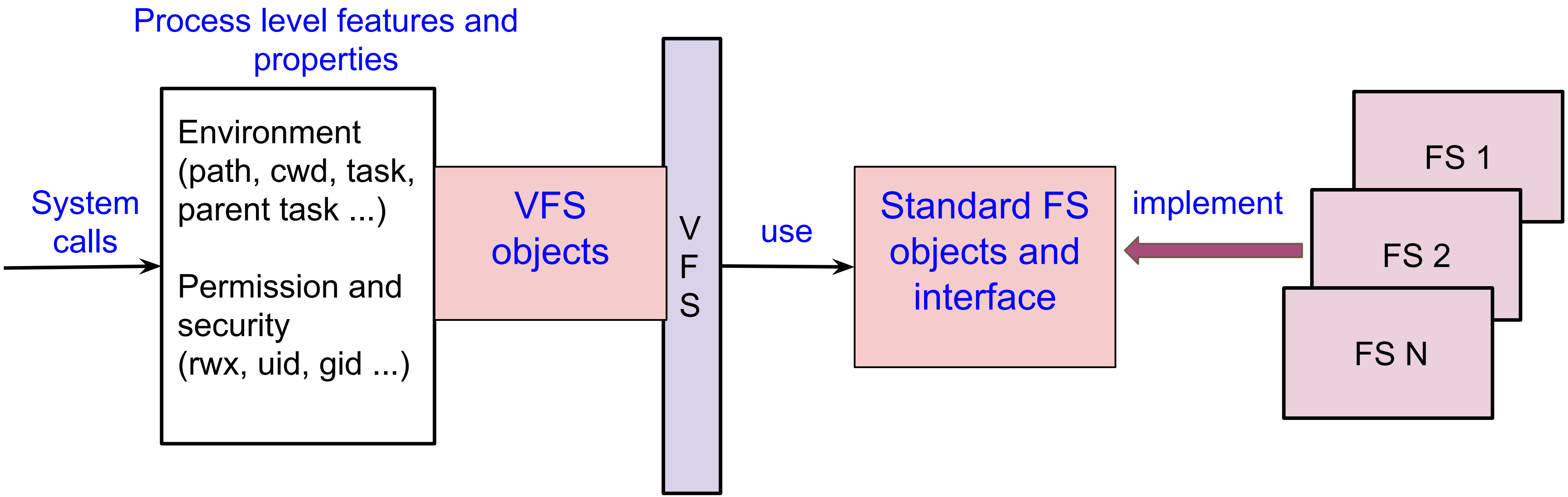
Linux Filesystem Interfacing

Recap: Linux virtual file system (VFS)



- ➔ User space can use the same API
- ➔ Implementation details encapsulated

Reading file blocks from disk

User process
read(fd, buffer, length)

↓

VFS
fs_read(?, ?, ?, ? ….)

↓

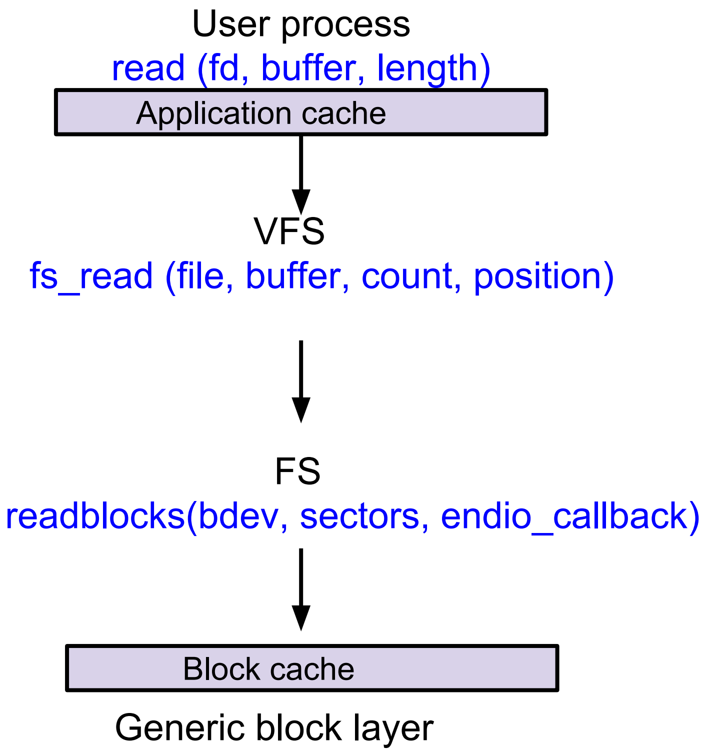
FS
readblocks(?, ?, ?, ? ...)

↓

Generic block layer

- What are
  - VFS responsibilities
  - FS operations
  - Block device operations
- Read destination buffer
  - Direct to user buffer
  - Kernel memory → user buffer

Reading file blocks from disk



User process

read (fd, buffer, length)

Application cache

VFS

fs_read (file, buffer, count, position)

FS

readblocks(bdev, sectors, endio_callback)

Block cache

Generic block layer

➔ Heard some form of caching---buffer cache, page cache?

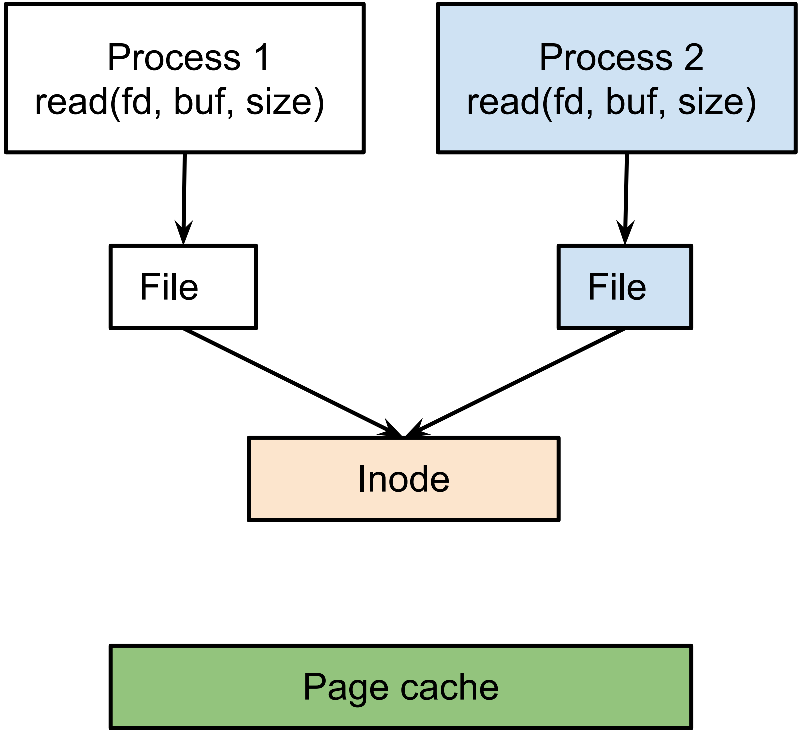
➔ Why cache?

➔ Where to cache? What is the design rationale?

Caching: where and why?

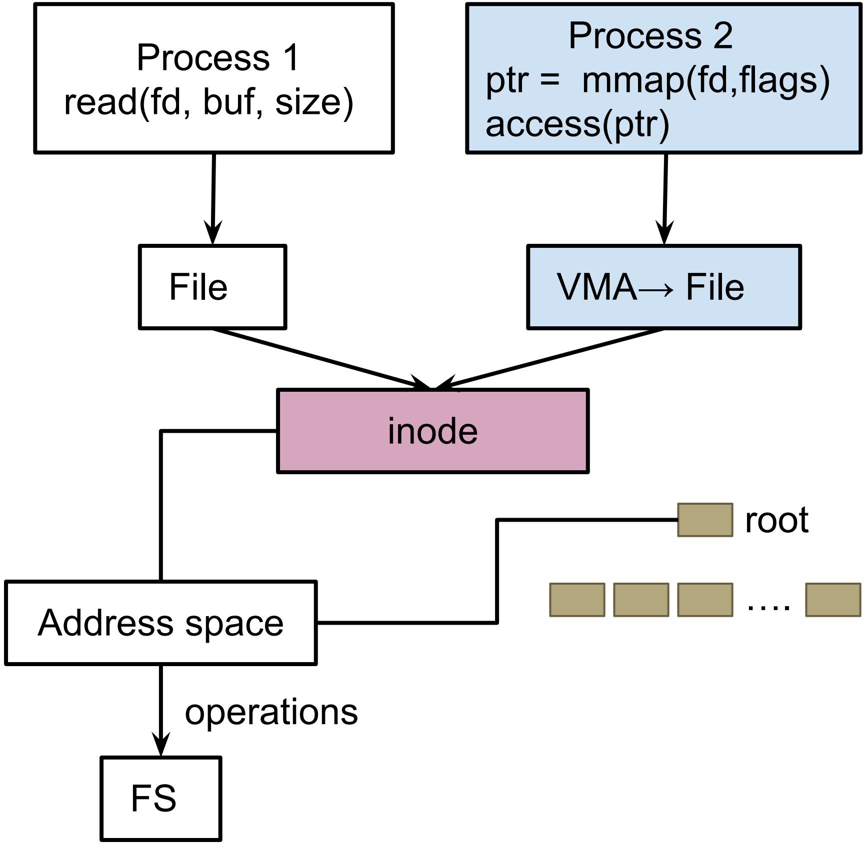
- Caching between VFS and FS layer
  - Linux page cache
  - What are the key design goals?
  - Advantages and disadvantages
  - Cache management: eviction, dirty block writing
- Block layer caches
  - Why not cache at block level?
  - Simpler design
  - System-wide applicability
- Hybrid design: Linux unified caching

Linux page cache: the gateway



- ➔ Requirement: File block lookup at different offsets
  - ◆ File size can range from very small to huge
- ➔ Entry point to the cache, File or Inode?
- ➔ Recall mmap-ing a file creates a VMA struct
- ➔ Should handle both file I/O and page faults

Address spaces



- ➔ A per inode cache
  - ◆ Lookup, insert, evict, dirty-flush
- ➔ Accessible from both file struct and vma struct
- ➔ Radix tree
  - ◆ Root pointed by address space struct
  - ◆ Operations at a page size (4K) granularity
  - ◆ Complexity, dynamic expansion (homework)

Page cache and memory management

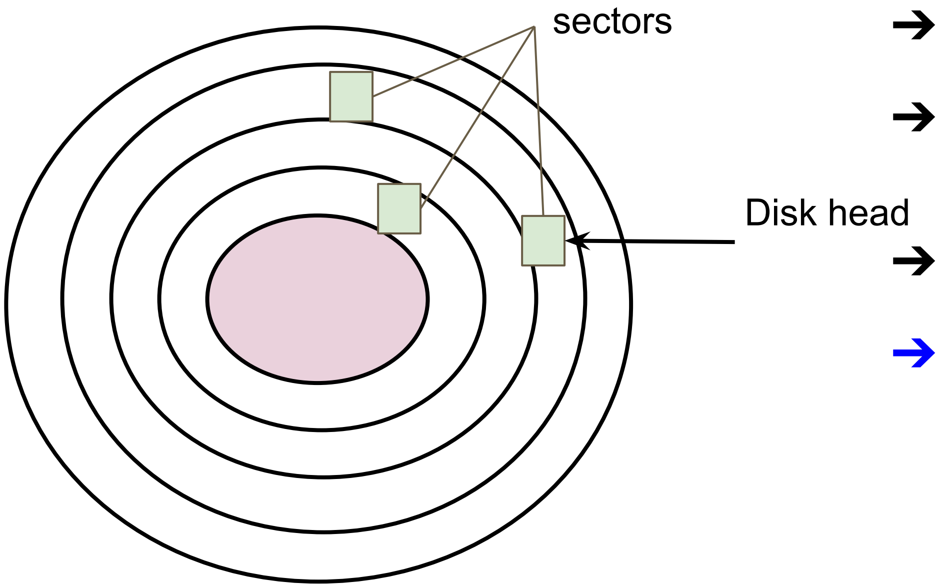
- Any limit on memory used for page caches?
- When low on memory
    - Some page frames need to be freed
    - Page cache pages: throw-away (clean) vs. require sync (dirty)
- When selecting a page to evict
    - For file backed page, how to get a handle on address space from a page?
    - How to address similar requirement for anonymous memory?

- Page reclamation
    - File backed pages and anonymous pages
    - Algorithm: ClockPro approximation (homework)

The generic block I/O interface: notes

- Device block size can be less than page size (4K)
  - Scenario: writing to one block (e.g., 512 bytes)
  - Solutions?
- Reordering I/O requests and correctness
  - Why reorder? Will come back to this in the next slide ...
  - Is it always safe to reorder?
- I/O finish call back
  - Required both for read and write, why?

Disk schedulers



- ➔ Why should it not be just FCFS?
- ➔ Given a set of block I/O requests
  - ◆ Why not a greedy scheme?
- ➔ Design objectives
- ➔ Linux provides a pluggable architecture: you can write one !

Elevator scheduler

- ➔ Disk head moves like an elevator
  - ◆ Between ground floor (outer track) and highest floor (inner track)
  - ◆ Reorder I/O requests depending on current head position and movement direction
- ➔ Advantages/Disadvantages
  - ◆ Starvation?
  - ◆ Fairness?
  - ◆ What about throughput? (rotational delay)

Fairness across processes: CFQ

- Process-level fairness
  - Per-process queue before the actual scheduling
  - Round-robin selection of disk I/O requests from process level queues
- Advantages/Disadvantages
  - Starvation?
  - Fairness?
  - What about throughput?

Some other disk schedulers

- Deadline
  - Associate a deadline with each request and reorder
- Anticipatory
  - Delay I/O requests from a process for improved batching
- And many more …
- If you want to design a new one
  - Goals and objectives
  - Device characteristics (e.g., SSDs require no seek)
  - Application requirements
  - Correctness issues

Conclusion

- ➔ Page cache: a huge part of memory management
- ➔ Many complexities lie in the details
- ➔ Next class: FIle system case study (ext4)