UNIVERSITY OF OREGON

CIS 415: Operating Systems

Filesystem Implementation

Spring 2014
Prof. Kevin Butler

- Last class:
  - File System Interface
- Today:
  - File System Implementation

Protection vs Security

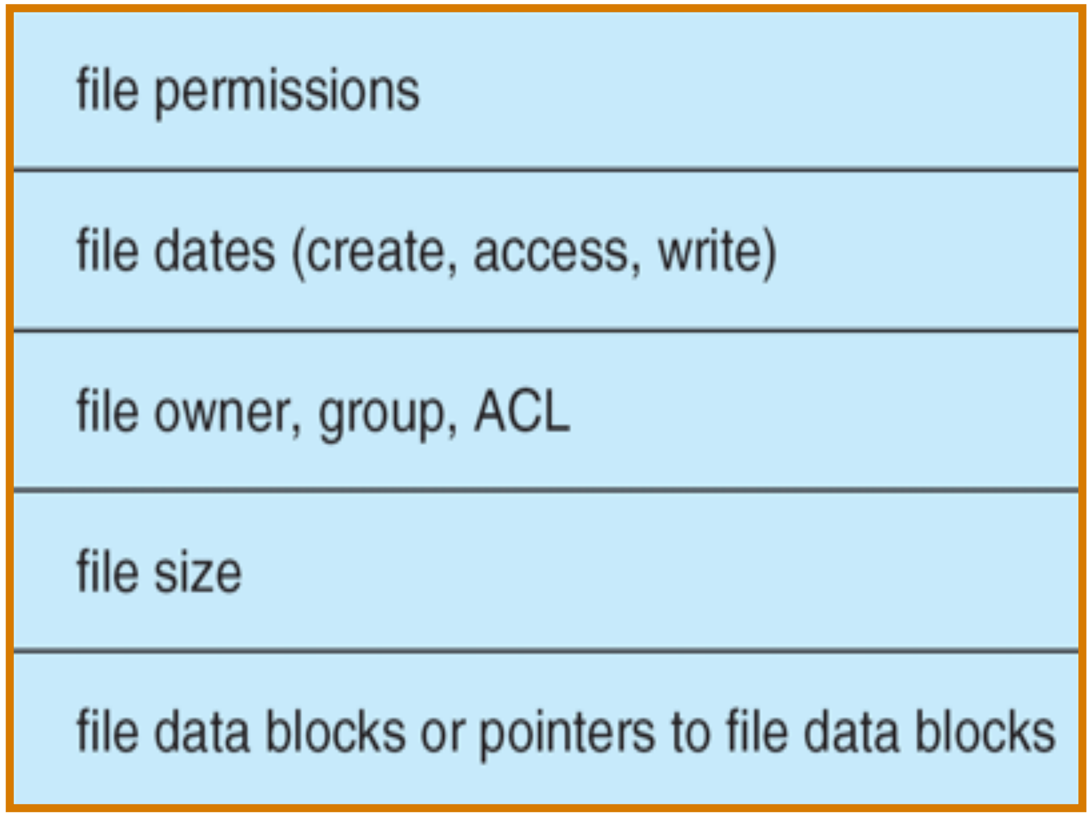
- Protection
  - Security goals met under *trusted* processes
  - Protects against an error by a non-malicious entity
- Security
  - Security goals met under *potentially malicious* processes
  - Protects against any malicious entity
  - Hence, For J:
    - Non-malicious process shouldn't leak the private key by writing it to O_3
    - A potentially malicious process may contain a Trojan horse that can write the private key to O_3

Disks as Secondary Store

- What makes them convenient for storing files?
  - Rewrite in place
    - Read, modify in memory, write back to original location
  - Access any block (sequentially or random)
    - Gotta move the head to get there
    - More detail next time

File Control Block

- Logical file system's representation of a file -- stored on disk
  - In UNIX, called an *inode*



file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Filesystem Structures

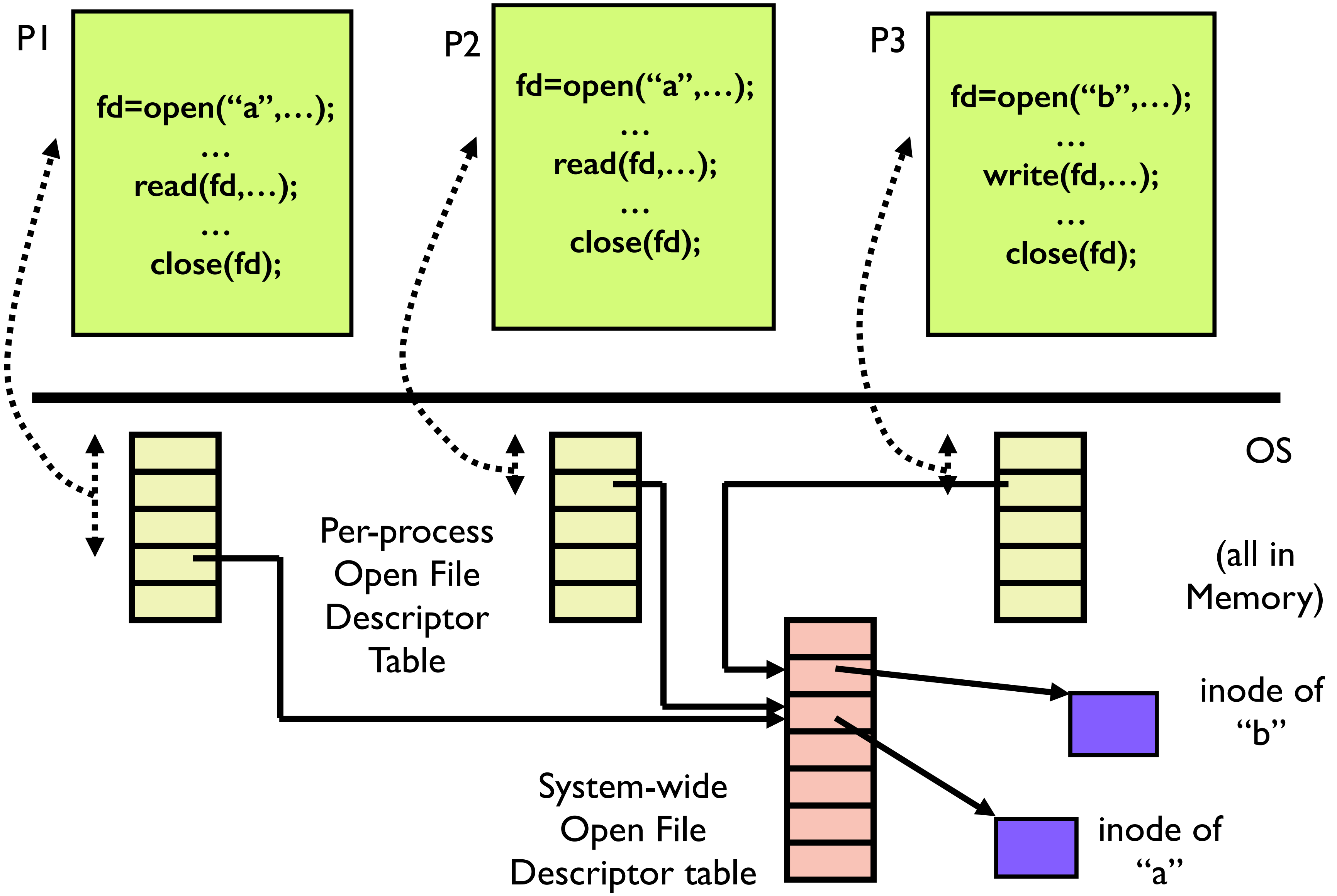
- On-disk structures
  - Boot control block
    - Info to boot the OS from the disk
    - Typically, the first block of the boot partition (*boot block*)
  - Partition control block
    - Info about a file system (number of blocks -- fixed size)
    - Includes free block information (*superblock*)
  - Directory Structure
  - File Control Blocks

Filesystem Structures

- In-memory structures
  - Partition table
    - Current mounted partitions
  - Directory structure
    - Information on recently accessed directories
  - System-wide, open file table
    - All currently open files
  - Per-process, open file table
    - All of a given process's open files

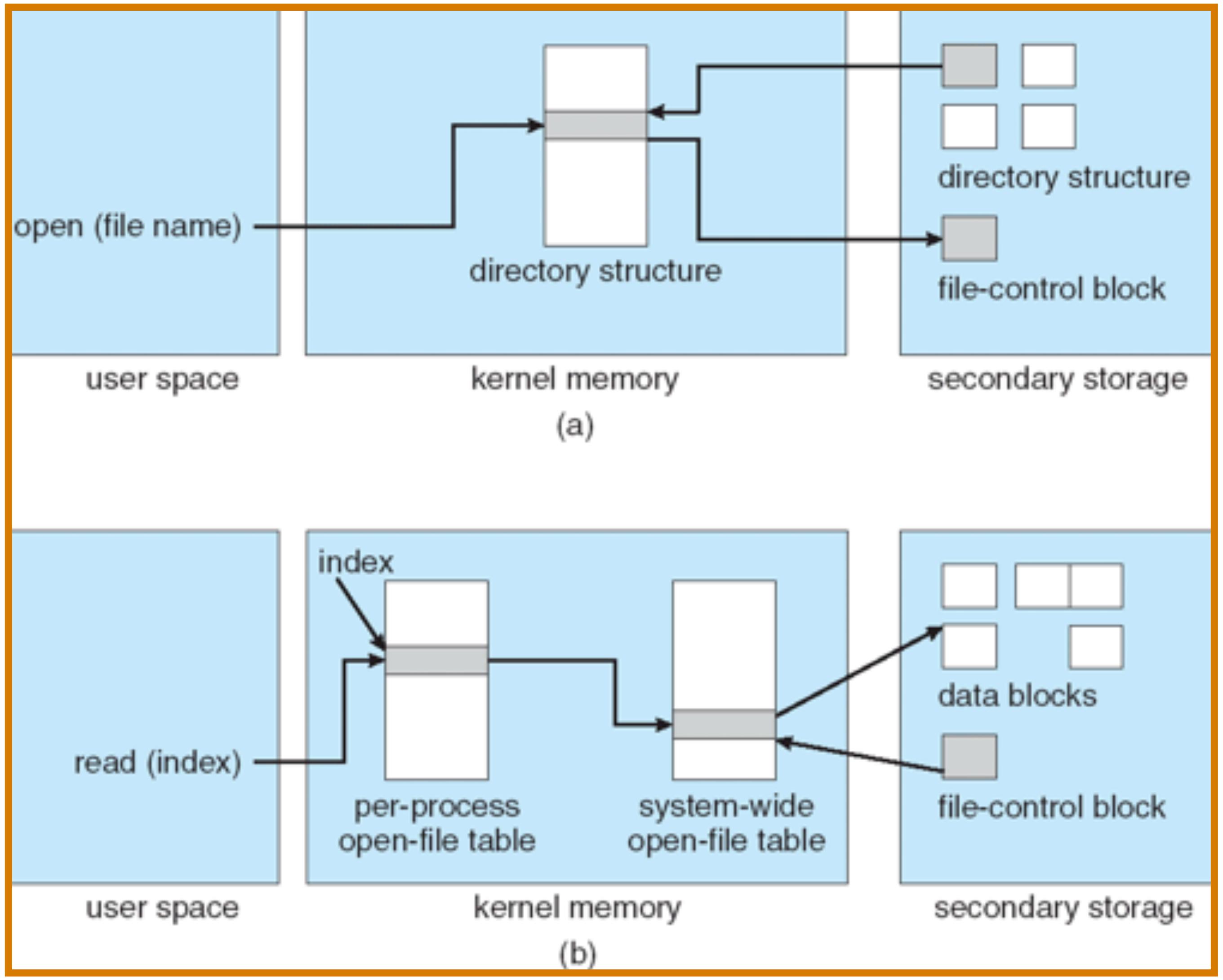
In-memory structures

For open( ) syscall:



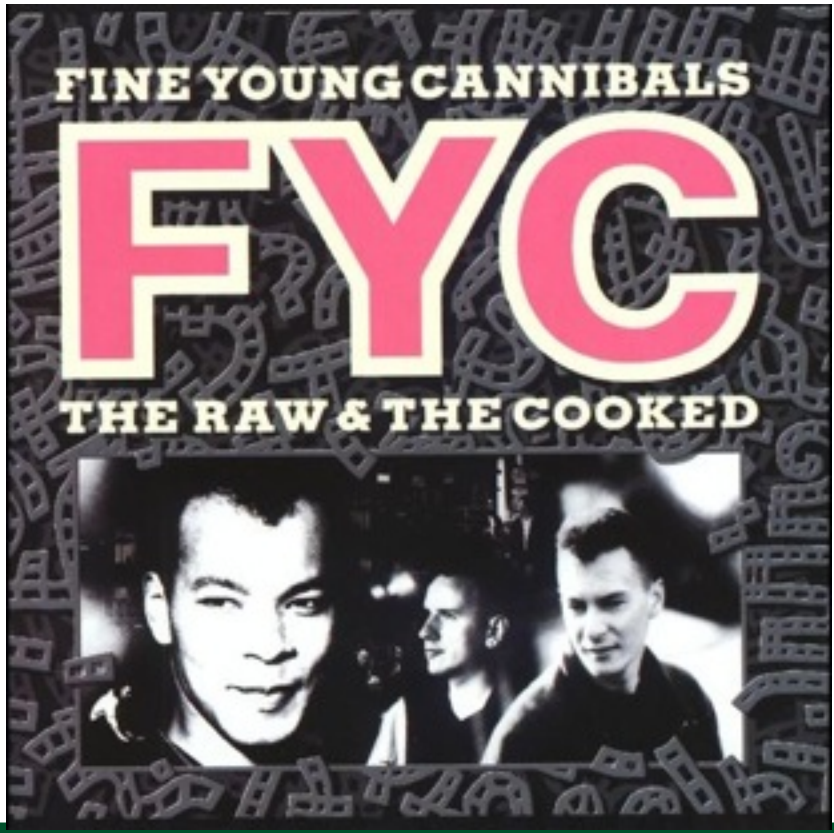
In-Memory File Structures

- To open and read a file
  - Index is a file descriptor



Partition Structures

- Layout of a file system on a disk
  - Raw: no file system structure
    - E.g., swap space, database
  - Cooked: a file system structure
    - E.g., directories



Boot Partition

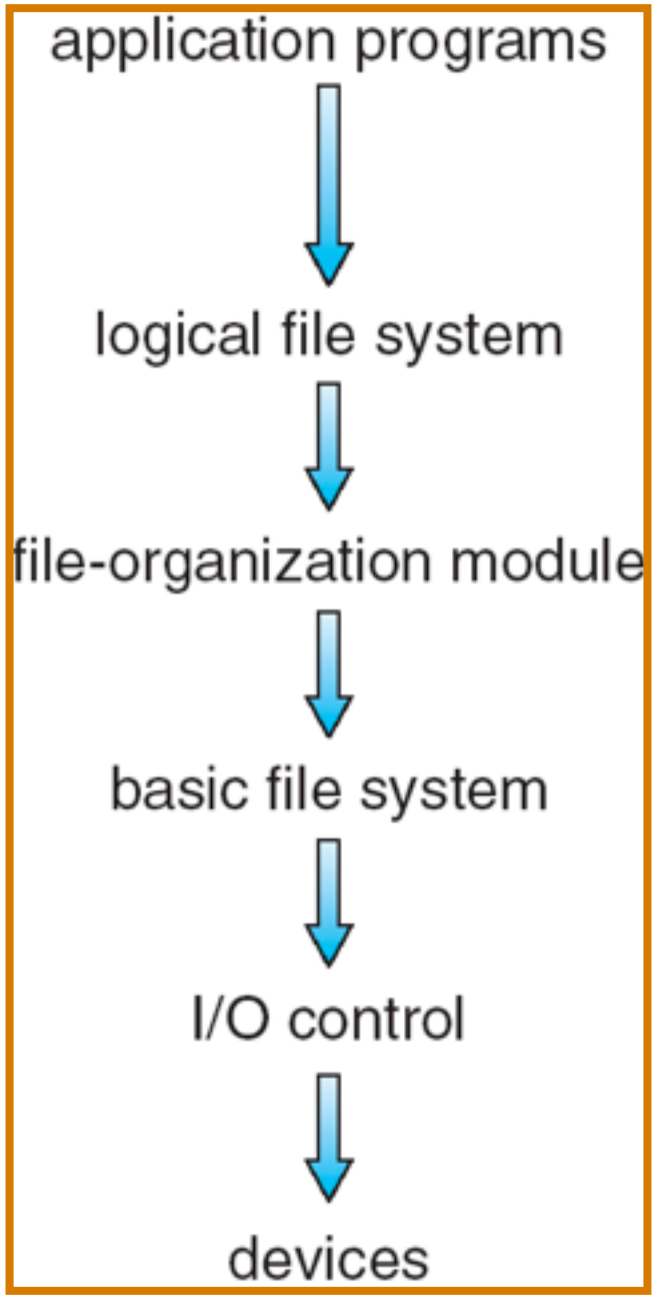
- Contains information to boot the system
  - Image to be loaded at boot time
- May boot more than one system
  - How is this done?
  - Bootloader is booted first (GRUB)
    - The you choose the OS to boot

Mount Table

- OS has an in-memory table to store…
  - Each file system that has been mounted
    - A file system (partition) is a device in UNIX
  - Where it is mounted
    - A directory
  - The type of the file system
    - Physical file system (e.g., ext3, ntfs, …)
  - Some other attributes
    - E.g., Read-only

File System Layers

UNIVERSITY OF OREGON



Virtual File System

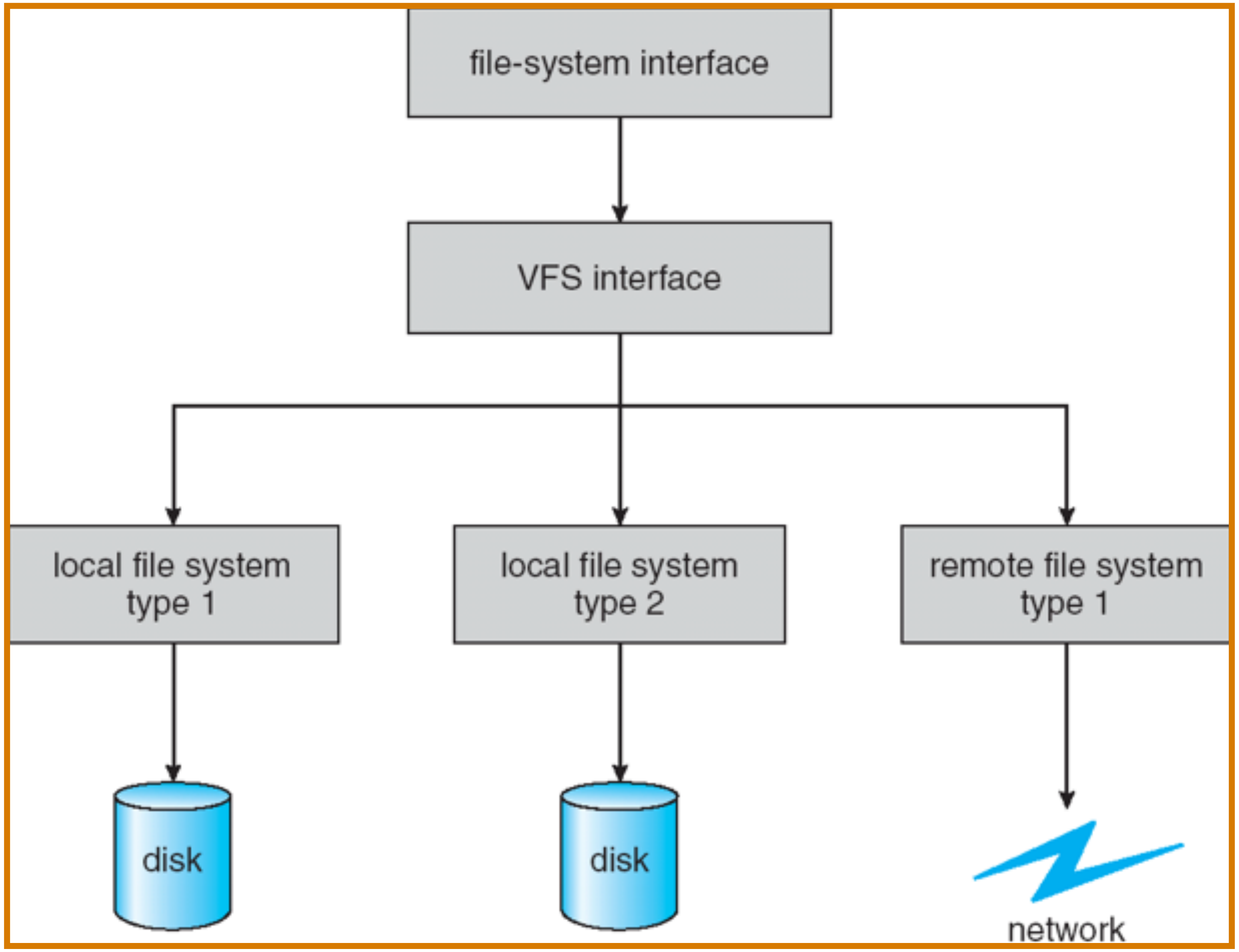
Physical File System

Device Drivers

Virtual File System

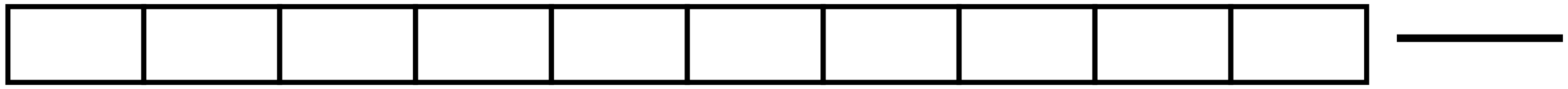
- File systems have general and storage method-dependent parts
  - *Virtual file system* is specific to the OS
    - File system-generic operations
    - Works with inodes (FCB), files, directories, superblocks (partitions)
    - The stuff that we have discussed
  - *Physical file system* is specific to how secondary storage will be used to manage data
    - Converts the objects above into blocks

Virtual File System



Filesystem Implementation

- View the disk as a logical sequence of blocks
- A block is the smallest unit of allocation.
- Issues:
  - How do you assign the blocks to files?
  - Given a file, how do you find its blocks?



File Allocation

UNIVERSITY OF OREGON

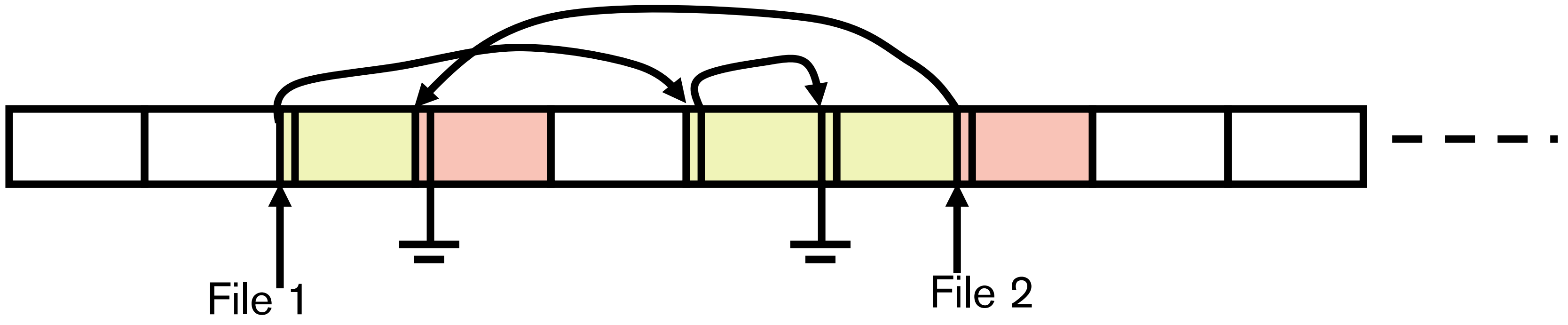
- Direct access of disks gives us flexibility in implementing files
    - Relate to memory management problem
- Choices
    - Contiguous
    - Non-contiguous
        - Linked
        - Indexed

Contiguous Allocation

- Allocate a sequence of contiguous blocks to a file.
- Advantages:
  - Need to remember only starting location to access any block
  - Good performance when reading successive blocks on disk
- Disadvantages:
  - File size has to be known *a priori*.
  - External fragmentation

Linked List Allocation

- Keep a pointer to first block of a file.
- The first few bytes of each block point to the next block of this file.
- Advantages: No external fragmentation
- Disadvantages: Random access is slow!



Linked List Using Index

- In the previous scheme, we needed to go to disk to chase pointers since memory cannot hold all the blocks.
- Why not remove the pointers from the blocks, and maintain the pointers separately?
- Perhaps, then all (or most) of the pointers can fit in memory.
- Allocation is still done using linked list.
- However, pointer chasing can be done "entirely" in memory.

Blocks & Pointers

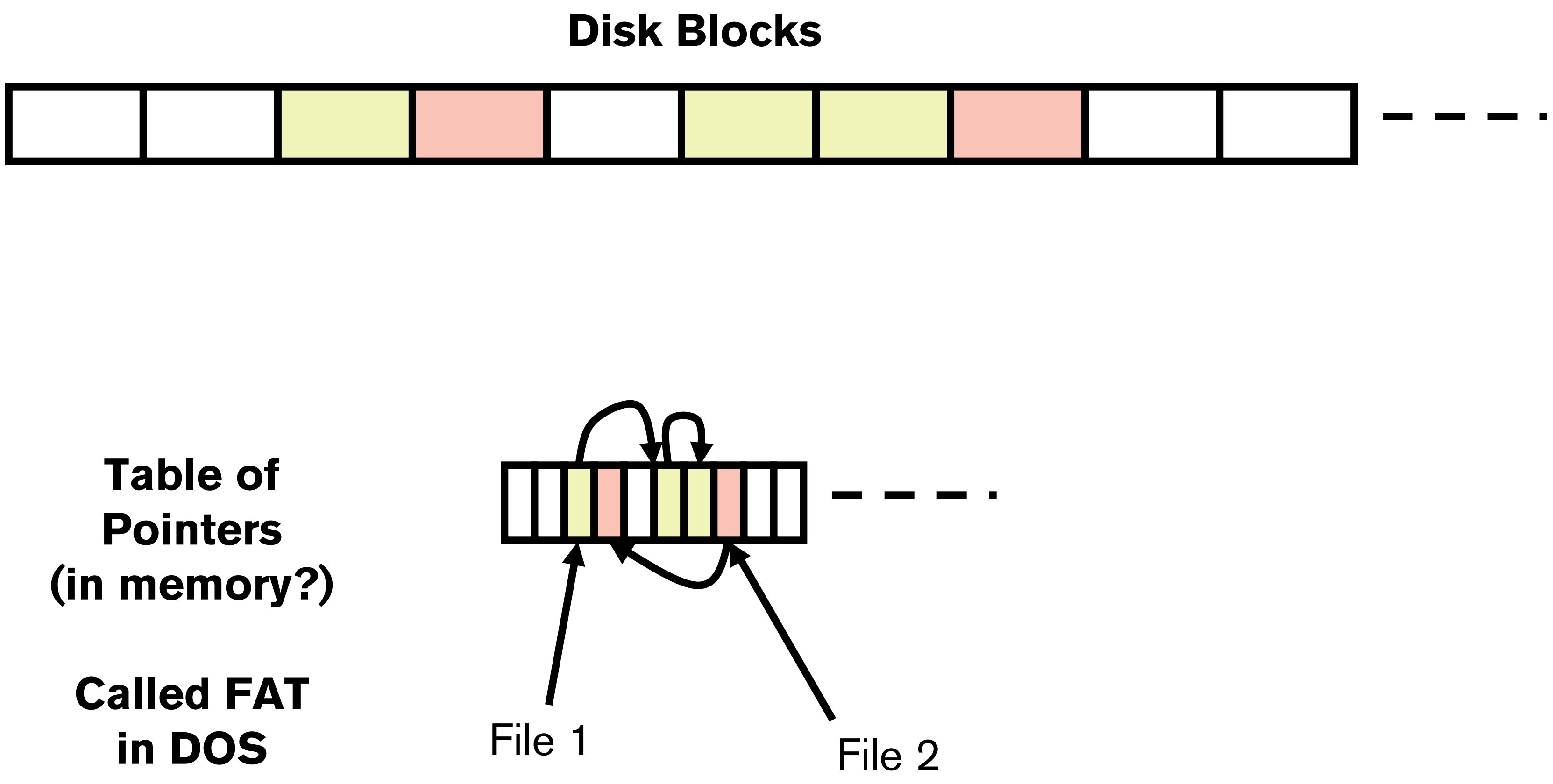


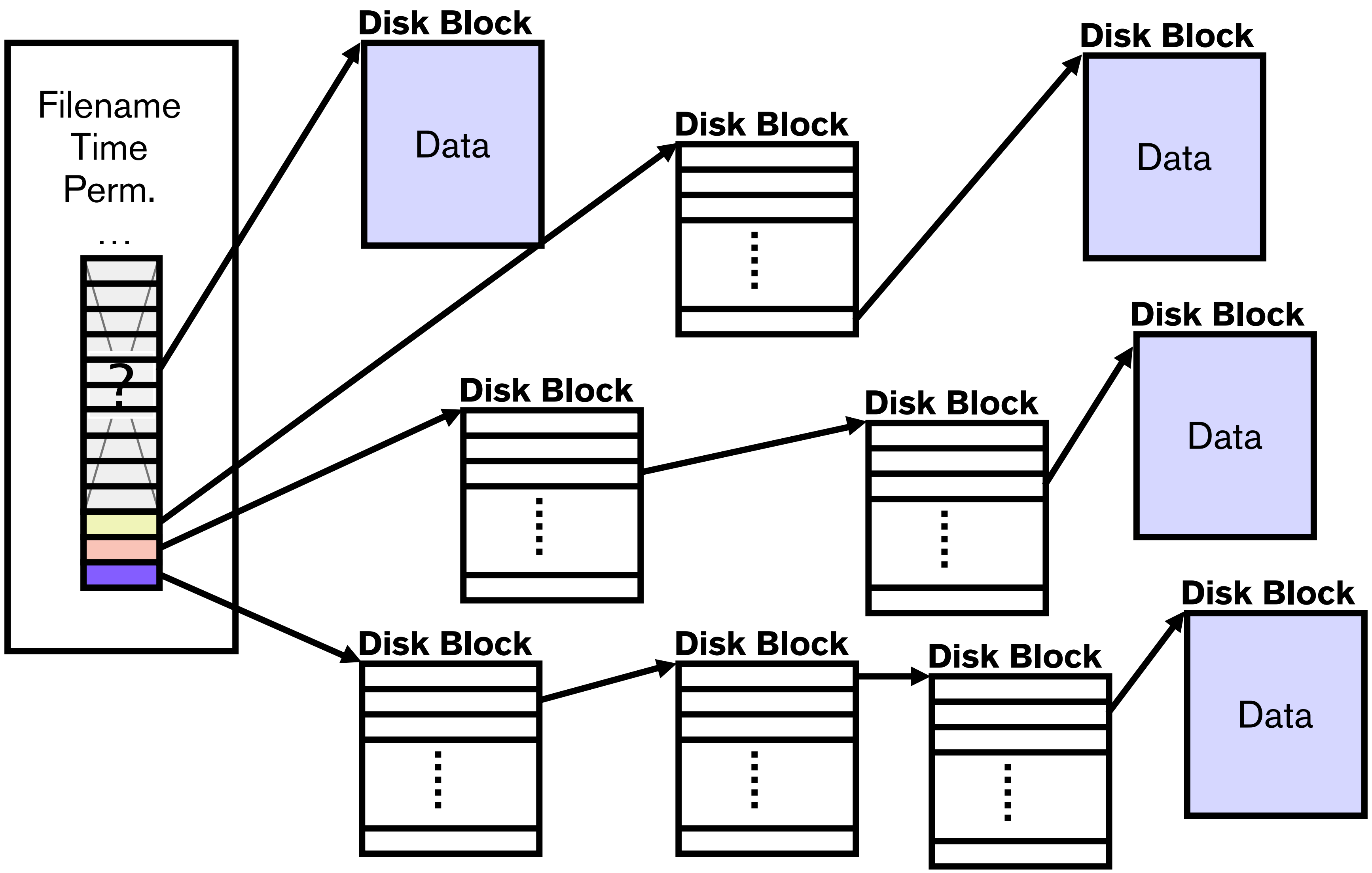
Table of Pointers (in memory?)

Called FAT in DOS

Indexed Allocation

- For each file, direct pointers to all its blocks.
- However, the number of pointers for a file can itself become large.
- UNIX uses *inodes*.
- An inode contains:
  - File attributes (time of creation, permissions, ….)
  - 10 direct pointers (logical disk block ids)
  - 1 one-level indirect pointer (points to a disk block which in turn contains pointers)
  - 1 two-level indirect pointer (points to a disk block of pointers to disk blocks of pointers)
  - 1 three-level indirect pointer (points to a disk block of pointers to disk blocks of pointers to pointers of disk blocks)

Inodes


Filename
Time
Perm.
...
?
Disk Block
Data
Disk Block
Disk Block
Data
Disk Block
Disk Block
Data
Disk Block
Disk Block
Disk Block
Disk Block
Data

Tracking free blocks

- List of free blocks
  - bit map: used when you can store the entire bit map in memory.
  - linked list of free blocks
    - each block contains ptrs to free blocks, and last ptr points to another block of ptrs. (in UNIX).
    - Pointer to a free FAT entry, which in turn points to another free entry, etc. (in DOS)

- Exercise: Given that the FAT is in memory, find out how many disk accesses are needed to retrieve block "x" of a file from disk. (in DOS)

- Exercise: Given that the inode for a file is in memory, find out how many disk access are needed to retrieve block "x" of this file from disk. (in UNIX)

Finding Files

- Now we know how to retrieve the blocks of a file once we know:
  - The FAT entry for DOS
  - The inode of the file in UNIX

- But how do we find these in the first place?
  - The directory where this file resides should contain this information

Directory

- Contains a sequence (table) of entries for each file.
- In DOS, each entry has
  - [Fname , Extension , Attributes , Time , Date , Size , First Block #]
- In UNIX, each entry has
  - [Fname, inode #]

Accessing a file block with DOS

- Go to "\" FAT entry (in memory)
- Go to corresponding data block(s) of "\" to find entry for "a"
- Read 1st data block of "a" to check if "b" present. Else, use the FAT entry to find the next block of "a" and search again for "b", and so on. Eventually you will find entry for "b".
- Read 1st data block of "b" to check if "c" present...
- Read the relevant block of "c", by chasing the FAT entries in memory.

Accessing File Block in UNIX

- Get "/" inode from disk (usually fixed, e.g. #2)
- Get block after block of "/" using its inode until entry for "a" is found (gives its inode #).
- Get inode of "a" from disk
- Get block after block of "a" until entry for "b" is found (gives its inode #)
- Get inode of "b" from disk

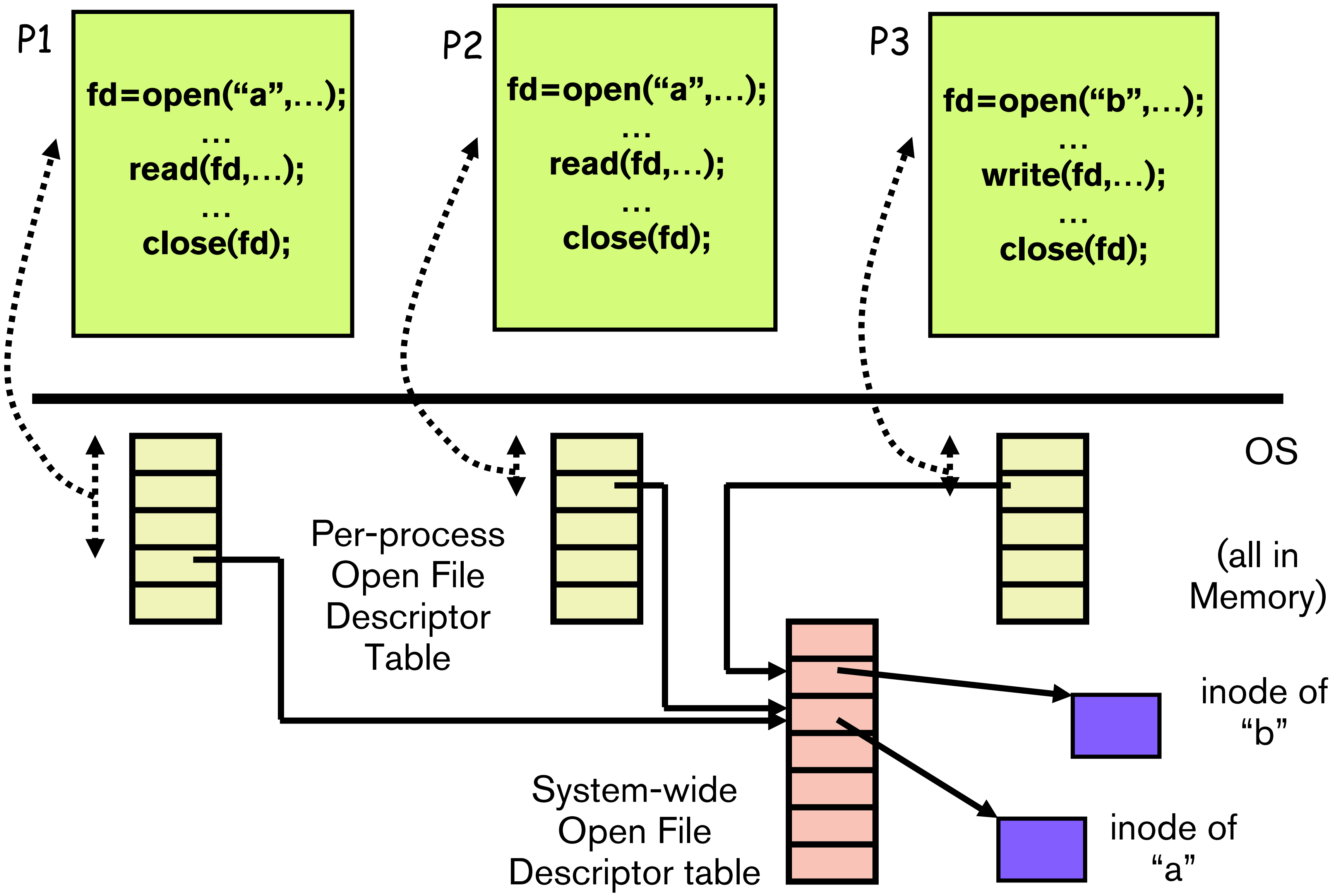
Accessing File Block in UNIX

- Get block after block of "b" till entry for "c" is found (gives its inode #)
- Get inode of "c" from disk
- Find out whether block you are searching for is in 1st 10 ptrs, or 1-level or 2-level or 3-level indirect.
- Based on this you can either directly get the block, or retrieve it after going through the levels of indirection.

Inode Caching

- Imagine searching through the inodes each time you do a `read()` or `write()` on a file
- Too much overhead!
- However, once you have the inode of the file (or a FAT entry in DOS), then it is fairly efficient!
- You want to cache the inode (or the ID of the FAT entry) for a file in memory and keep re-using it.
- Open file descriptor table kept in memory does this for us

open() syscall



P1

```
fd=open("a",…);
…
read(fd,…);
…
close(fd);
```

P2

```
fd=open("a",…);
…
read(fd,…);
…
close(fd);
```

P3

```
fd=open("b",…);
…
write(fd,…);
…
close(fd);
```

OS

(all in Memory)

Per-process Open File Descriptor Table

System-wide Open File Descriptor table

inode of "b"

inode of "a"

File Caching

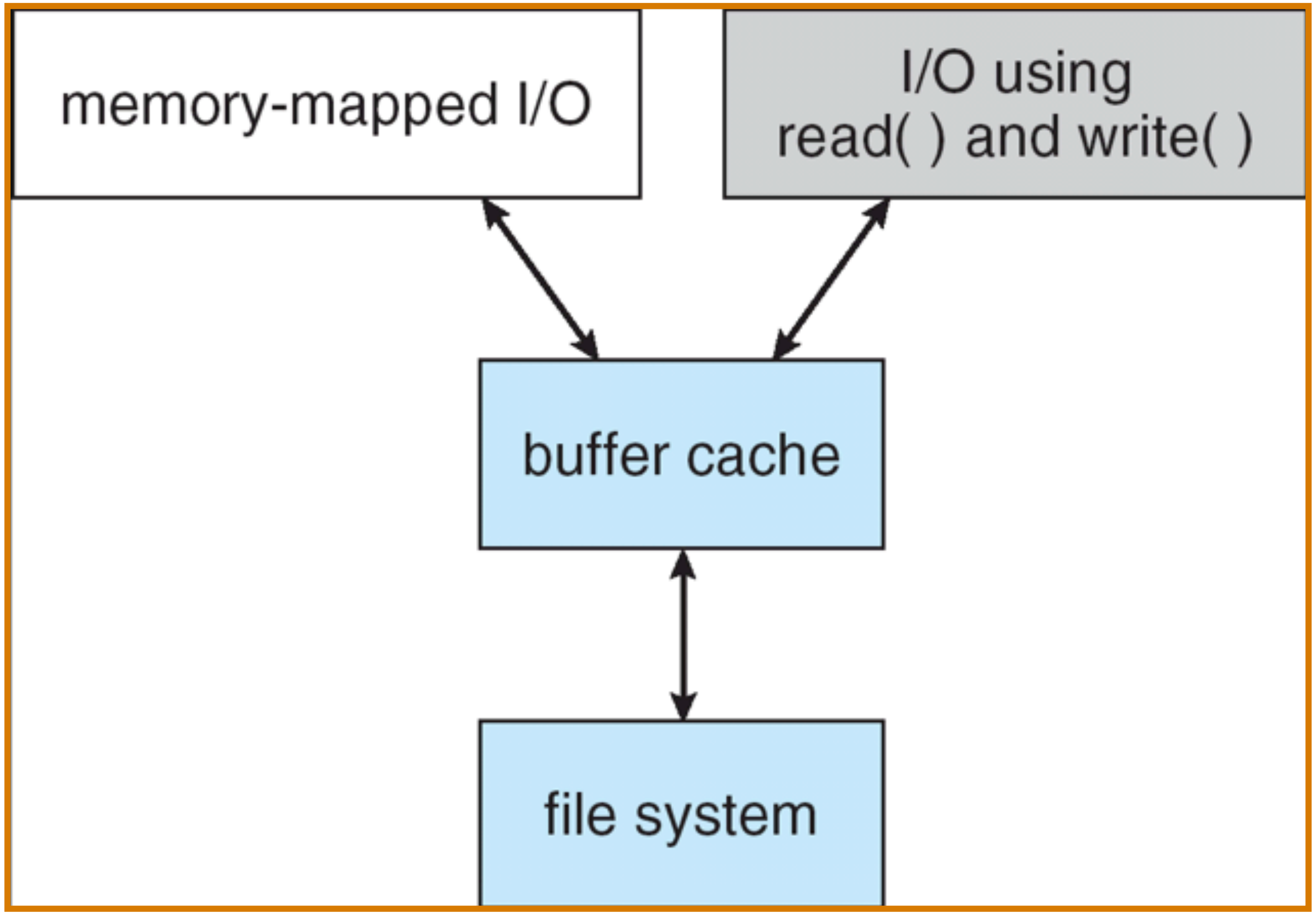
- Even if after all this (i.e. bringing the pointers to blocks of a file into memory), may not suffice since we still need to go to disk to get the blocks themselves.

- How do we address this problem?
  - Cache disk (data) blocks in main memory
  - Called *file caching*

File Caching/Buffering

- Cache disk blocks that are in need in physical memory.
- On a `read()` system call, first look up this cache to check if block is present.
  - This is done in software
  - Look up is done based on logical block id.
  - Typically perform some kind of hashing
- If present, copy this from OS cache/buffer into the data structure passed by user in the `read()` call.
- Else, read block from disk, put in OS cache and then copy to user data structure.

File Caching/Buffering



Filesystem Caching

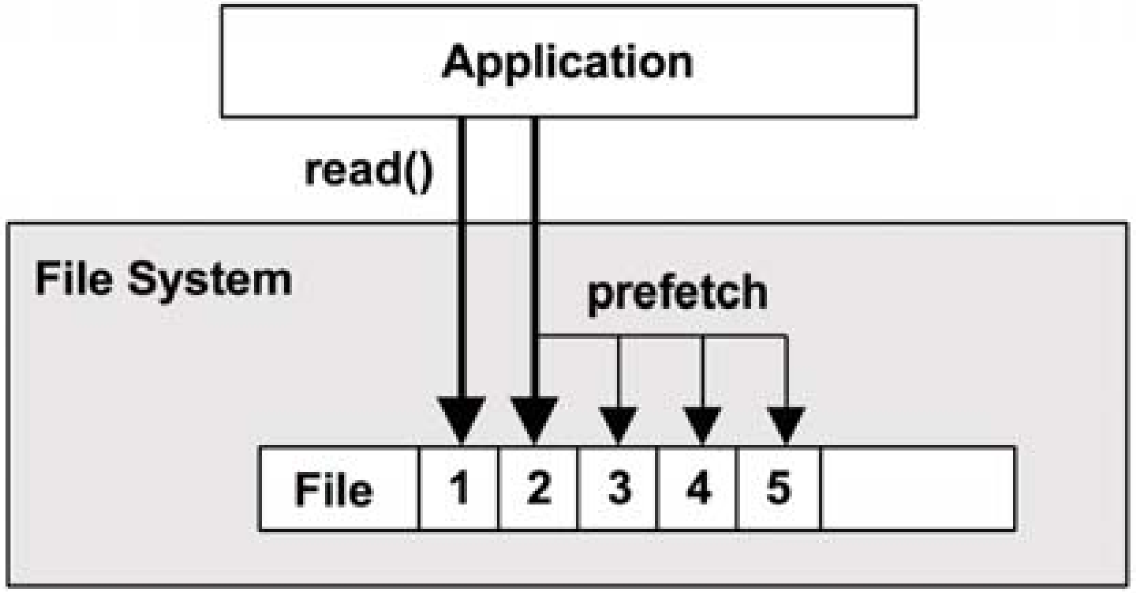
- On a write, should we do write-back or a write-through?
  - With write-back, you may lose data that is written if machine goes down before write-back
  - With write-through, you may be losing performance
    - Loss in opportunity to perform several writes at a time
    - Perhaps the write may not even be needed!
- DOS uses write-through

Filesystem Caching (UNIX)

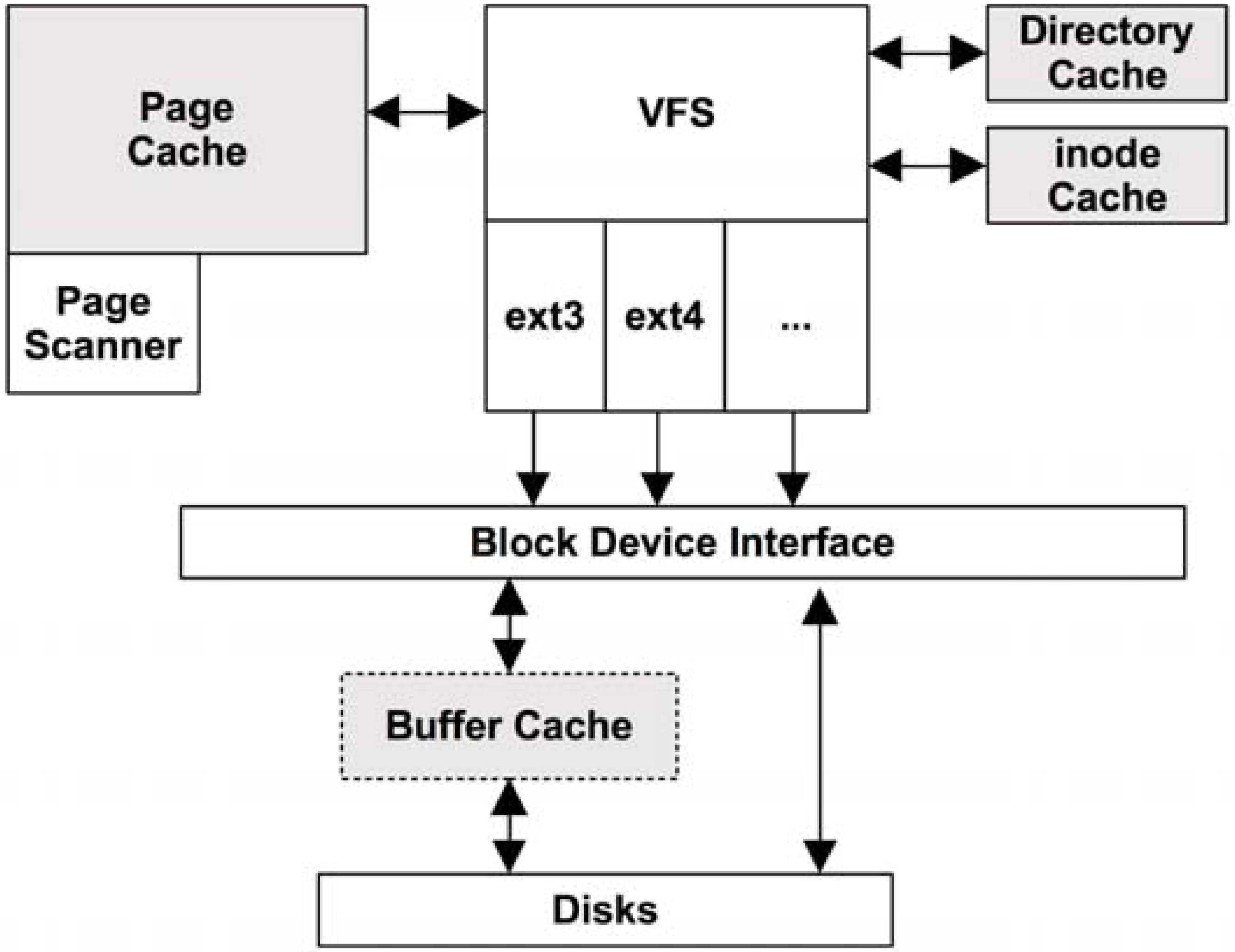
- In UNIX,
  - writes are buffered, and they are propagated in the background after a delay, i.e. every 30 secs there is a `sync()` call which propagates dirty blocks to disk.
  - This is usually done in the background.
  - Metadata (directories/inodes) writes are propagated immediately.

FS Prefetch

- Often we are reading data sequentially (e.g., file system backup)
- If FS detects sequential workload based on current/ previous I/O offsets, it can predict that the next reads will likely be sequential as well
- Populate the filesystem cache with these pre-fetches



Linux FS Caches



Page cache: cache of virtual memory pages including memory-mapped files

Buffer cache: stores blocks from disk, now unified with the page cache (why?)

Page scanner: schedules dirty pages to be written to disk

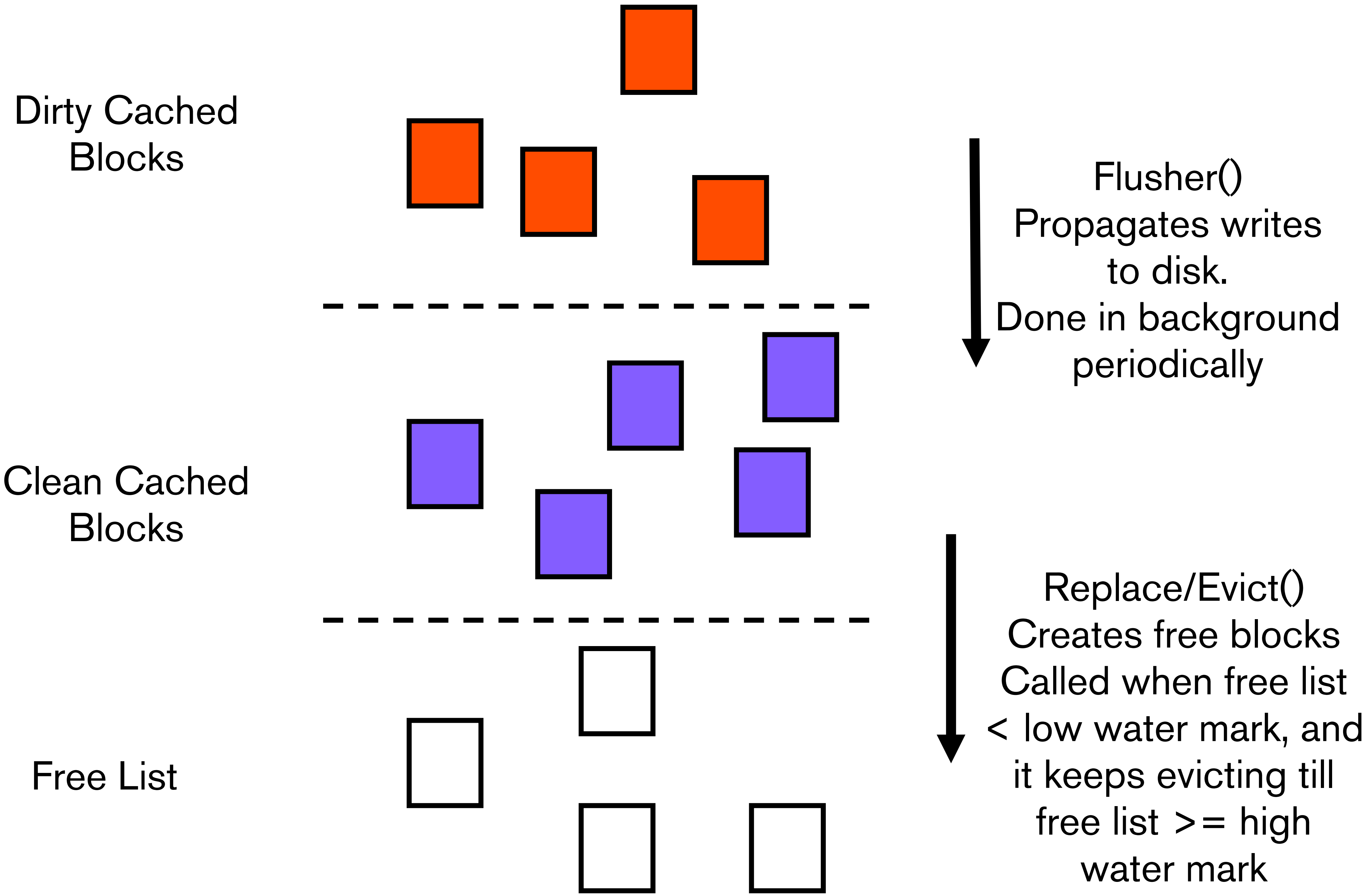
Dentry cache: remembers mappings from directory entry to inode (for what purpose?)

Inode cache: stores inodes in hash table for fast lookup

Cache space Limitations

- We need a replacement algorithm.
- Here we can use LRU, since the OS gets called on each reference to a block and the management is done in software.
- However, you typically do not do this on demand!
- Use *High* and *Low* water marks:
    - When the # of free blocks falls below *Low water mark*, evict blocks from memory till it reaches *High water mark*.

Buffer/Cache Management



Dirty Cached Blocks

Flusher()
Propagates writes
to disk.
Done in background
periodically

Clean Cached Blocks

Replace/Evict()
Creates free blocks
Called when free list
< low water mark, and
it keeps evicting till
free list >= high
water mark

Free List

Block Sizes

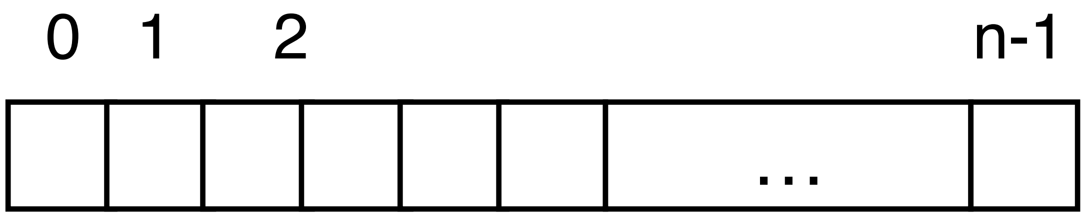
- Larger block sizes => higher internal fragmentation.
- Larger block sizes => higher disk transfer rates
- Median file size in UNIX environments ~ 1K
- Typical block sizes are of the order of 512, 1K or 4K.

Free Space

- Find the block to use when one is needed
  - Find space quickly
  - Keep storage reasonable
- Options
  - Bit vector
  - Linked List
  - Grouping
  - Counting

Free-Space Management

- Bit vector (*n* blocks)



0 1 2 n-1

…

$$\text{bit}[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ free} \\ 1 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

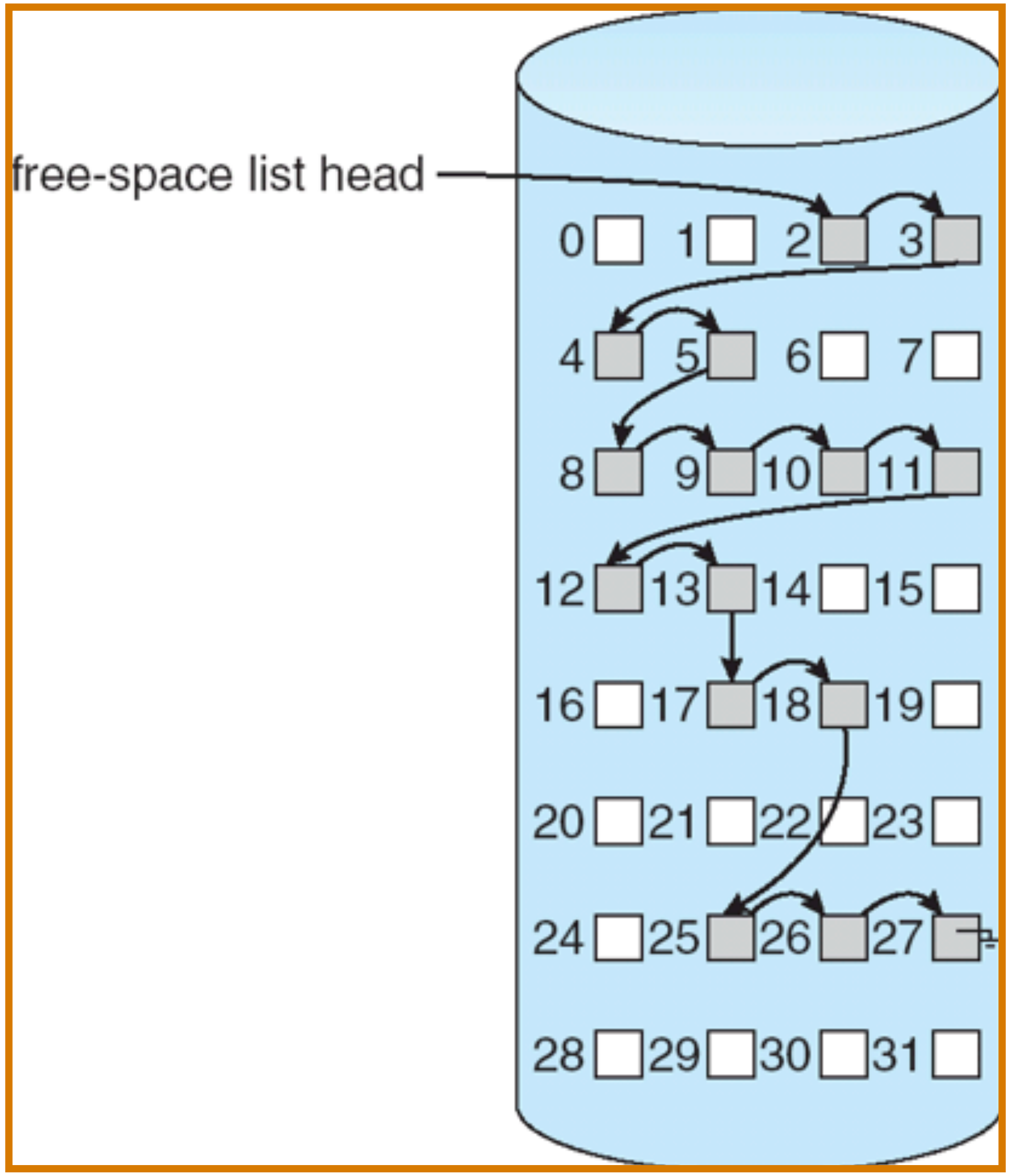
Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

Free-Space Management

- Bit vector downside
  - Space
- Example:
- block size = 2^12 bytes
- disk size = 2^30 bytes (1 gigabyte)
- n = 2^30/2^12 = 2^18 bits (or 32K bytes)

Free-Space Linked List



Linked List Optimizations

- Grouping
  - Store *n* free blocks in first free block
  - Last entry points to next block of free blocks
- Counting
  - Specify start block and number of contiguous free blocks

File System Reliability

- *Availability* of data and *integrity* of this data are both equally important.

- Need to allow for different scenarios:
  - Disks (or disk blocks) can go bad
  - Machine can crash
  - Users can make mistakes

Disks (blocks) go Bad

- Typically provide some kind of redundancy, e.g. Redundant Arrays of Inexpensive Disks (RAID)
  - Parity
  - Complete Mirroring
- When the data from the replicas/parity do not match, you employ some kind of voting to figure out which is correct.
- Once bad blocks/sectors are detected, you mark them, and do not allocate on them.

Machine crashes

- Note that data loss due to writes not being flushed immediately to disk is handled separately by setting frequency of *flusher()*.
- When the machine comes back up, we want to make sure the file system comes back up in a consistent state, e.g. a block does not appear in a file and free list at same time.
- This is done by a routine called *fsck()*.

Fsck – File System Consistency Check

- Blocks:
  - for every block keep 2 counters:
    - a) # occurrences in files
    - b) # occurrences in free list.
  - For every inode, increment all the (a)s for the blocks that the file covers.
  - For the free list, increment (b) for all blocks in the free list.
  - Ideally (a) + (b) = 1 for every block.
  - However,
    - If (a) = (b) = 0, missing block, add to free list.
    - If (a) = (b) = 1, remove the block from free list
    - If (b) > 1, remove duplicates from free list.
    - If (a) > 1, make copies of this block, and insert into each of the other files.

File System Updates

- To create a new file, we need to update:
  - Directories
  - File control blocks
  - Data blocks
  - Meta data -- free counts
- What happens if there is a crash in the middle?

Journaling File Systems

- File system changes are applied in a transaction
- Once these changes are written, user process can proceed
  - Can then apply changes to actual file system structures
- On crash, can apply committed transactions
  - What about those that were not completed?

Summary

- File System Implementation
  - FAT and inodes
  - dentries
  - Directories
  - File Retrieval
  - Caching
  - Free-Space Management
  - Recovery

- Next time: Storage