

# File Systems: Introduction

## CS 111

### Operating Systems

#### Peter Reiher

# Outline

- File systems:
  - Why do we need them?
  - Why are they challenging?
- Basic elements of file system design
- Designing file systems for disks
  - Basic issues
  - Free space, allocation, and deallocation

# Introduction

- Most systems need to store data persistently
  - So it's still there after reboot, or even power down
- Typically a core piece of functionality for the system
  - Which is going to be used all the time
- Even the operating system itself needs to be stored this way
- So we must store some data persistently

# Our Persistent Data Options

- Use raw disk blocks to store the data
  - Those make no sense to users
  - Not even easy for OS developers to work with
- Use a database to store the data
  - Probably more structure (and possibly overhead) than we need or can afford
- Use a file system
  - Some organized way of structuring persistent data
  - Which makes sense to users and programmers

# File Systems

- Originally the computer equivalent of a physical filing cabinet
- Put related sets of data into individual containers
- Put them all into an overall storage unit
- Organized by some simple principle
  - E.g., alphabetically by title
  - Or chronologically by date
- Goal is to provide:
  - Persistence
  - Ease of access
  - Good performance

# The Basic File System Concept

- Organize data into natural coherent units
  - Like a paper, a spreadsheet, a message, a program
- Store each unit as its own self-contained entity
  - *A file*
  - Store each file in a way allowing efficient access
- Provide some simple, powerful organizing principle for the collection of files
  - Making it easy to find them
  - And easy to organize them

# File Systems and Hardware

- File systems are typically stored on hardware providing persistent memory
  - Disks, tapes, flash memory, etc.
- With the expectation that a file put in one “place” will be there when we look again
- Performance considerations will require us to match the implementation to the hardware
  - Remember seek time and rotational latency?
- But ideally, the same user-visible file system should work on any reasonable hardware

# File Systems and OS Abstractions

- Obviously a version of the basic memory abstraction
- So we'd expect `read()` and `write()` operations for it
- We could have a file system abstraction very close to the hardware reality
  - E.g., exposing disk cylinders or flash erase cycles
- But it's better to hide the messy details
  - Treat files as magically persistent memory

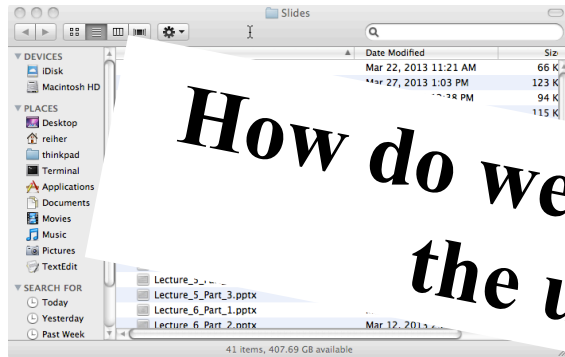


# Data and Metadata

- File systems deal with two kinds of information
- *Data* – the information that the file is actually supposed to store
  - E.g., the instructions of the program or the words in the letter
- *Metadata* – Information about the information the file stores
  - E.g., how many bytes are there and when was it created
  - Sometimes called *attributes*
- Ultimately, both data and metadata must be stored persistently
  - And usually on the same piece of hardware

# Bridging the Gap

We want something like . . .



But we've got something like . . .



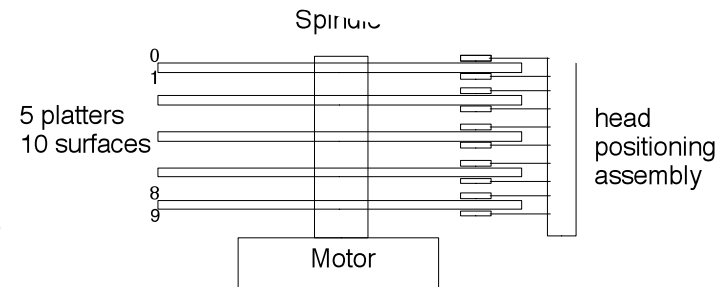
**How do we get from the hardware to the useful abstraction?**

Or . . .



Or at least

```
drwxr-xr-x  8 root  wheel   272 May  4  2010 X11
lrwxr-xr-x  1 root  wheel     3 May  4  2010 X11R6 -> X11
drwxr-xr-x 913 root  wheel 31042 Apr 21 12:21 bin
drwxr-xr-x 336 root  wheel 11424 Mar 17 09:13 lib
drwxr-xr-x 103 root  wheel  3502 Apr 21 12:23 libexec
drwxr-xr-x  7 root  wheel   238 Jan 16 23:00 local
drwxr-xr-x 238 root  wheel  8092 Mar 17 09:13 sbin
drwxr-xr-x  59 root  wheel  2006 Apr 21 12:21 share
drwxr-xr-x  4 root  wheel   136 May  4  2010 standalone
```



# A Further Wrinkle

- We want our file system to be agnostic to the storage medium
- Same program should access the file system the same way, regardless of medium
  - Otherwise it's hard to write portable programs
- Should work the same for disks of different types
- Or if we use a RAID instead of one disk
- Or if we use flash instead of disks
- Or if even we don't use persistent memory at all
  - E.g., RAM file systems

# Desirable File System Properties

- What are we looking for from our file system?
  - Persistence
  - Easy use model
    - For accessing one file
    - For organizing collections of files
  - Flexibility
    - No limit on number of files
    - No limit on file size, type, contents
  - Portability across hardware device types
  - Performance
  - Reliability
  - Suitable security

# The Performance Issue

- How fast does our file system need to be?
- Ideally, as fast as everything else
  - Like CPU, memory, and the bus
  - So it doesn't provide a bottleneck
- But these other devices operate today at nanosecond speeds
- Disk drives operate at millisecond speeds
- Suggesting we'll need to do some serious work to hide the mismatch

# The Reliability Issue

- Persistence implies reliability
- We want our files to be there when we check, no matter what
- Not just on a good day
- So our file systems must be free of errors
  - Hardware or software
- Remember our discussion of concurrency, race conditions, etc.?
  - Might we have some challenges here?

# “Suitable” Security

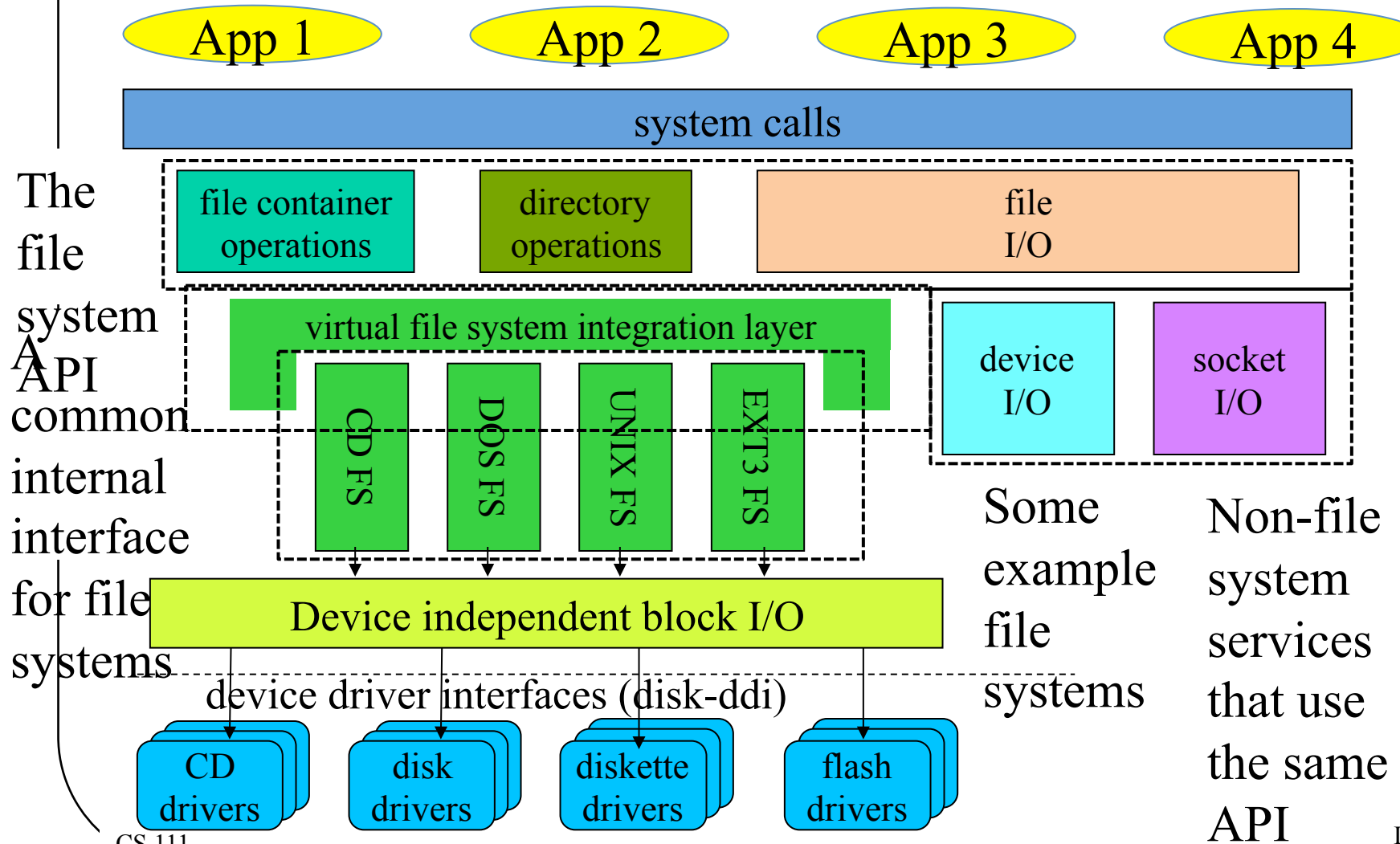
- What does that mean?
- Whoever owns the data should be able to control who accesses it
  - Using some well-defined access control model and mechanism
- With strong guarantees that the system will enforce his desired controls
  - Implying we’ll apply complete mediation
  - To the extent performance allows

# Basics of File System Design

- Where do file systems fit in the OS?
- File control data structures



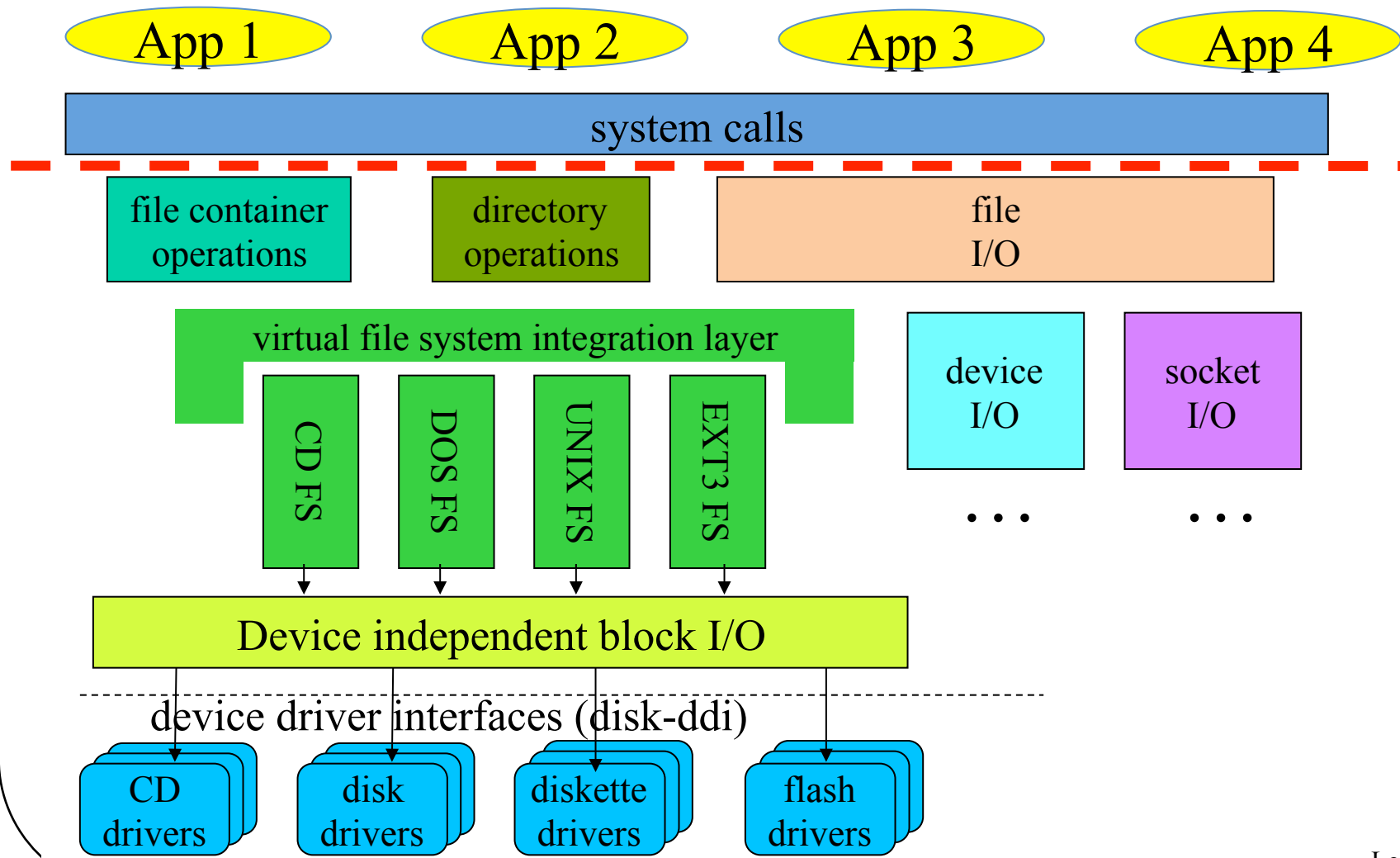
# File Systems and the OS



# File Systems and Layered Abstractions

- At the top, apps think they are accessing files
- At the bottom, various block devices are reading and writing blocks
- There are multiple layers of abstraction in between
- Why?
- Why not translate directly from application file operations to devices' block operations?

# The File System API



# The File System API

- Highly desirable to provide a single API to programmers and users for all files
- Regardless of how the file system underneath is actually implemented
- A requirement if one wants program portability
  - Very bad if a program won't work because there's a different file system underneath
- Three categories of system calls here
  1. File container operations
  2. Directory operations
  3. File I/O operations

# File Container Operations

- Standard file management system calls
  - Manipulate files as objects
  - These operations ignore the contents of the file
- Implemented with standard file system methods
  - Get/set attributes, ownership, protection ...
  - Create/destroy files and directories
  - Create/destroy links
- Real work happens in file system implementation

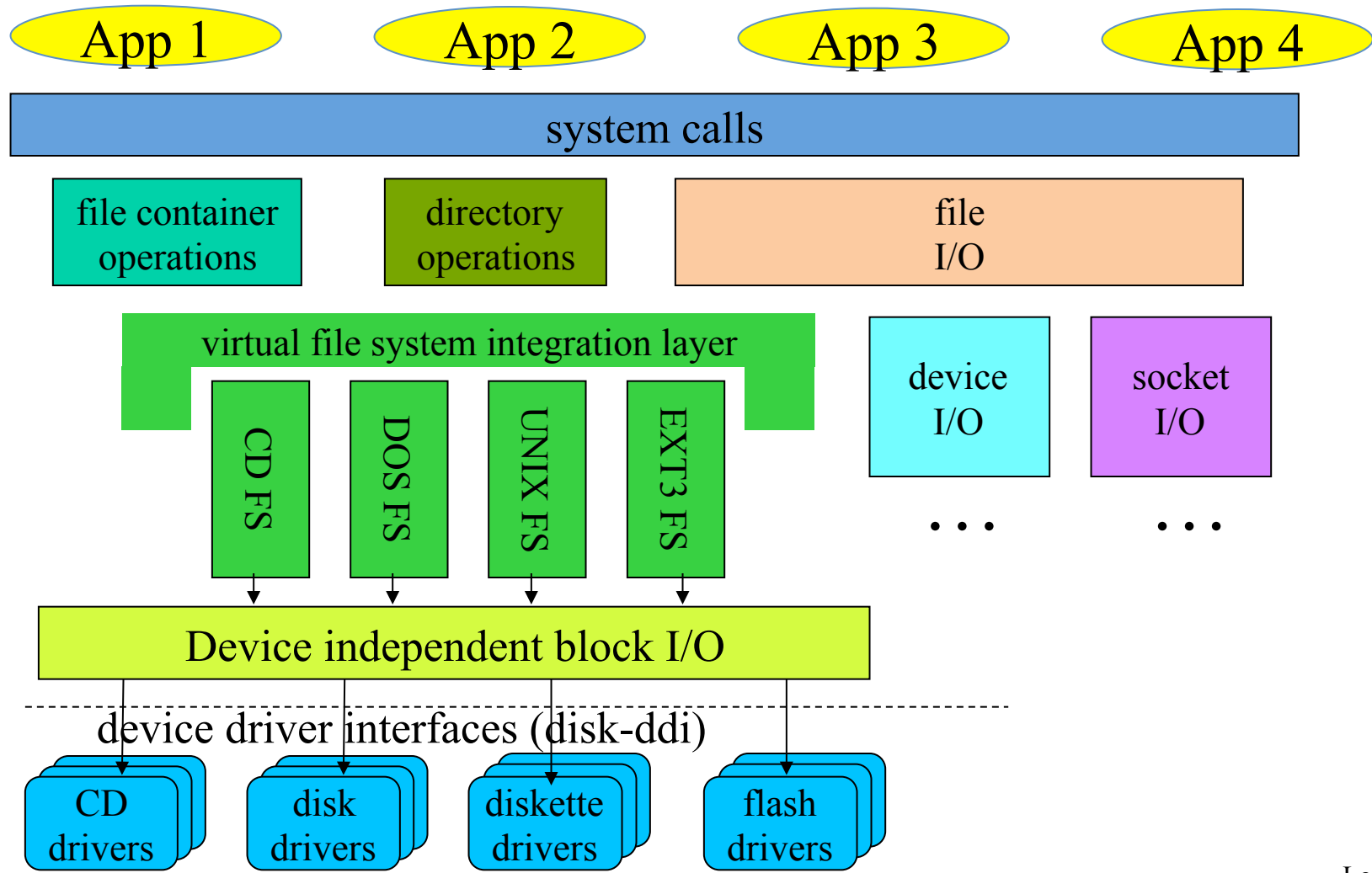
# Directory Operations

- Directories provide the organization of a file system
  - Typically hierarchical
  - Sometimes with some extra wrinkles
- At the core, directories translate a name to a lower-level file pointer
- Operations tend to be related to that
  - Find a file by name
  - Create new name/file mapping
  - List a set of known names

# File I/O Operations

- Open – map name into an open instance
- Read data from file and write data to file
  - Implemented using logical block fetches
  - Copy data between user space and file buffer
  - Request file system to write back block when done
- Seek
  - Change logical offset associated with open instance
- Map file into address space
  - File block buffers are just pages of physical memory
  - Map into address space, page it to and from file system

# The Virtual File System Layer

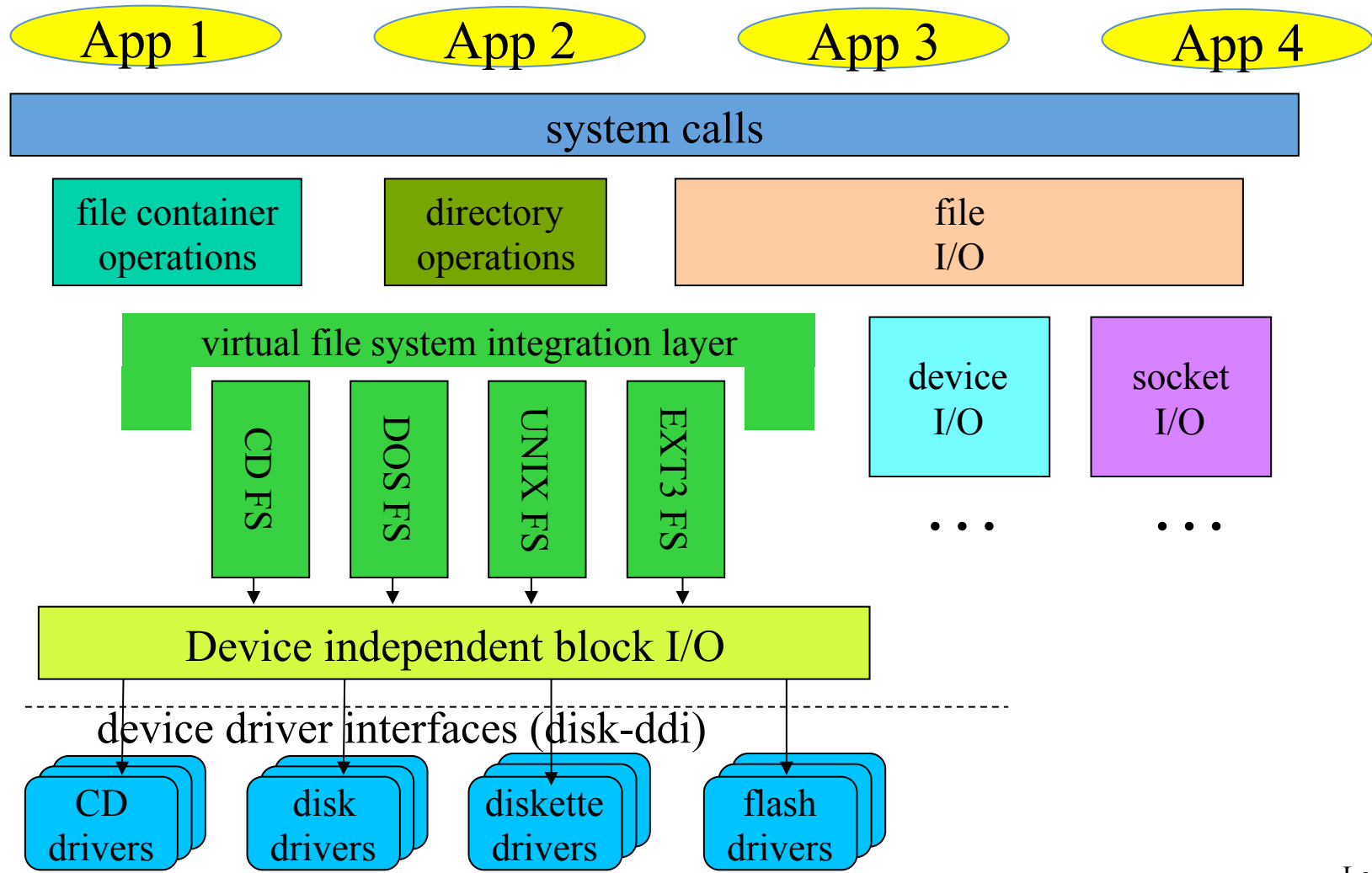




# The Virtual File System (VFS) Layer

- Federation layer to generalize file systems
  - Permits rest of OS to treat all file systems as the same
  - Support dynamic addition of new file systems
- Plug-in interface or file system implementations
  - DOS FAT, Unix, EXT3, ISO 9660, network, etc.
  - Each file system implemented by a plug-in module
  - All implement same basic methods
    - Create, delete, open, close, link, unlink,
    - Get/put block, get/set attributes, read directory, etc.
- Implementation is hidden from higher level clients
  - All clients see are the standard methods and properties

# The File System Layer



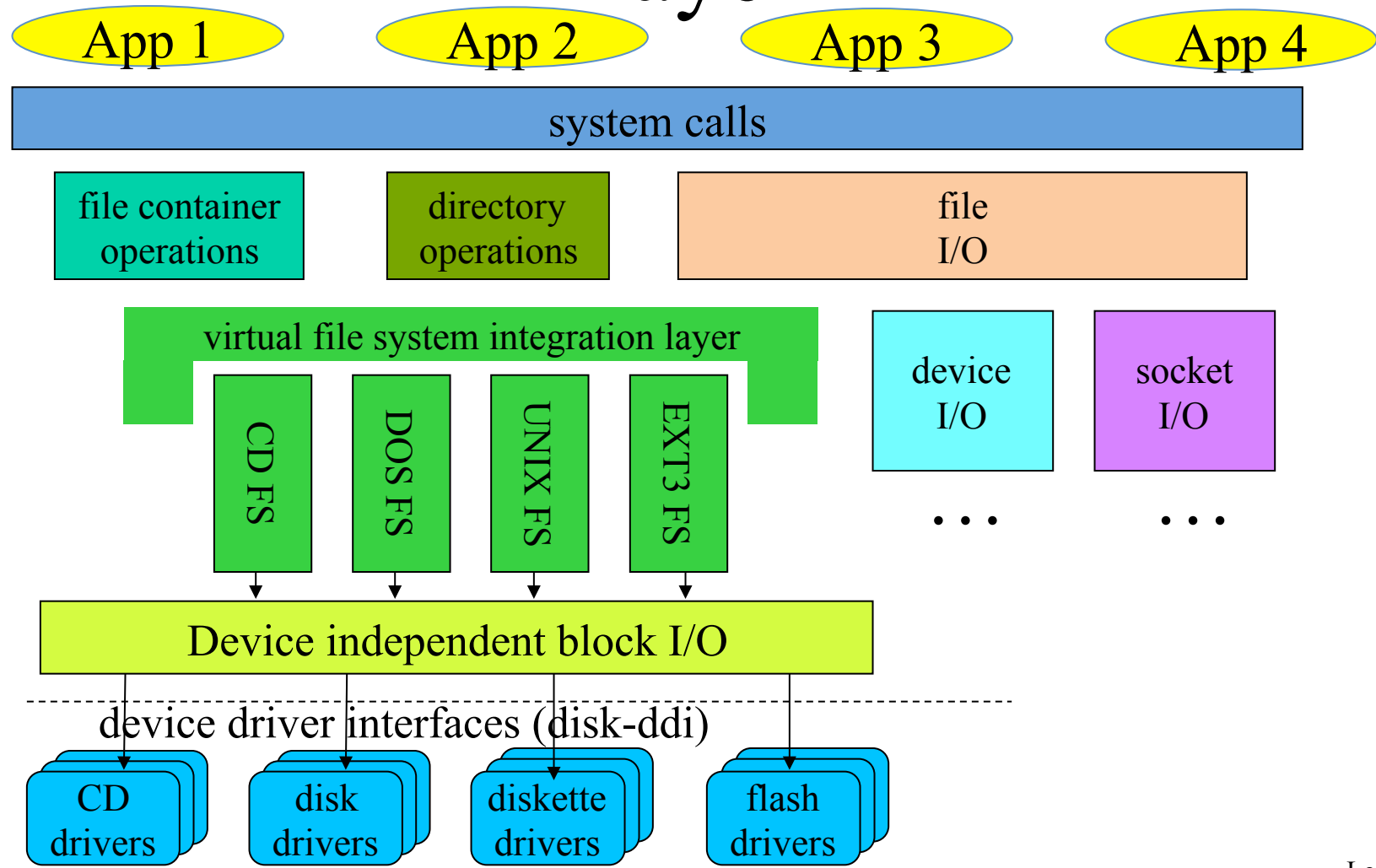
# The File Systems Layer

- Desirable to support multiple different file systems
- All implemented on top of block I/O
  - Should be independent of underlying devices
- All file systems perform same basic functions
  - Map names to files
  - Map <file, offset> into <device, block>
  - Manage free space and allocate it to files
  - Create and destroy files
  - Get and set file attributes
  - Manipulate the file name space

# Why Multiple File Systems?

- Why not instead choose one “good” one?
- There may be multiple storage devices
  - E.g., hard disk and flash drive
  - They might benefit from very different file systems
- Different file systems provide different services, despite the same interface
  - Differing reliability guarantees
  - Differing performance
  - Read-only vs. read/write
- Different file systems used for different purposes
  - E.g., a temporary file system

# Device Independent Block I/O Layer



# File Systems and Block I/O Devices

- File systems typically sit on a general block I/O layer
- A generalizing abstraction – make all disks look same
- Implements standard operations on each block device
  - Asynchronous read (physical block #, buffer, bytecount)
  - Asynchronous write (physical block #, buffer, bytecount)
- Map logical block numbers to device addresses
  - E.g., logical block number to <cylinder, head, sector>
- Encapsulate all the particulars of device support
  - I/O scheduling, initiation, completion, error handlings
  - Size and alignment limitations

# Why Device Independent Block I/O?

- A better abstraction than generic disks
- Allows unified LRU buffer cache for disk data
  - Hold frequently used data until it is needed again
  - Hold pre-fetched read-ahead data until it is requested
- Provides buffers for data re-blocking
  - Adapting file system block size to device block size
  - Adapting file system block size to user request sizes
- Handles automatic buffer management
  - Allocation, deallocation
  - Automatic write-back of changed buffers

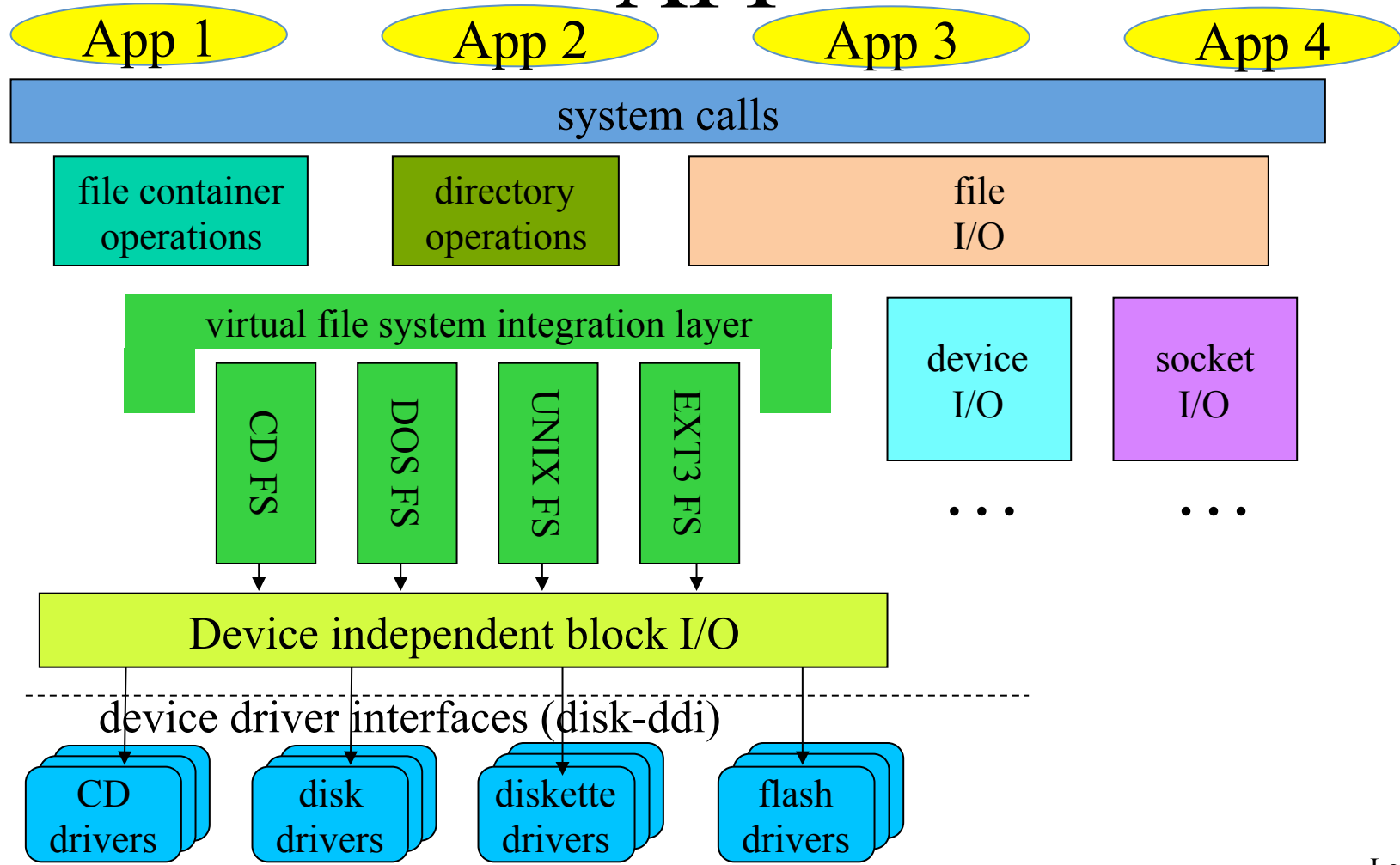
# Why Do We Need That Cache?

- File access exhibits a high degree of reference locality at multiple levels:
  - Users often read and write a single block in small operations, reusing that block
  - Users read and write the same files over and over
  - Users often open files from the same directory
  - OS regularly consults the same meta-data blocks
- Having common cache eliminates many disk accesses, which are slow



# Devices, Sockets and File System

## API



# Device and Socket I/O

- Devices are, well, devices
- Sockets are an IPC mechanism
- What are they doing in this description of file systems?
- Unix systems typically abstract them using the file interface
  - Which allows file-type operations to be performed on them

# File Systems Control Structures

- A file is a named collection of information
- Primary roles of file system:
  - To store and retrieve data
  - To manage the media/space where data is stored
- Typical operations:
  - Where is the first block of this file?
  - Where is the next block of this file?
  - Where is block 35 of this file?
  - Allocate a new block to the end of this file
  - Free all blocks associated with this file

# Finding Data On Disks

- Essentially a question of how you managed the space on your disk
- Space management on disk is complex
  - There are millions of blocks and thousands of files
  - Files are continuously created and destroyed
  - Files can be extended after they have been written
  - Data placement on disk has performance effects
  - Poor management leads to poor performance
- Must track the space assigned to each file
  - On-disk, master data structure for each file

# On-Disk File Control Structures

- On-disk description of important attributes of a file
  - Particularly where its data is located
- Virtually all file systems have such data structures
  - Different implementations, performance & abilities
  - Implementation can have profound effects on what the file system can do (well or at all)
- A core design element of a file system
- Paired with some kind of in-memory representation of the same information

# The Basic File Control Structure Problem

- A file typically consists of multiple data blocks
- The control structure must be able to find them
- Preferably able to find any of them quickly
  - I.e., shouldn't need to read the entire file to find a block near the end
- Blocks can be changed
- New data can be added to the file
  - Or old data deleted
- Files can be sparsely populated

# The In-Memory Representation

- There is an on-disk structure pointing to disk blocks (and holding other information)
- When file is opened, an in-memory structure is created
- Not an exact copy of the disk version
  - The disk version points to disk blocks
  - The in-memory version points to RAM pages
    - Or indicates that the block isn't in memory
  - Also keeps track of which blocks are dirty and which aren't

# In-Memory Structures and Processes

- What if multiple processes have a given file open?
- Should they share one control structure or have one each?
- In-memory structures typically contain a cursor pointer
  - Indicating how far into the file data has been read/written
- Sounds like that should be per-process . . .



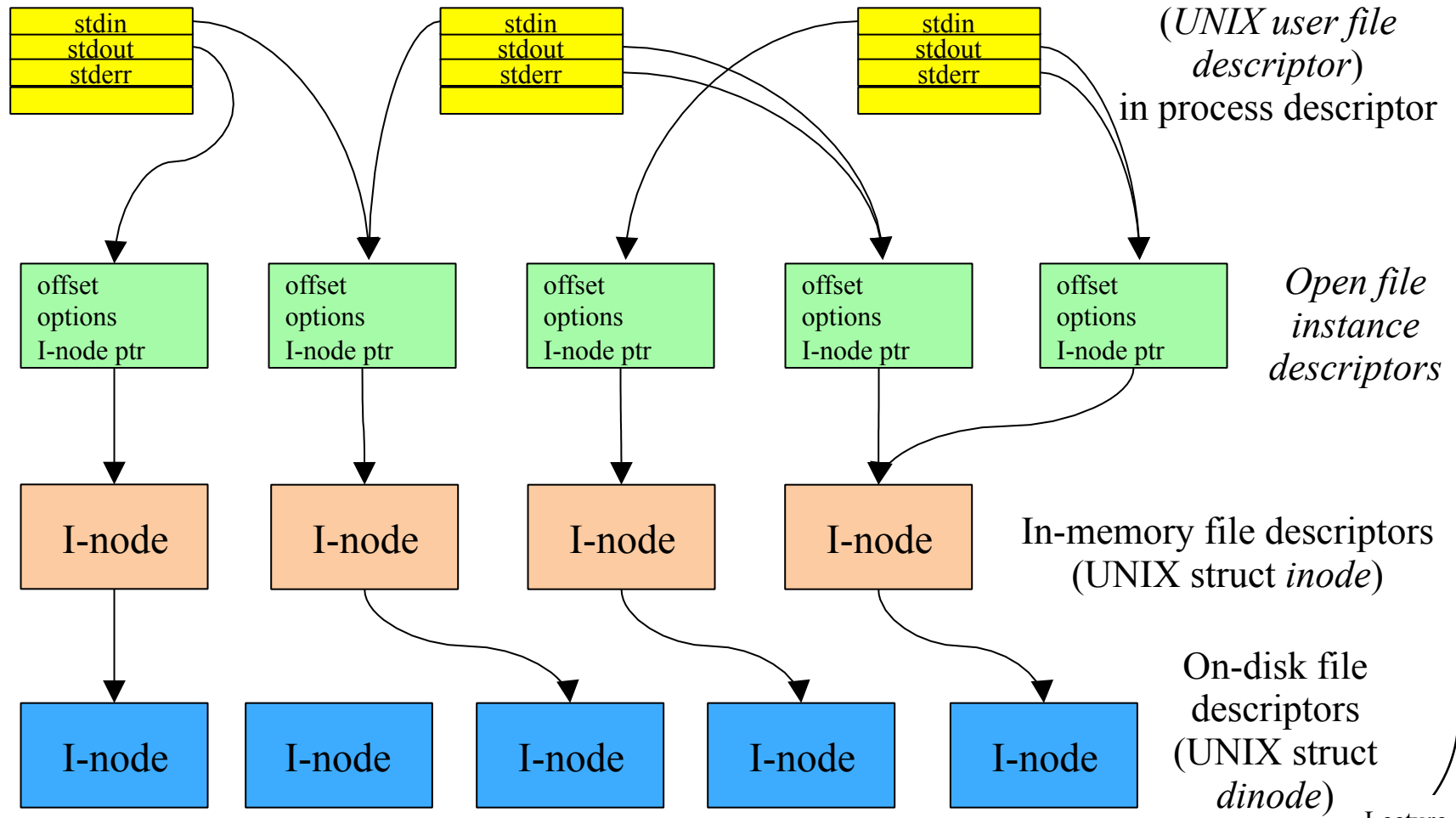
# Per-Process or Not?

- What if cooperating processes are working with the same file?
  - They might want to share a cursor
- And how can we know when all processes are finished with an open file?
  - So we can reclaim space used for its in-memory descriptor
- Implies a two-level solution
  1. A structure shared by all
  2. A structure shared by cooperating processes

# The Unix Approach

Two processes can  
share one descriptor

Two descriptors can  
share one inode



# File System Structure

- How do I organize a disk into a file system?
  - Linked extents
    - The DOS FAT file system
  - File index blocks
    - Unix System V file system

# Basics of File System Structure

- Most file systems live on disks
- Disk volumes are divided into fixed-sized blocks
  - Many sizes are used: 512, 1024, 2048, 4096, 8192 ...
- Most blocks will be used to store user data
- Some will be used to store organizing “meta-data”
  - Description of the file system (e.g., layout and state)
  - File control blocks to describe individual files
  - Lists of free blocks (not yet allocated to any file)
- All operating systems have such data structures
  - Different OSes and file systems have very different goals

CS 111 These result in very different implementations

# The Boot Block

- The 0<sup>th</sup> block of a disk is usually reserved for the boot block
  - Code allowing the machine to boot an OS
- Not usually under the control of a file system
  - It typically ignores the boot block entirely
- Not all disks are bootable
  - But the 0<sup>th</sup> block is usually reserved, “just in case”
- So file systems start work at block 1

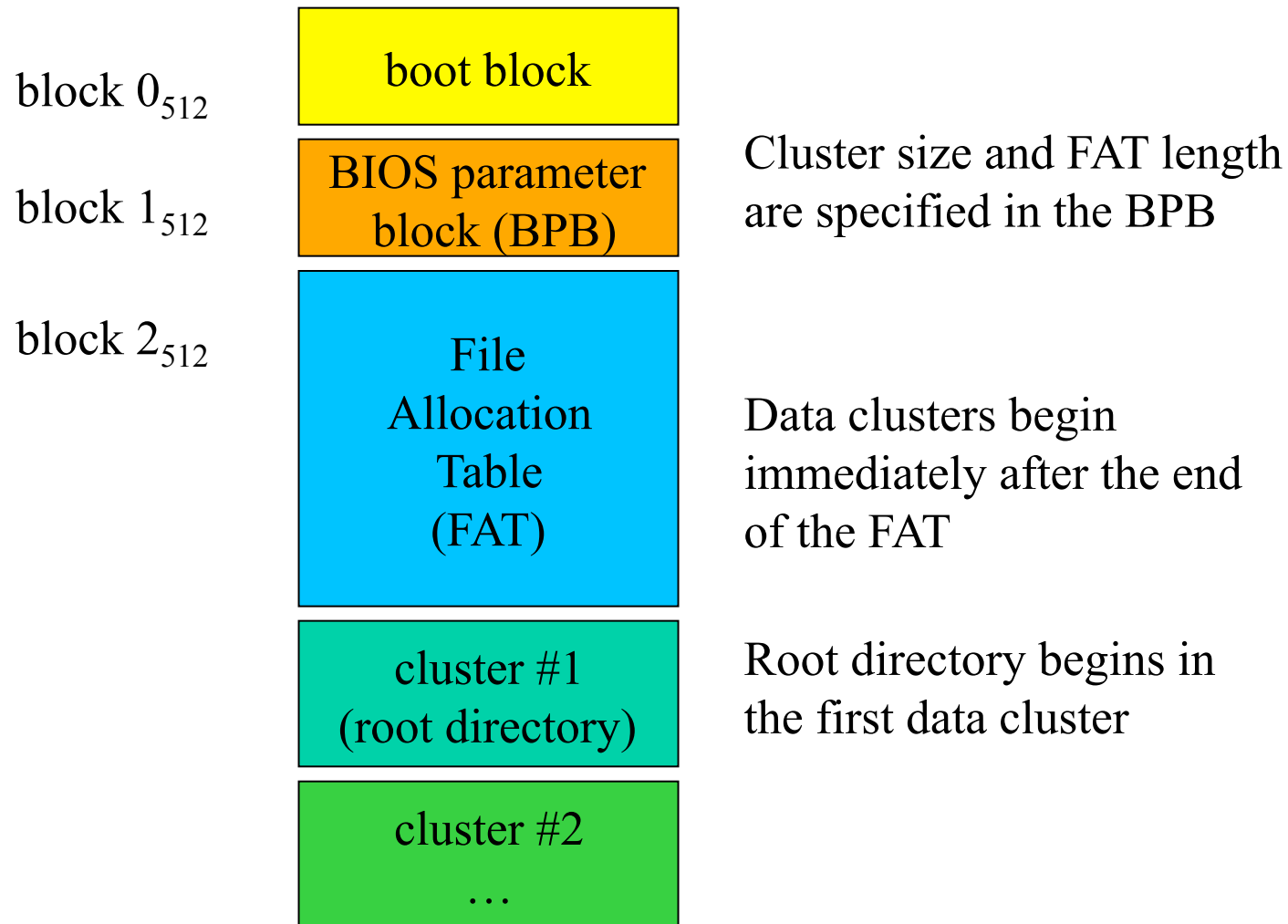
# Managing Allocated Space

- A core activity for a file system, with various choices
- What if we give each file same amount of space?
  - Internal fragmentation ... just like memory
- What if we allocate just as much as file needs?
  - External fragmentation, compaction ... just like memory
- Perhaps we should allocate space in “pages”
  - How many chunks can a file contain?
- The file control data structure determines this
  - It only has room for so many pointers, then file is “full”
- So how do we want to organize the space in a file?

# Linked Extents

- A simple answer
- File control block contains exactly one pointer
  - To the first chunk of the file
  - Each chunk contains a pointer to the next chunk
  - Allows us to add arbitrarily many chunks to each file
- Pointers can be in the chunks themselves
  - This takes away a little of every chunk
  - To find chunk N, you have to read the first N-1 chunks
- Pointers can be in auxiliary “chunk linkage” table
  - Faster searches, especially if table kept in memory

# The DOS File System





# DOS File System Overview

- DOS file systems divide space into “clusters”
  - Cluster size (multiple of 512) fixed for each file system
  - Clusters are numbered 1 through N
- File control structure points to first cluster of a file
- File Allocation Table (FAT), one entry per cluster
  - Contains the number of the next cluster in file
  - A 0 entry means that the cluster is not allocated
  - A -1 entry means “end of file”
- File system is sometimes called “FAT,” after the name of this key data structure

# DOS FAT Clusters

## directory entry

name:	myfile.txt
length:	1500 bytes
1 <sup>st</sup> cluster:	3

## File Allocation Table

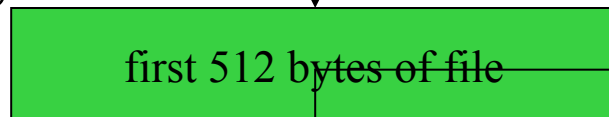
1	x
2	x
3	4
4	5
5	-1
6	0

Each FAT entry corresponds to a cluster, and contains the number of the next cluster.

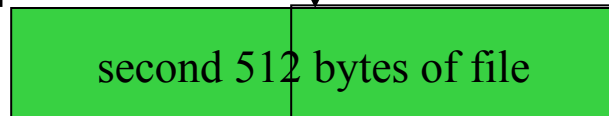
-1 = End of File

0 = free cluster

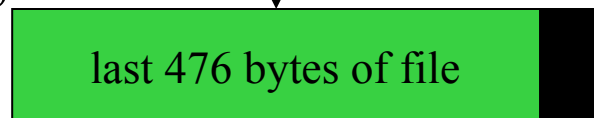
cluster #3



cluster #4



cluster #5



# DOS File System Characteristics

- To find a particular block of a file
  - Get number of first cluster from directory entry
  - Follow chain of pointers through File Allocation Table
- Entire File Allocation Table is kept in memory
  - No disk I/O is required to find a cluster
  - For very large files the search can still be long
- No support for “sparse” files
  - If a file has a block  $n$ , it must have all blocks  $< n$
- Width of FAT determines max file system size
  - How many bits describe a cluster address?
  - Originally 8 bits, eventually expanded to 32

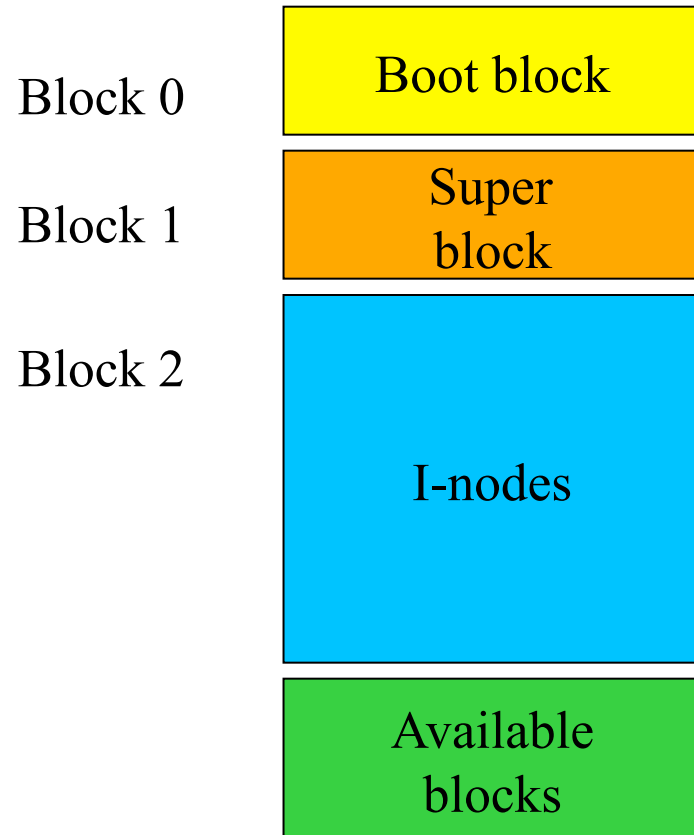
# File Index Blocks

- A different way to keep track of where a file's data blocks are on the disk
- A file control block points to all blocks in file
  - Very fast access to any desired block
  - But how many pointers can the file control block hold?
- File control block could point at extent descriptors
  - But this still gives us a fixed number of extents

# Hierarchically Structured File Index Blocks

- To solve the problem of file size being limited by entries in file index block
- The basic file index block points to blocks
- Some of those contain pointers which in turn point to blocks
- Can point to many extents, but still a limit to how many
  - But that limit might be a very large number
  - Has potential to adapt to wide range of file sizes

# Unix System V File System

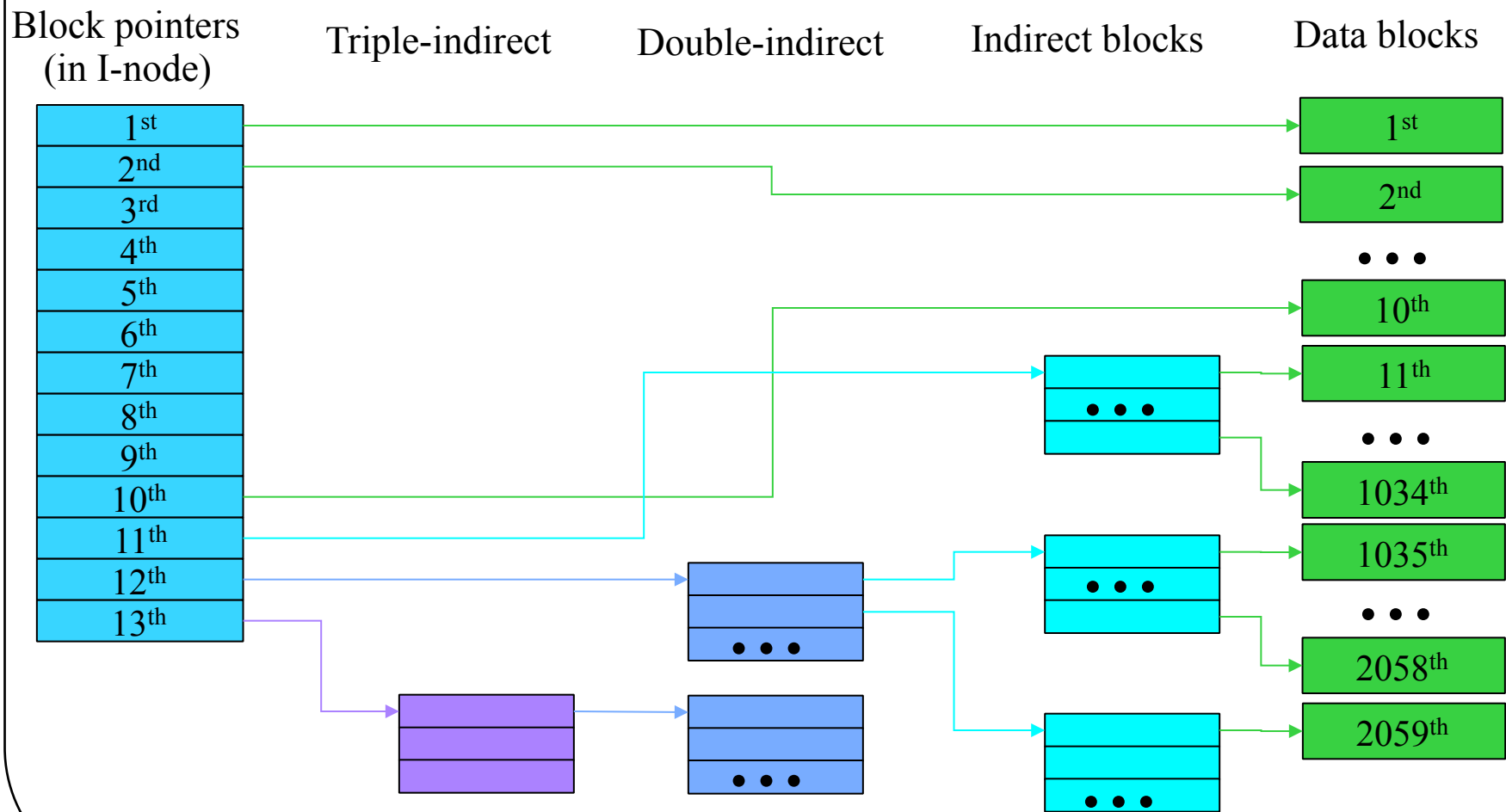


Block size and number of I-nodes are specified in super block

I-node #1 (traditionally) describes the root directory

Data blocks begin immediately after the end of the I-nodes.

# Unix Inodes and Block Pointers



# Why Is This a Good Idea?

- The UNIX pointer structure seems ad hoc and complicated
- Why not something simpler?
  - E.g., all block pointers are triple indirect
- File sizes are not random
  - The majority of files are only a few thousand bytes long
- Unix approach allows us to access up to 40Kbytes (assuming 4K blocks) without extra I/Os
  - Remember, the double and triple indirect blocks must themselves be fetched off disk



# How Big a File Can Unix Handle?

- The on-disk inode contains 13 block pointers
  - First 10 point to first 10 blocks of file
  - 11th points to an indirect block (which contains pointers to 1024 blocks)
  - 12th points to a double indirect block (pointing to 1024 indirect blocks)
  - 13th points to a triple indirect block (pointing to 1024 double indirect blocks)
- Assuming 4k bytes per block and 4 bytes per pointer
  - 10 direct blocks =  $10 * 4K \text{ bytes} = 40K \text{ bytes}$
  - Indirect block =  $1K * 4K = 4M \text{ bytes}$
  - Double indirect =  $1K * 4M = 4G \text{ bytes}$
  - Triple indirect =  $1K * 4G = 4T \text{ bytes}$
  - At the time system was designed, that seemed impossibly large
  - But . . .

# Unix Inode Performance Issues

- The inode is in memory whenever file is open
- So the first ten blocks can be found with no extra I/O
- After that, we must read indirect blocks
  - The real pointers are in the indirect blocks
  - Sequential file processing will keep referencing it
  - Block I/O will keep it in the buffer cache
- 1-3 extra I/O operations per thousand pages
  - Any block can be found with 3 or fewer reads
- Index blocks can support “sparse” files
  - Not unlike page tables for sparse address spaces