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

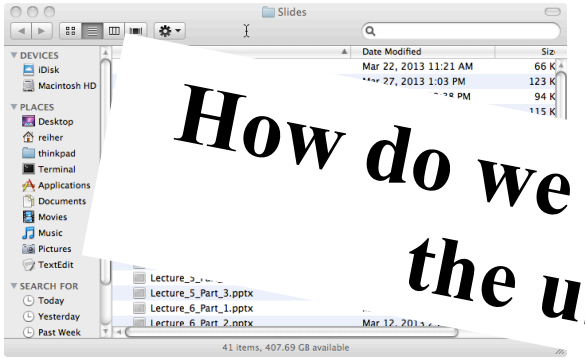
- File systems are typically stored on hardware providing persistent memory
  - Disks, tapes, flash memory, etc.
- With the expectation that a file put in one "place" will be there when we look again
- Performance considerations will require us to match the implementation to the hardware
- But ideally,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

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 . . .



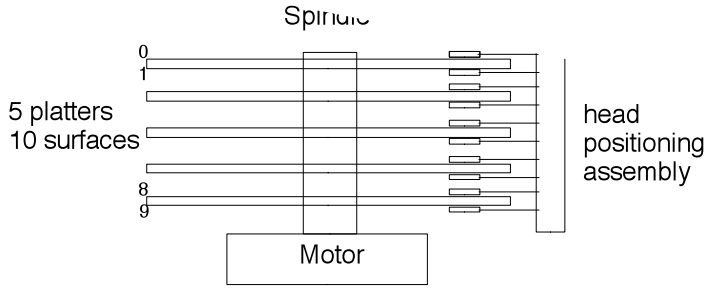
But we've got something like . . .

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 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

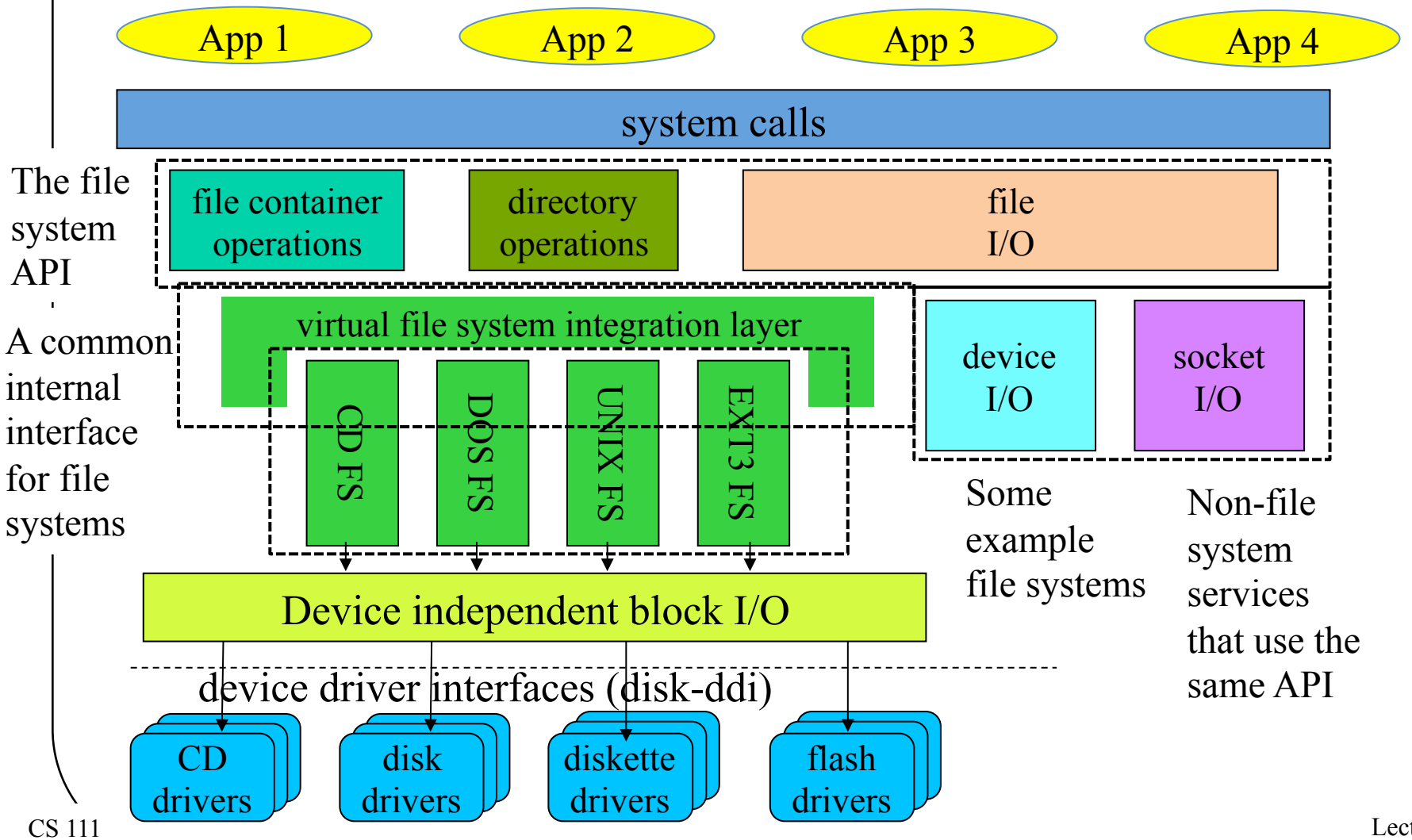
Desirable File System Properties

- 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

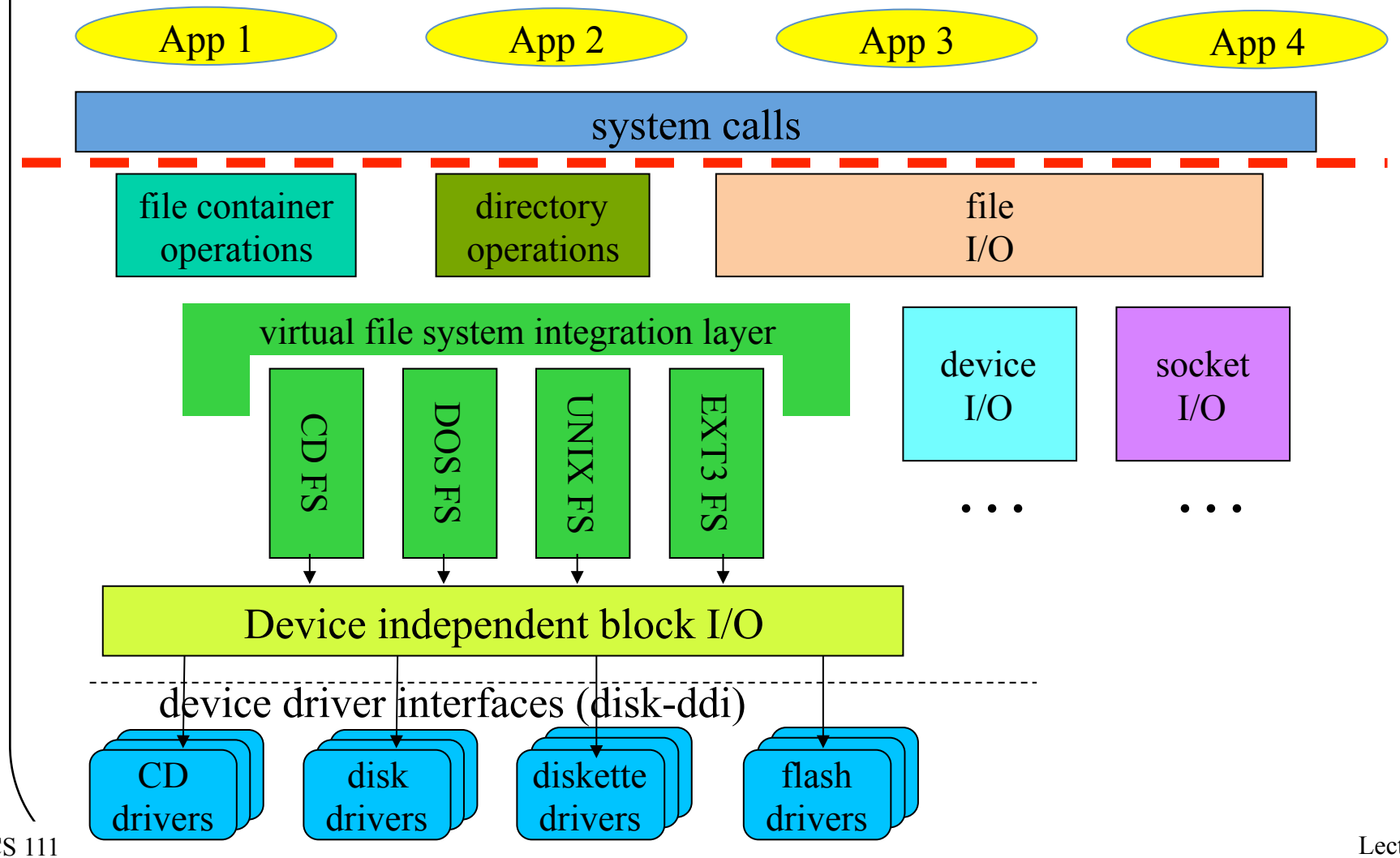
Basics of File System Design

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

File Systems and the OS



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

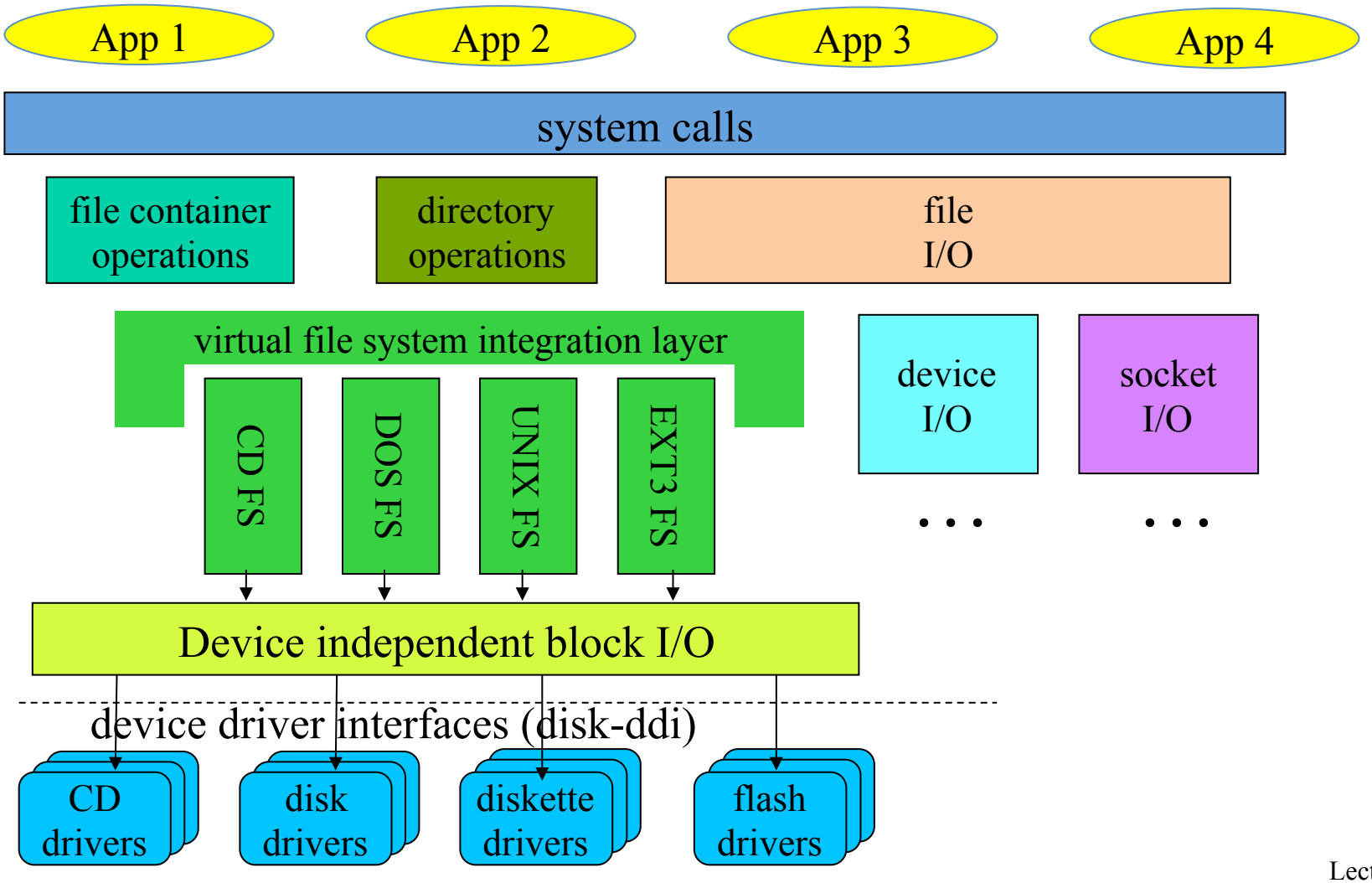
Directory Operations

- 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

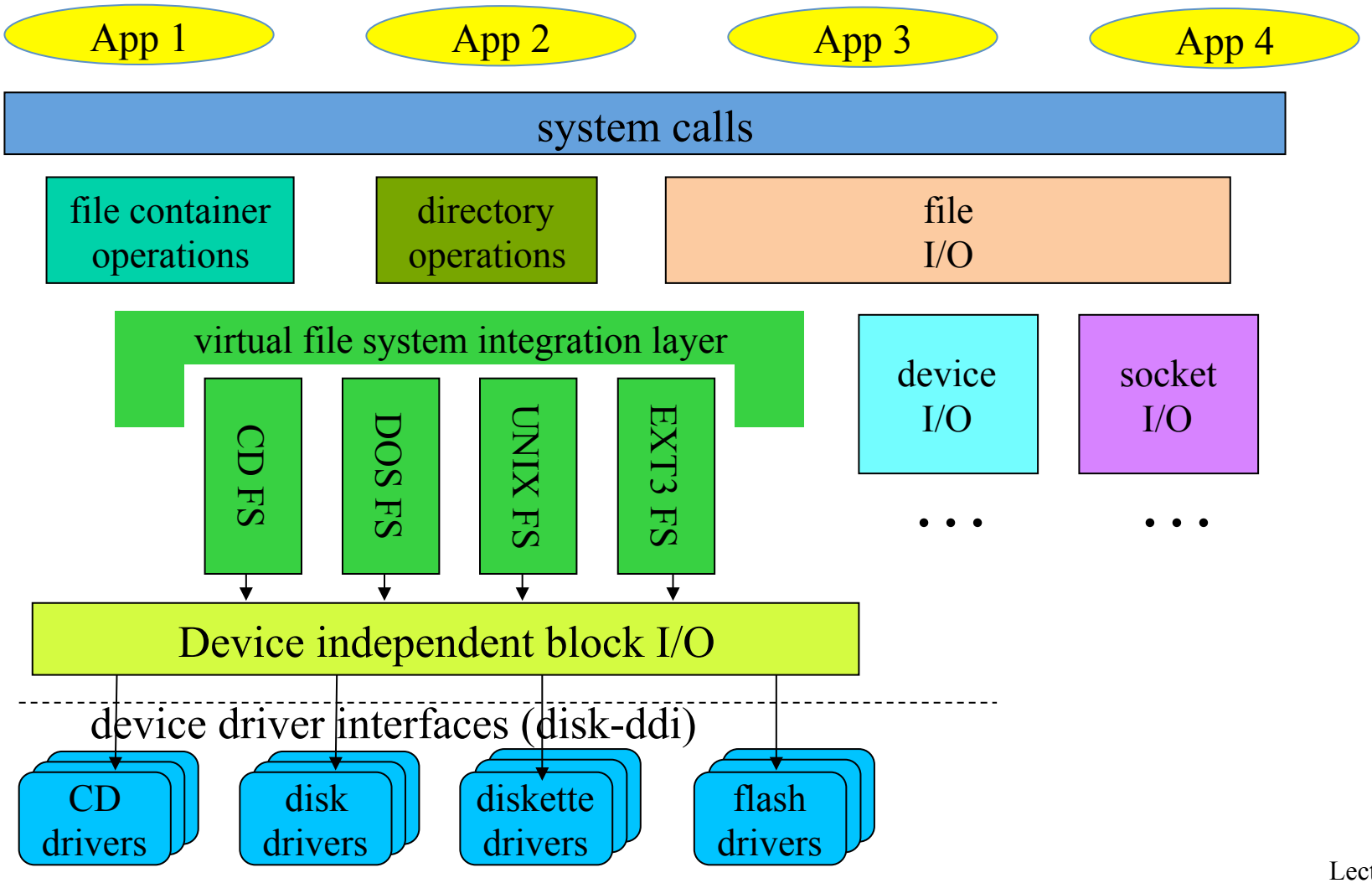
The Virtual File System Layer



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



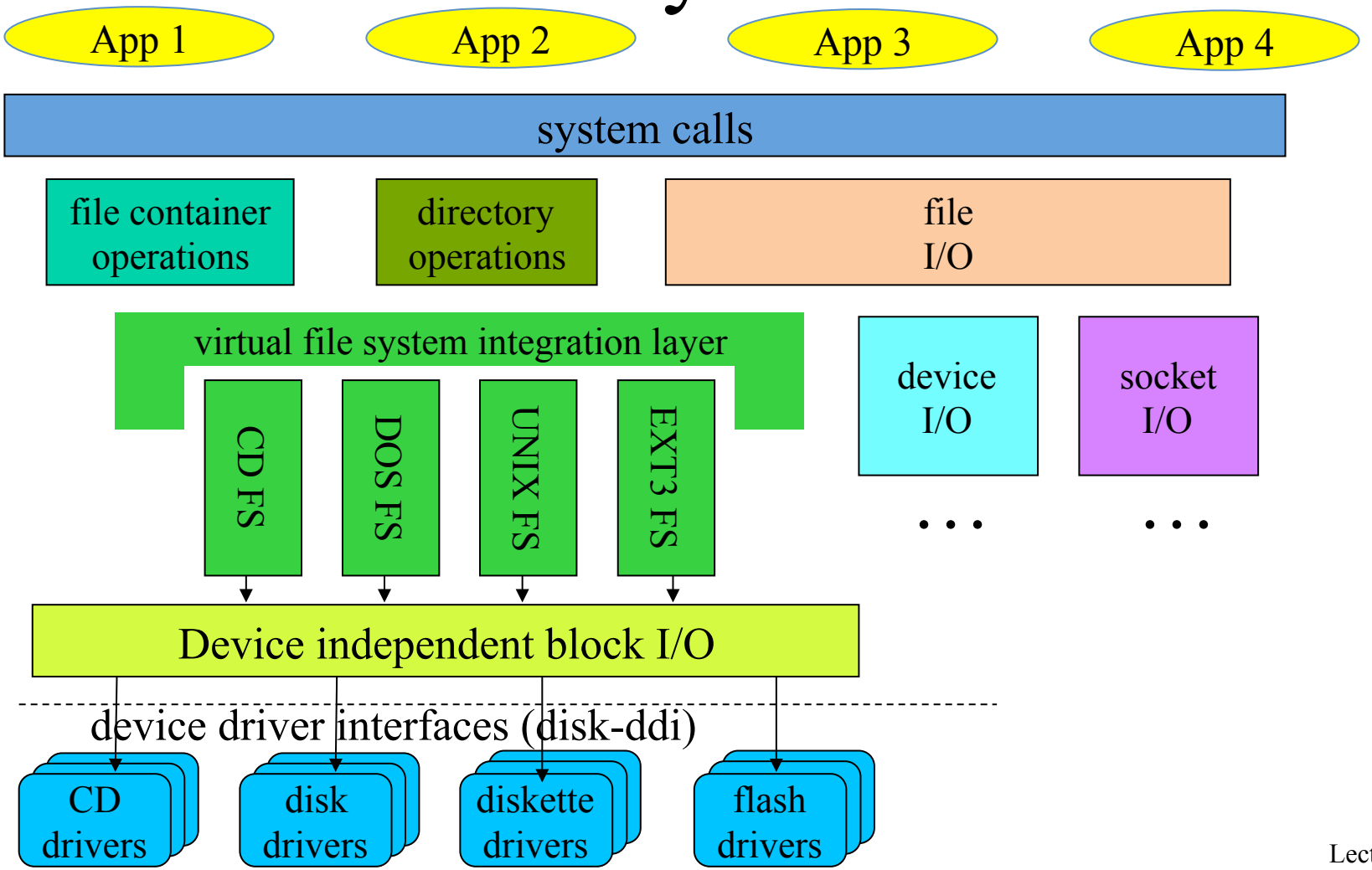
The File Systems Layer

- 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

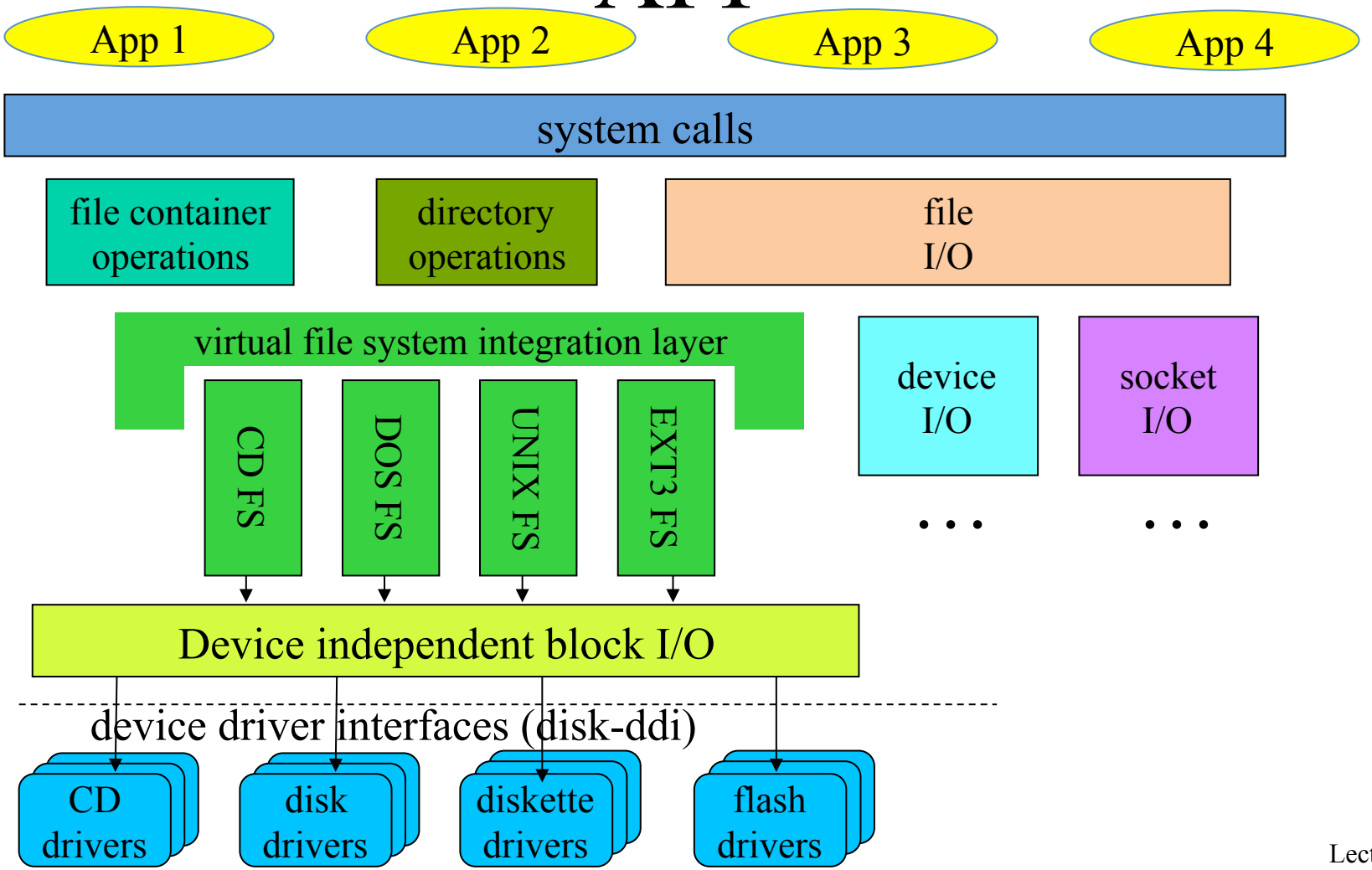
Why Device Independent Block I/O?

- 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



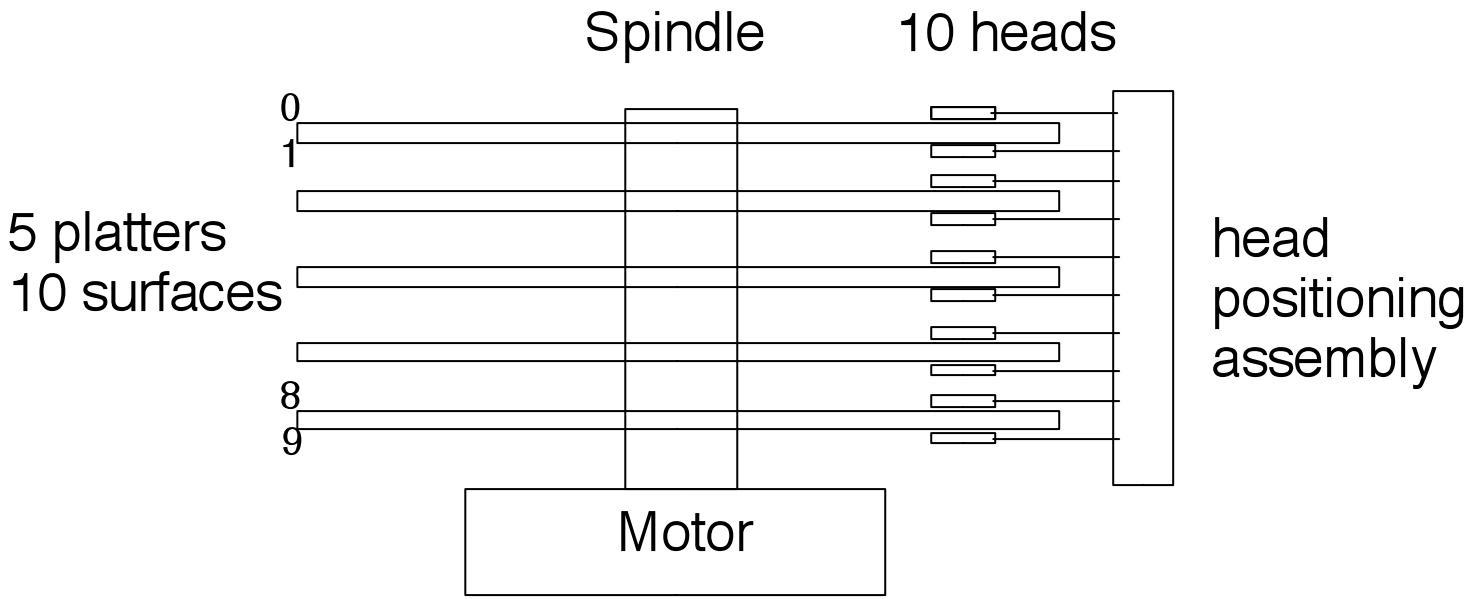
Disk Drives

- Still the primary method of providing stable storage
  - Storage meant to last beyond a single power cycle of the computer
  - Particularly for file systems
- Getting good performance from disk drives is critical for file system performance
- A place where physics meets computer science
  - Somewhat uncomfortably

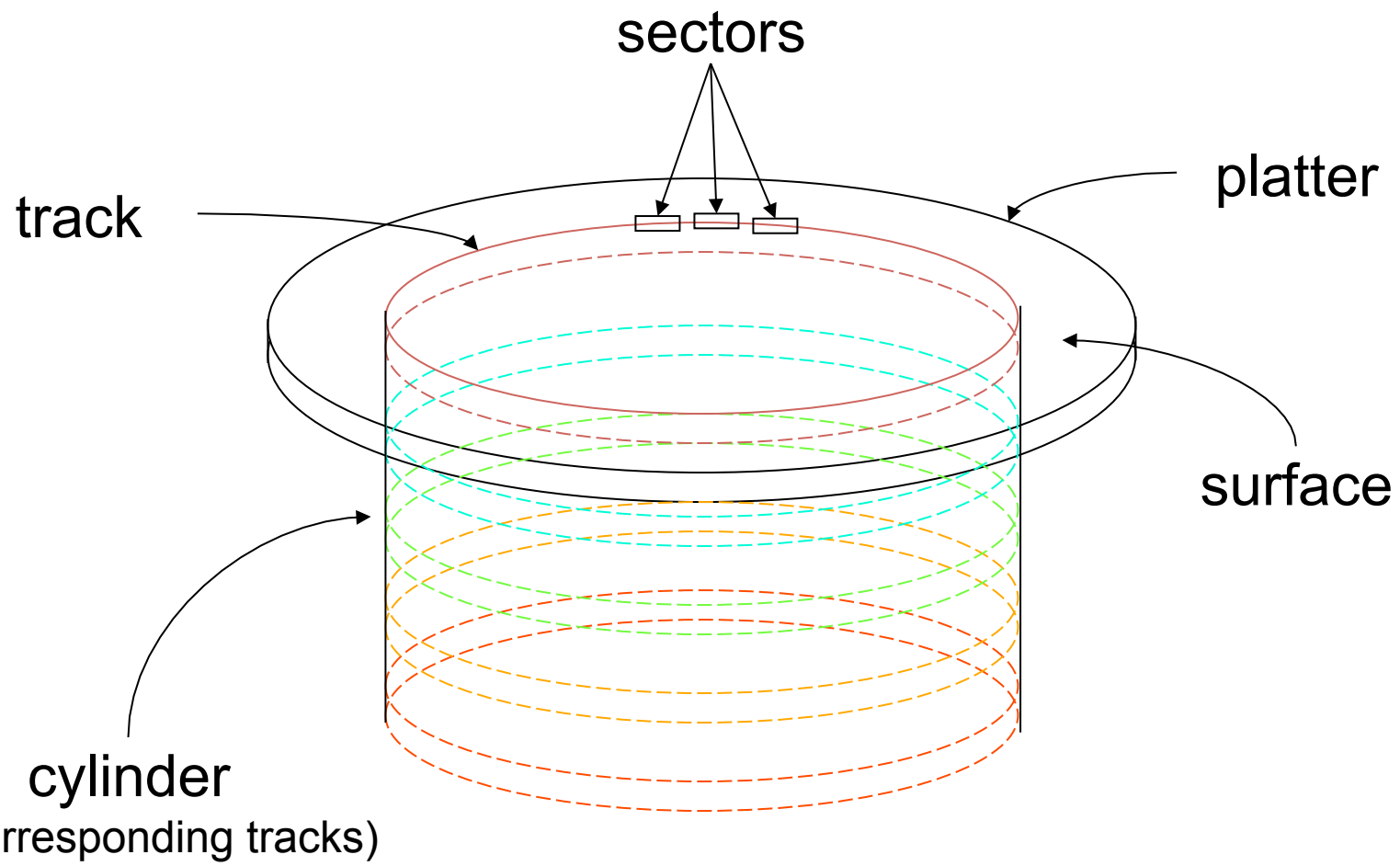
Some Important Disk Characteristics

- Disks are random access devices (mostly . . .)
  - With complex usage, performance, and scheduling
- Key OS services depend on disk I/O
  - Program loading, file I/O, paging
  - Disk performance drives overall performance
- Disk I/O operations are subject to overhead
  - Higher overhead means fewer operations/second
  - Careful scheduling can reduce overhead
  - Clever scheduling can improve throughput, delay

Disk Drives – A Physical View



Disk Drives – A Logical View



Disk Drive Terms

- *Spindle*
  - A mounted assembly of circular platters
- *Head assembly*
  - Read/write head per surface, all moving in unison
- *Track*
  - Ring of data readable by one head in one position
- *Cylinder*
  - Corresponding tracks on all platters
- *Sector*
  - Logical records written within tracks
- *Disk address* = <cylinder / head / sector >

Seek Time

- At any moment, the heads are over some track
  - All heads move together, so all over the same track on different surfaces
- If you want to read another track, you must move the heads
- The time required to do that is seek time
- Seek time is not constant
  - Amount of time to move from one track to another depends on start and destination
  - Usually reported as an average

Rotational Delay

- Once you have the heads over the right track, you need to get them to the right sector
- The head is over only one sector at a time
- If it isn't the right sector, you have to wait for the disk to rotate over that one
- Like seek time, not a constant
  - Depends on which sector you're over
  - And which sector you're looking for
  - Also usually reported as an average
- Also called *rotational latency*

Transfer Time

- Once you're on the correct track and the head's over the right sector, you need to transfer data
- You don't read/write an entire sector at a time
- There is some delay associated with reading every byte in the sector
- All sectors are usually the same size
- So transfer time is usually constant

Typical Disk Drive Performance

heads	10	platters	5
cylinders	17,000	tracks/inch	18,000
sectors/track	400	bytes/sector	512
RPM	7200	speed	196Mb/sec
seek time	0-15 ms	latency	0-8ms

Time to read one 8192 byte block

	seek	rotate	transfer	total
best case	0ms	0ms	333us	333us
worst case	15ms	8ms	333us	23.3ms (70X)
average	9ms	4ms	333us	13.3ms (40X)

Why Is This Problematic For the OS?

- When you go to disk, it could be fast or slow
  - If you go to disk a lot, that matters
- The OS can make choices that make it faster or slower
  - Deciding where to put a piece of data on disk
  - Deciding when to perform an I/O
  - Reordering multiple I/Os to minimize seek time and latency
  - Perhaps optimistically performing I/Os that haven't been requested