Chapters 10, 11 and 12: File Systems

CSCI 3753 Operating Systems

Prof. Rick Han

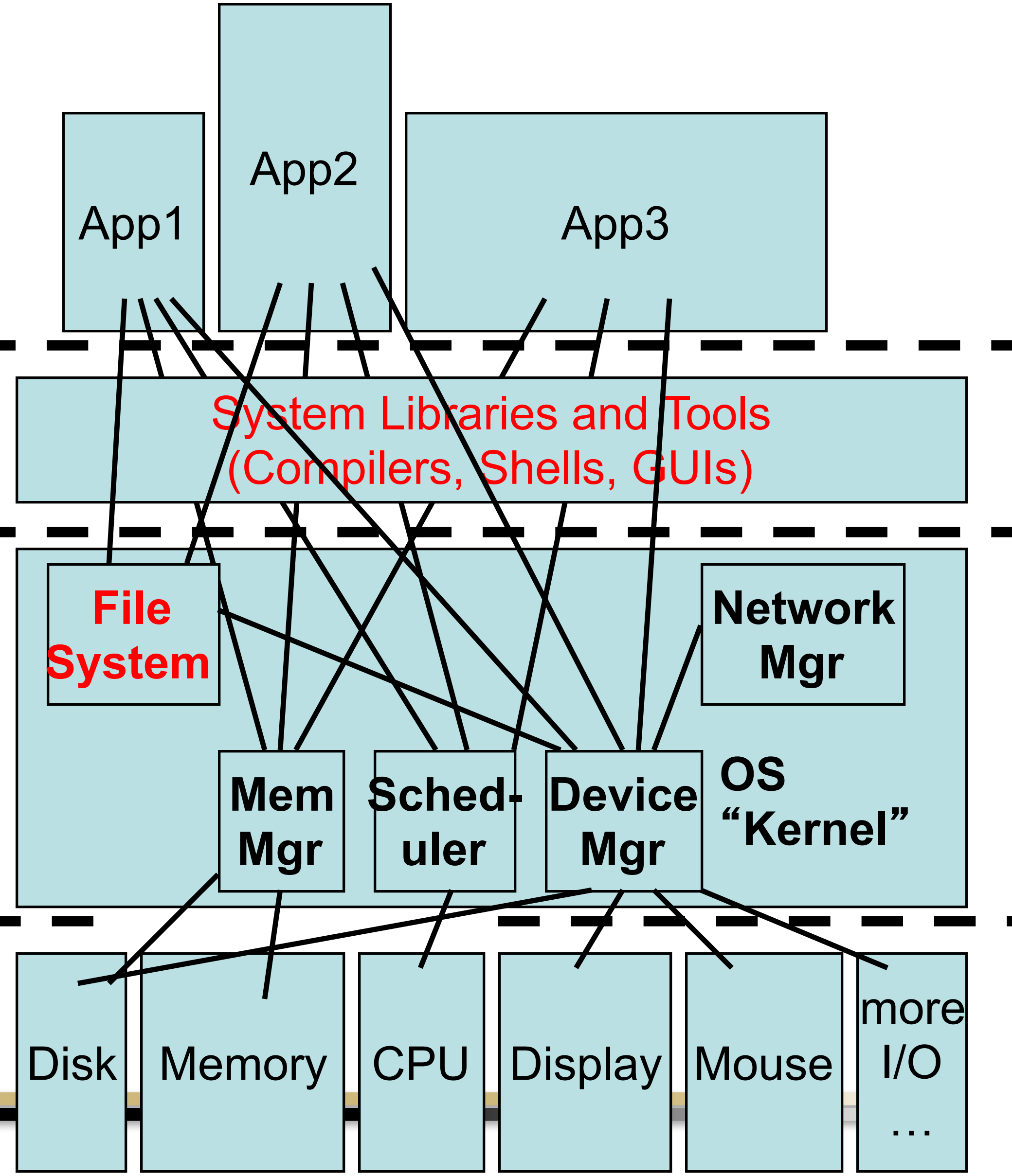
OS Architecture

Posix, Win32, Java, C library API

System call API

– >300 in Linux 3.1

Device driver “API”



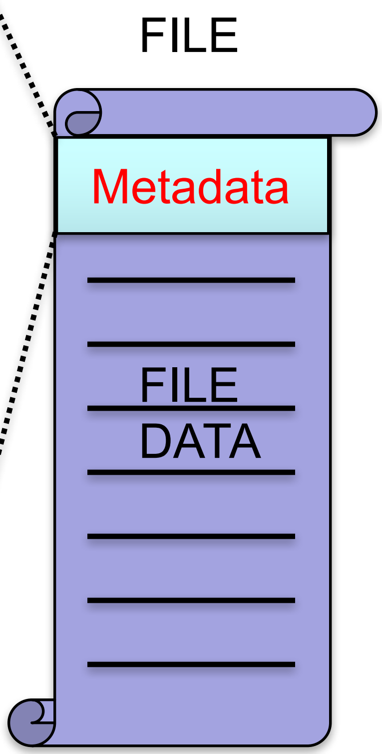
Note: different OS kernels can support the same system call API

File Systems

- What is a file?
  - Human perspective: a file is a logical storage unit or abstract data type that is conceptually useful to humans
  - OS perspective: a file is a sequence of bytes that is mapped by OS to a section of a physical storage device, e.g. disk or flash
    - UNIX views a file as a sequence of bytes
      - each byte is addressable by its offset from the beginning of the file
      - it is up to the application to interpret these file data bytes

File Systems

- A file consists of data and attributes:
  - a file's attributes include (usually not more than 1 KB of metadata):
    - name
    - unique ID
    - size
    - protection info - read/write/execute
    - timing info - when created, last modified, last accessed
    - location on permanent storage (disk or flash)
    - some OS's support a "type".
    - some OS's support a "creator" field that indicates which application created this file, e.g. MS Word or Adobe Acrobat.
  - These attributes are usually collected together and stored in a *file header* or *file control block* (FCB). In UNIX, this is called an *inode*.



File Systems

- The OS provides basic system calls to manipulate files:
  - create() - find space in the file system, and add file to the file system
  - read()
  - write()
  - both read and write can be either
    - *Sequential access* – remember where the last read or write was positioned in the file (the file position pointer), and start the next read or write from that position
    - Direct access (also called *random access*, or relative access) – given an offset n, go directly n bytes into the file to read or write

File Systems

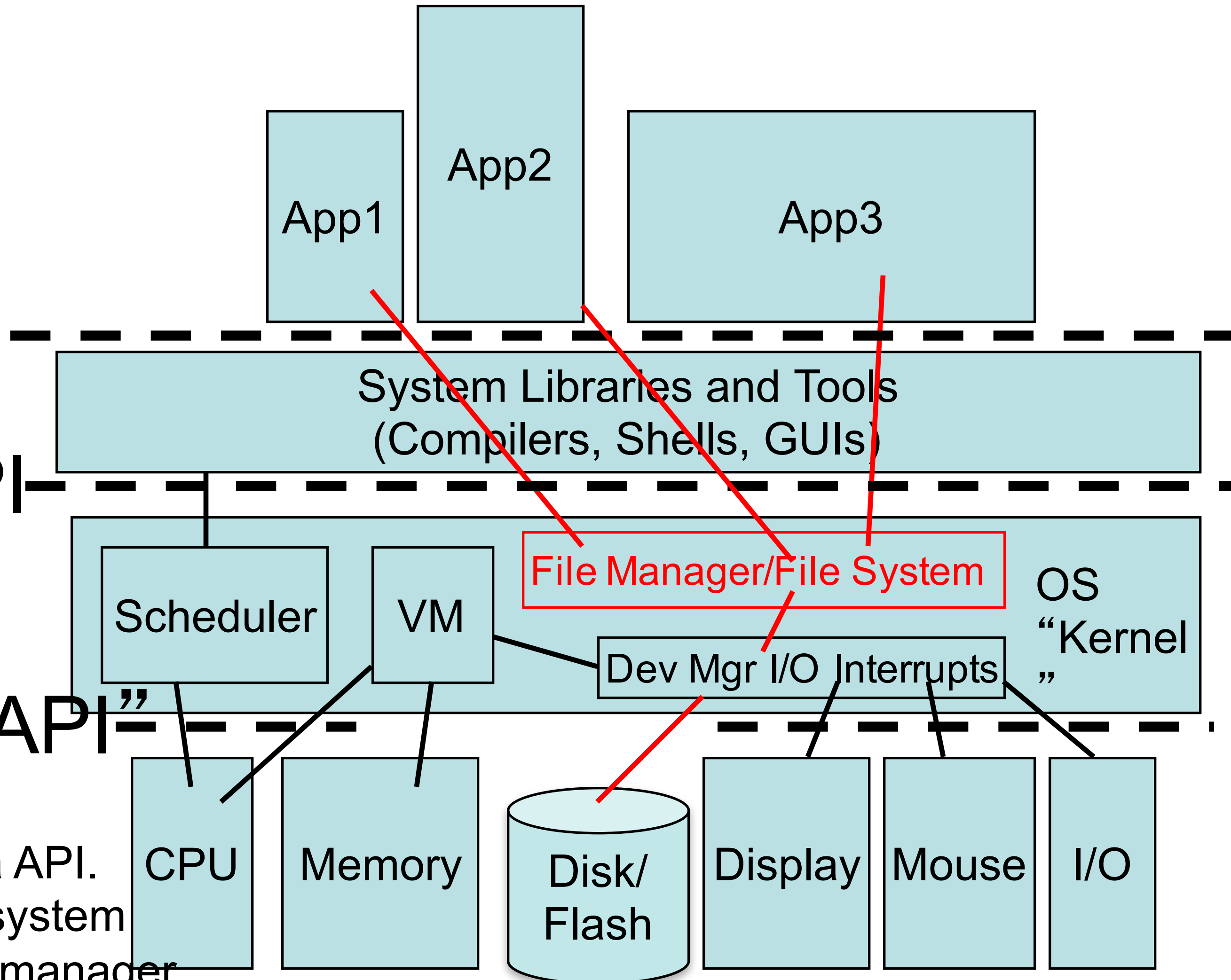
- The OS provides basic system calls to manipulate files: (cont.)
  - open() and close() makes it easy & fast to reference/name files
  - seek()/reposition() - update a *file-position pointer* during reads and writes of a file
  - delete()
  - truncate()
  - append()
  - rename()
  - copy(), move(), …

File Manager/File System

fopen(),fclose(),
fread(),fwrite(),

System call API

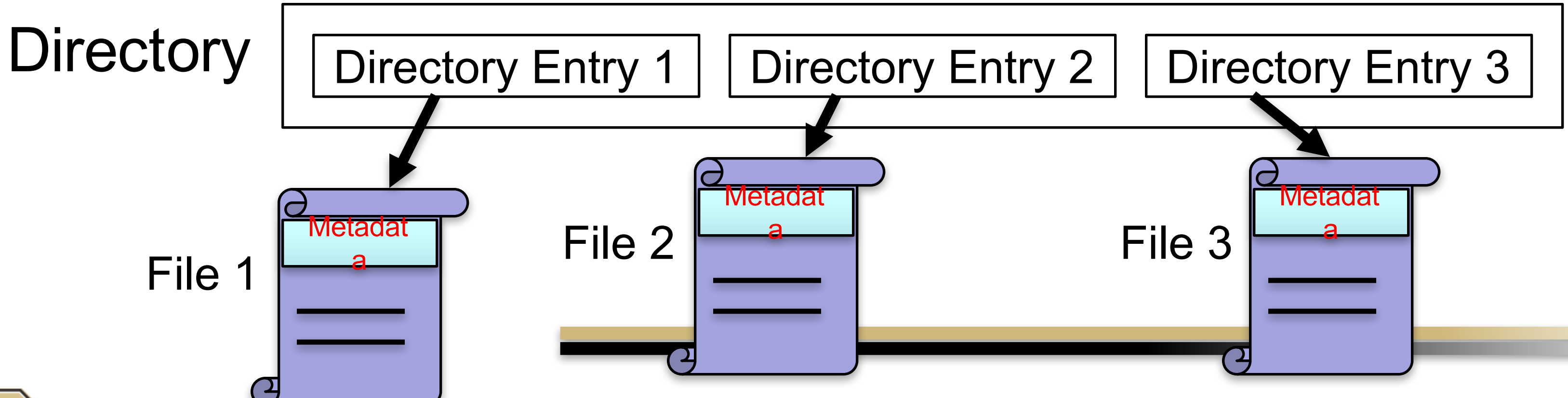
Device driver “API”



An app makes file calls via API.
These are translated into system
calls that invoke OS’ s file manager.
OS turns them into disk read/writes

Directories

- Files alone are not enough – humans need Directories to help organize files
  - a directory is a collection of entries that provide information for accessing files in that directory
  - each directory entry contains:
    - a file's (or another directory's) name
    - and unique ID for locating other file (or directory) attributes



Directories

- Some operations on directories:
  - list files in a directory
  - create a directory
  - delete a directory
    - MSDOS won't delete a directory if it contains files, i.e. the directory is not empty
    - UNIX has in the past allowed deletion of a directory that contains files (and all subdirectories and their files) by using the rm command, so the user must be careful
  - move, copy, or rename a directory
  - search for a file in a directory

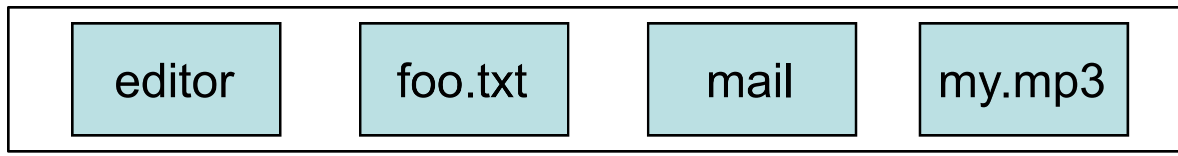
Directories

- File system stores information about all files and directories in a *directory structure* that is also stored on disk/flash
- Directory structures:
  1. Single Level (flat) Directory
  2. Two Level Directory
  3. Tree-structured Directories

Directories

- Single Level Directory
  – one directory in which all files are stored

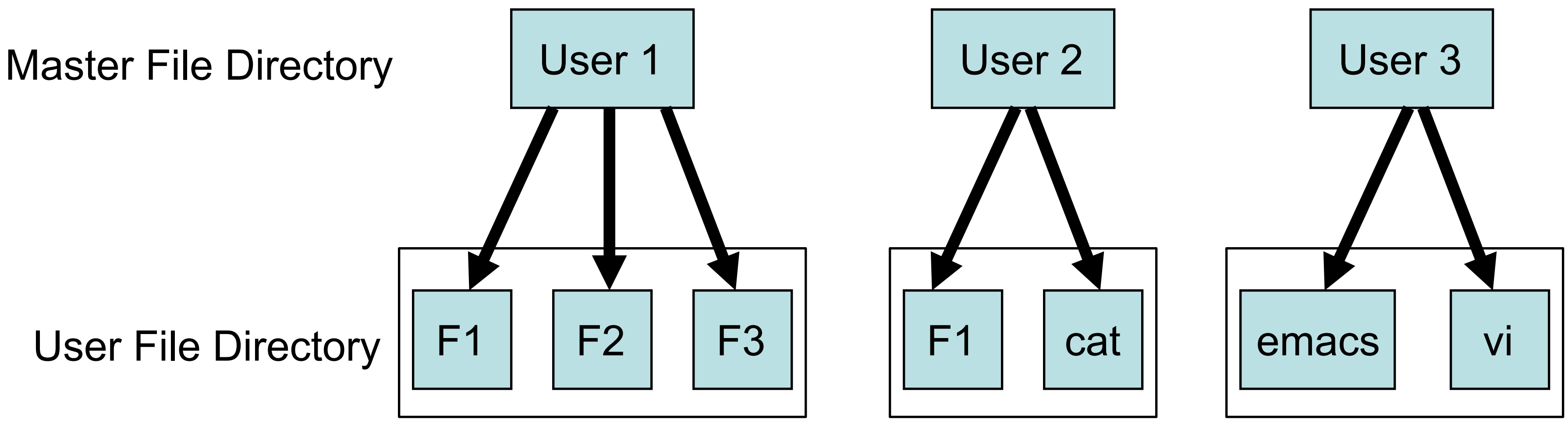
Directory



- Problems:
  – single user can't organize information clearly as the number of files grows
  – name conflicts among file names chosen by a single user
  – name conflicts among file names chosen by different users
  – multiple users have files all jumbled together

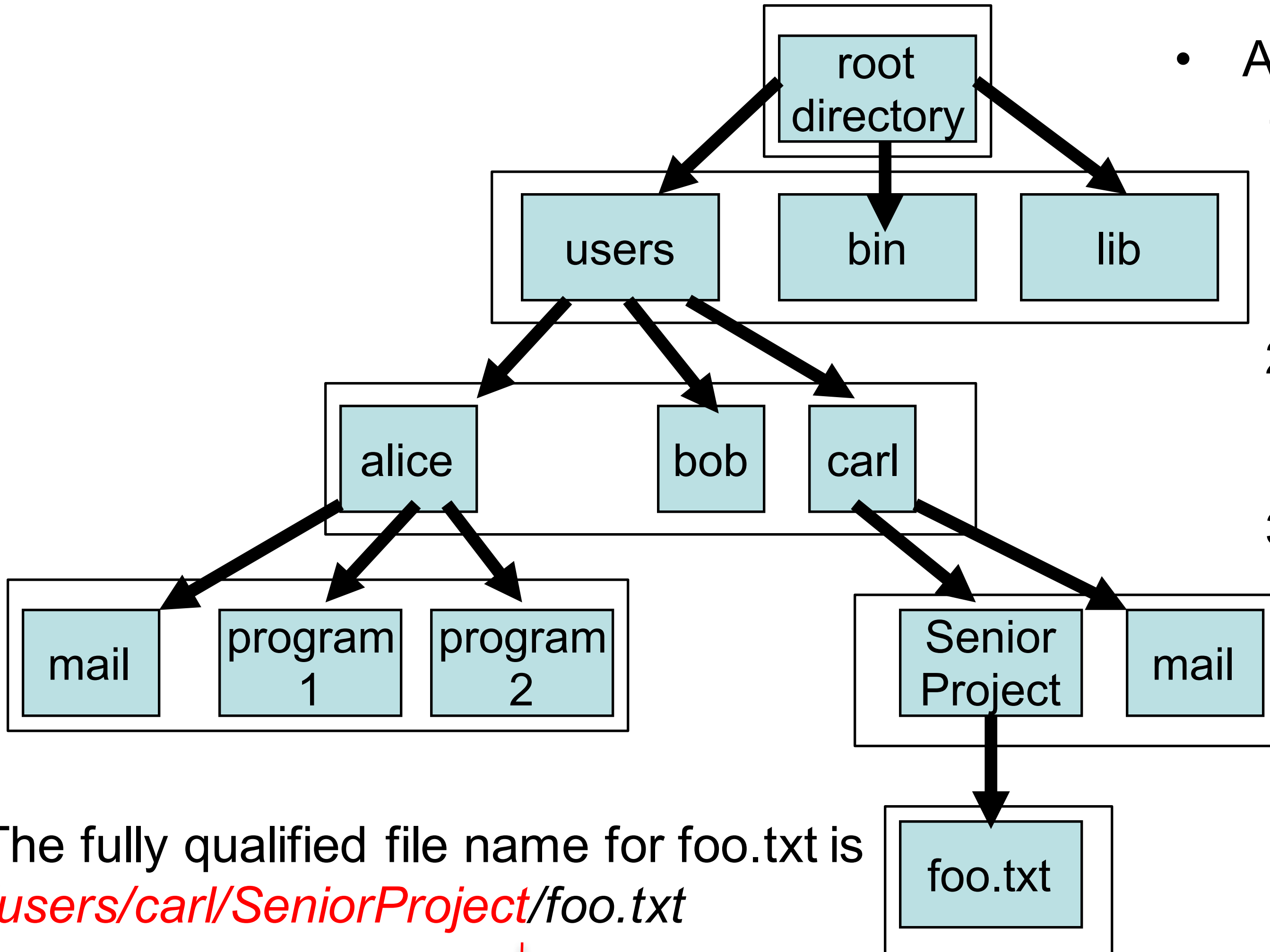
Directories

- Two Level Directory
  - Each user is given their own flat directory



- Fixes some problems of a universal flat directory, but there are still per-user flat directories

Tree-structured Directory



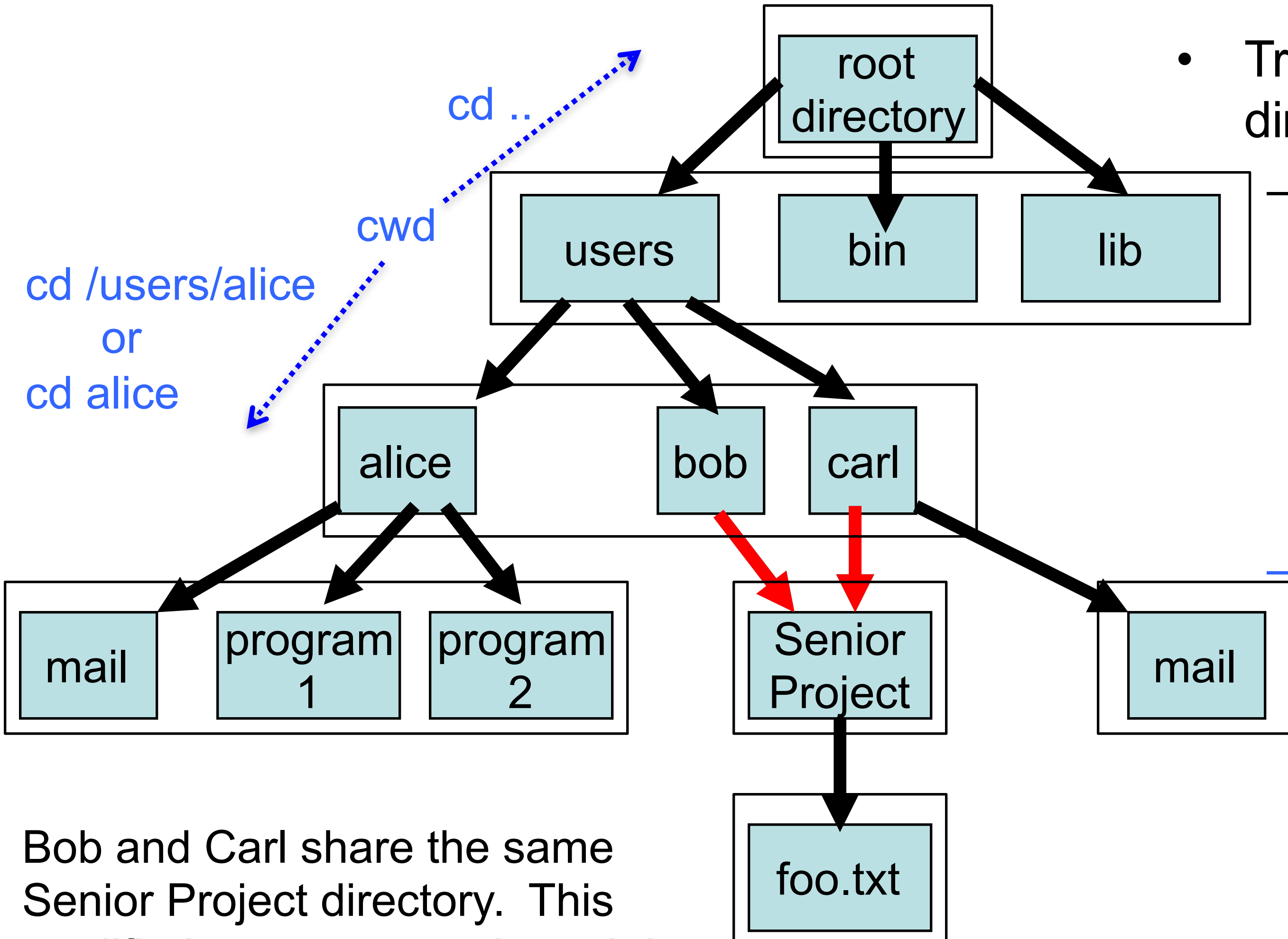
- Advantages:
  1. hierarchical & customized organization of files by each user
  2. Unique naming
     - no name conflicts
  3. users can share & access files in other directories

carl's file name "mail" does not conflict with alice's file name "mail"

The fully qualified file name for foo.txt is
/users/carl/SeniorProject/foo.txt

"path" or directory name

Tree-structured Directory



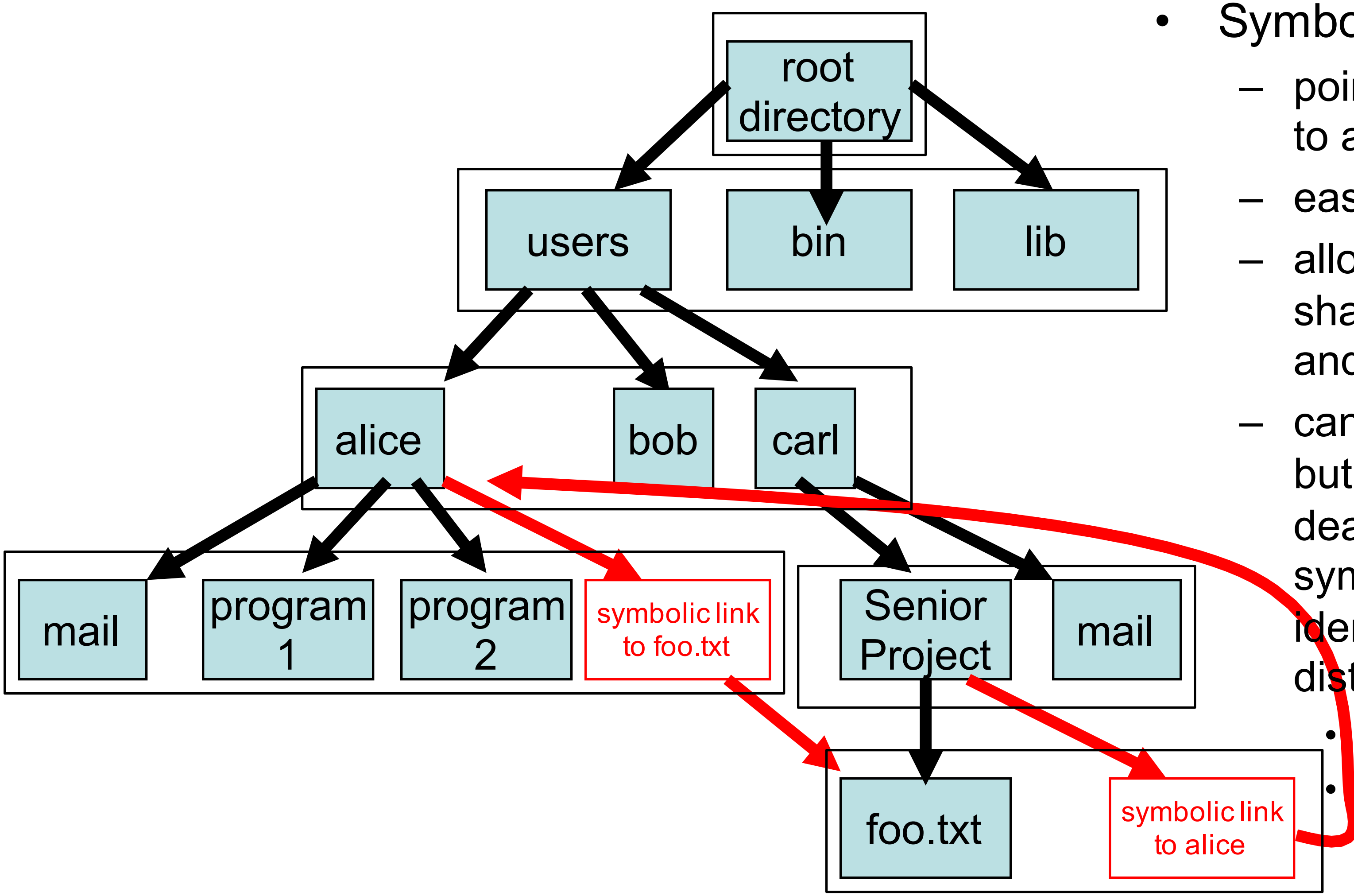
- Traversing the directory structure
  - each process keeps track of its *current working directory*
    - e.g. login shell is initialized to a user's home directory
  - change directory by specifying path name
    - absolute (full path)
    - relative
      - down: cd carl/
      - up: cd ../../

Bob and Carl share the same Senior Project directory. This modified tree-structured graph has no cycles and is termed an *acyclic graph*.

Sharing Directories & Files

- option 1: implemented via *symbolic links*
  – a symbolic link is a pointer to a directory entry, which in turn points to a file or directory
    - in UNIX, use "ln ___" to set up a symbolic link
    - On Mac, use "Make Alias" in CTRL-click menu
  – It is possible to create a cycle using symbolic links
    - e.g. symbolic link L1 in directory A points to directory B, while symbolic link L2 in directory B point to directory A
    - this is not a problem if handled correctly

Symbolic Links



- Symbolic links
  - pointer or reference to a file
  - easy to implement
  - allow convenient sharing of files and/or directories
  - can create loops, but these can be dealt with, because symlinks are easily identifiable as distinct from files
    - by their name
    - or by a special "type" if supported by OS

Symbolic Links

- a symbolic link is not a file, it is a pointer to a file
  - so operations on a link behave differently than operations on a file
- when searching for a file through the directory tree, the OS needs to avoid cycles, because otherwise it will search endlessly
  - One policy is to avoid traversing any symbolic links. This policy avoid cycles
  - or the OS could keep a record of all visited directories to avoid revisiting the same directory – expensive!

Symbolic Links

- when deleting a link, the file pointed to is not deleted
- when deleting a file
  - can leave symbolic links dangling, and leave it to the user to clean up dangling links - this is the policy of Windows, UNIX
  - Or… (see next slide)

Symbolic Links

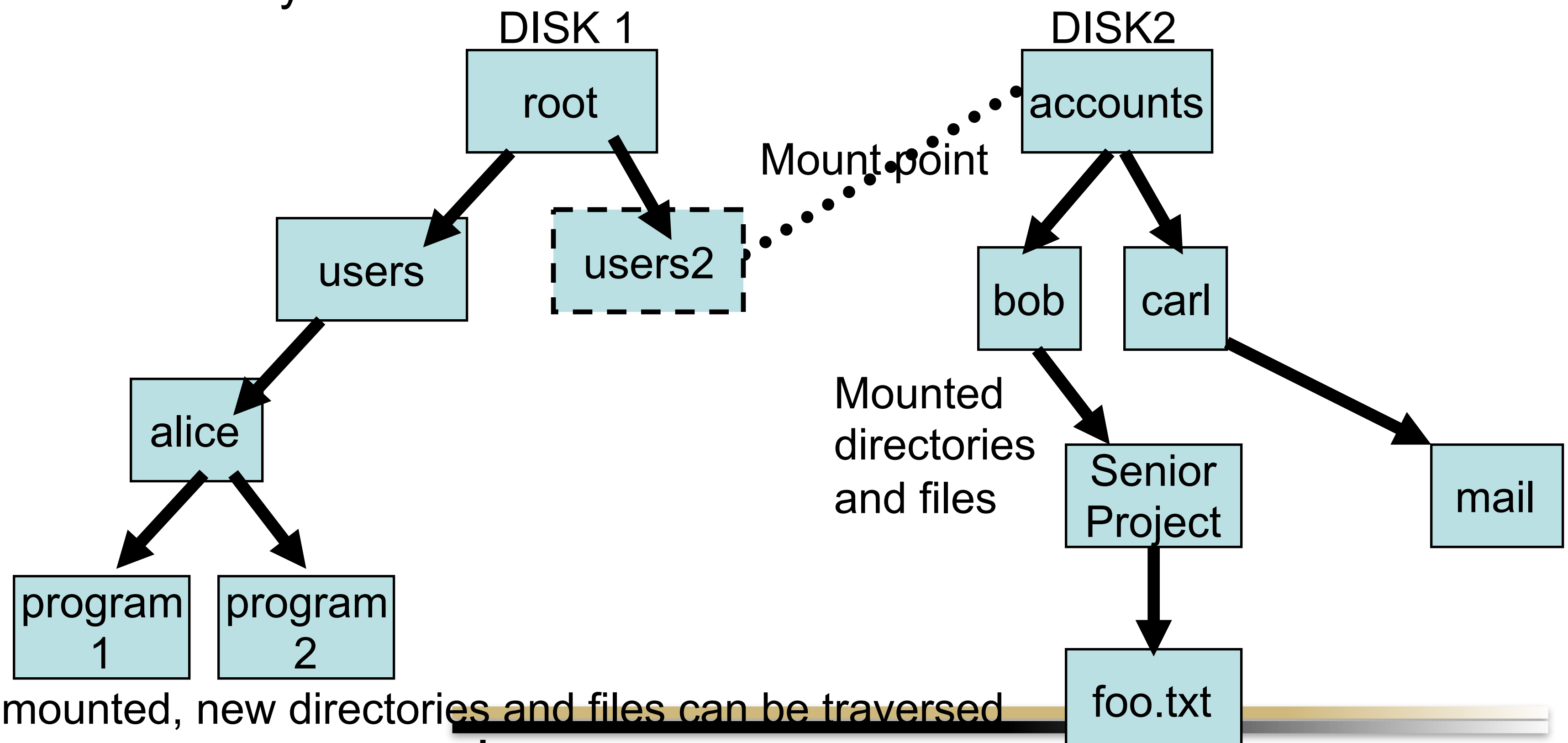
- when deleting a file (cont.)
  - the file is not deleted until all links pointing to the file are also deleted. OS keeps a count of links.
    - Each time a link to a file is added, count++. Each time a link is deleted, count--. When the count = 0, then the file can be safely deleted.
    - When UNIX uses this approach for a link, then the links are termed *hard links*, and the reference count is kept in the FCB or inode.

Sharing Directories & Files

- option 2: implemented by duplicating directories
  - hard to maintain consistency
- option 3: implemented by setting permissions so users can access the same files/directories

Mounting File Systems

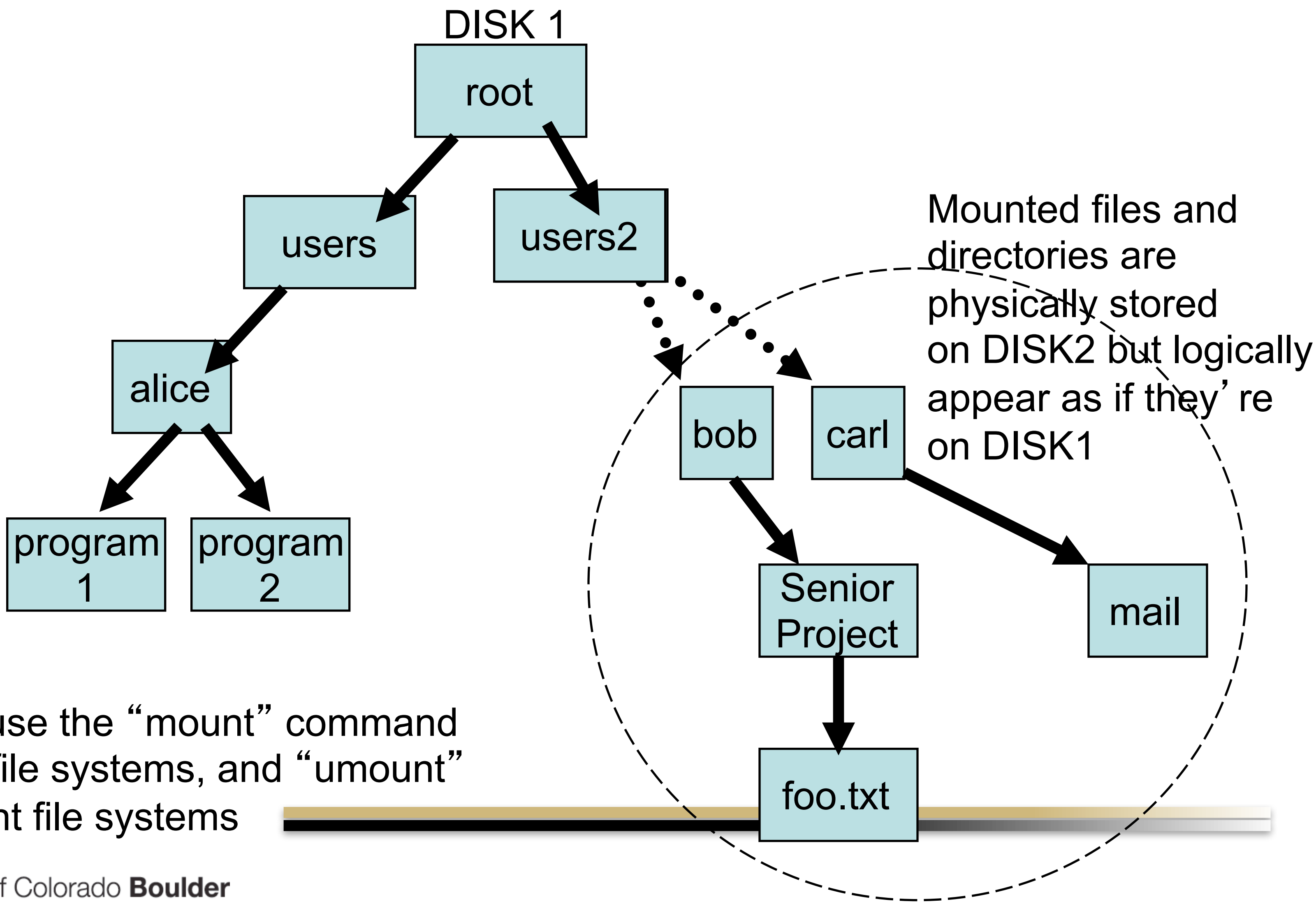
- Want to share files within the same directory structure though some files may be stored on different disks, or different partitions within a disk
  - *Mount* these new file systems so they appear within your current directory structure



Once mounted, new directories and files can be traversed just like any of the original tree's directories and files

Mounting File Systems

Final result of file mounting



In Linux, use the "mount" command to mount file systems, and "umount" to unmount file systems

Mounting File Systems

- Example: to mount a remote directory, say the /xfs filesystem at home.colorado.EDU, as a local directory /xfs, type:

  ```
  mkdir  /xfs
  /bin/mount  home.colorado.edu:/vol/xfs  /xfs
  ```
- When a file system is no longer needed, you can unmount the file system

Mounting File Systems

- Ideally, you can mount the new file system anywhere within the current directory tree
  - Unix follows this flexible approach.
    - The Unix file manager keeps track of what file systems are mounted in which directory by setting a flag in the in-memory copy of the inode for that directory. The flag indicates that the directory is a mount point.
    - A field then points to an entry in the mount table, indicating which device is mounted there.
    - The mount table entry contains a pointer to the disk location of the mounted file system.

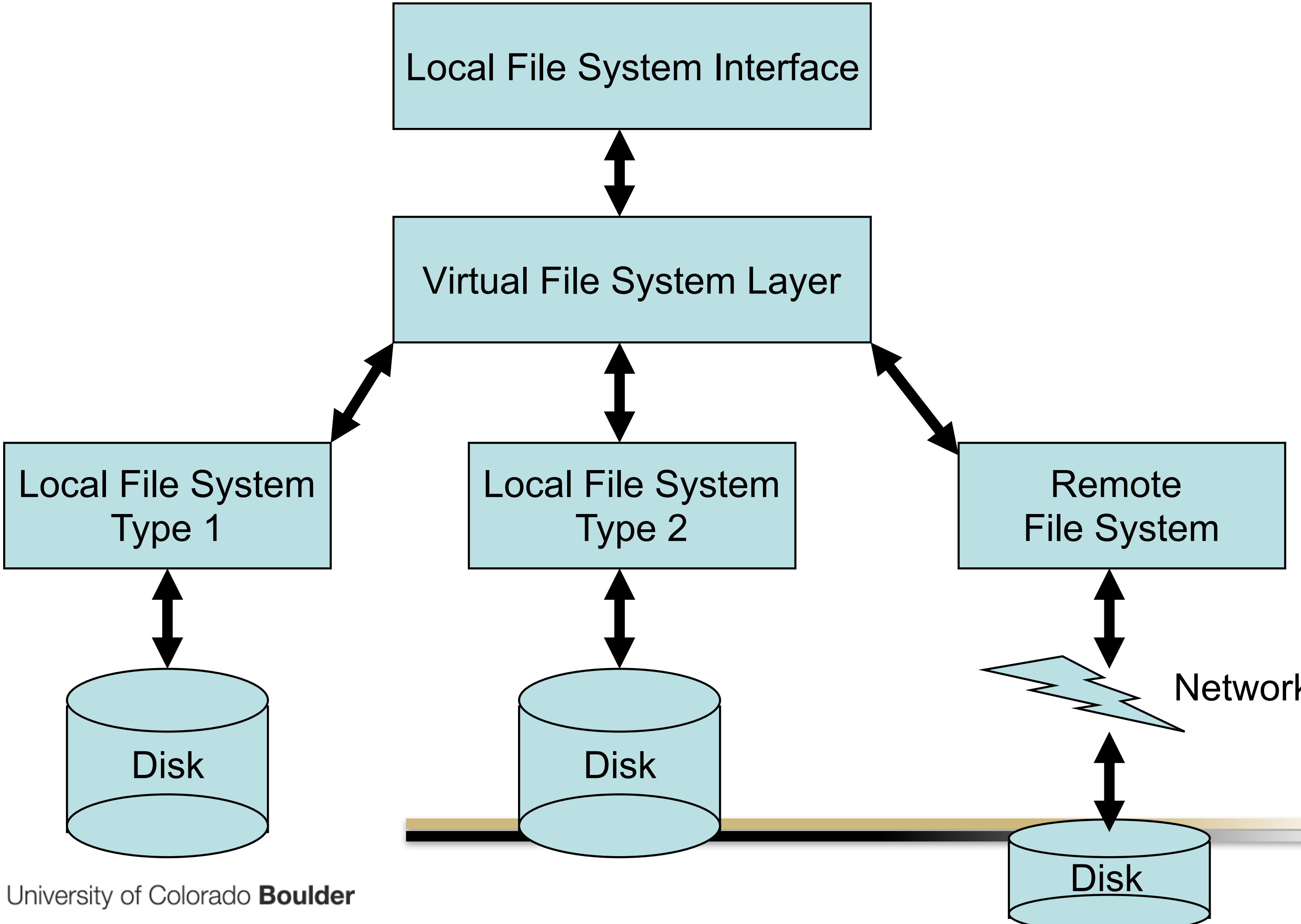
Mounting File Systems

- Ideally, you can mount the new file system anywhere within the current directory tree (cont.)
  - Windows mounts a new device containing a file system at the top level, e.g. D:\ or F:\, though later versions also allow mounting anywhere
  - Mac OS mounts a new device with a file system, e.g. USB stick, at the root level and adds a folder icon on the screen

Virtual File Systems

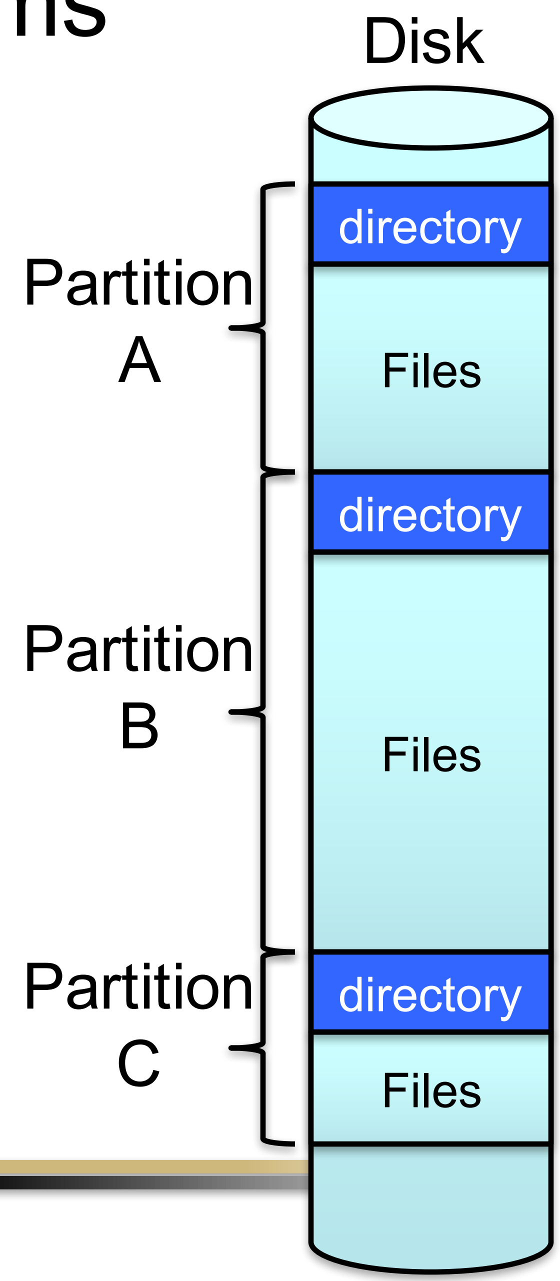
- In the most general case, the mounted file system could be of a different type than the current OS file system – how does the OS manage this heterogeneity?
  - Solution: OS can implement a *virtual file system* (VFS) layer that abstracts file representation and manipulation
  - VFS layer specifies an abstract model of a file and directory and abstract operations on files and directories
- The VFS translates abstract read()/write()/… operations to/from the specific language of read/write/… that is understood by each mounted file system
- Note, the mounted file system need not be local, i.e. it could be remote across the network!
  - Distributed file systems – the VFS handles this as well

Virtual File Systems



Multiple File Systems

- A typical disk may have multiple partitions
  - Each partition may contain a separate file system, and possibly OS as well
  - Each file system will have its own directory structure to keep track of its files
  - A file system may also span multiple disks (e.g. RAID, not shown)
- Other I/O devices may contain their own file systems
  - e.g. a USB flash drive
- Want to share these files…



File System Implementation

- File system elements are stored on *both*:
  - Disk/flash – persistent storage
  - Main memory/RAM – volatile storage
- On *disk/flash*, the entire file system is stored, including 5 main elements:
  1. its entire directory tree structure
  2. each file's file header/FCB/inode
  3. each file's data
  4. a *boot block*, typically the first block of a volume, that contains info needed to boot an operating system from this volume. Empty if no OS to boot.
  5. a *volume control block* that contains volume or partition details, e.g. tracks free blocks on disk, the number of blocks in a partition, size of a block, etc.

example FCB

name
unique ID
file permissions
dates (created,...)
size
location on disk

File System Implementation

- In memory/RAM, the OS file manager maintains only a subset of open files and recently accessed directories
  - memory is used as a cache to improve performance.
  - All the information is available for a fast search of memory, rather than a slow search of disk, e.g. for a file's FCB.
- The four main file system components in memory are:
  1. recently accessed parts of the directory structure tree are stored in memory

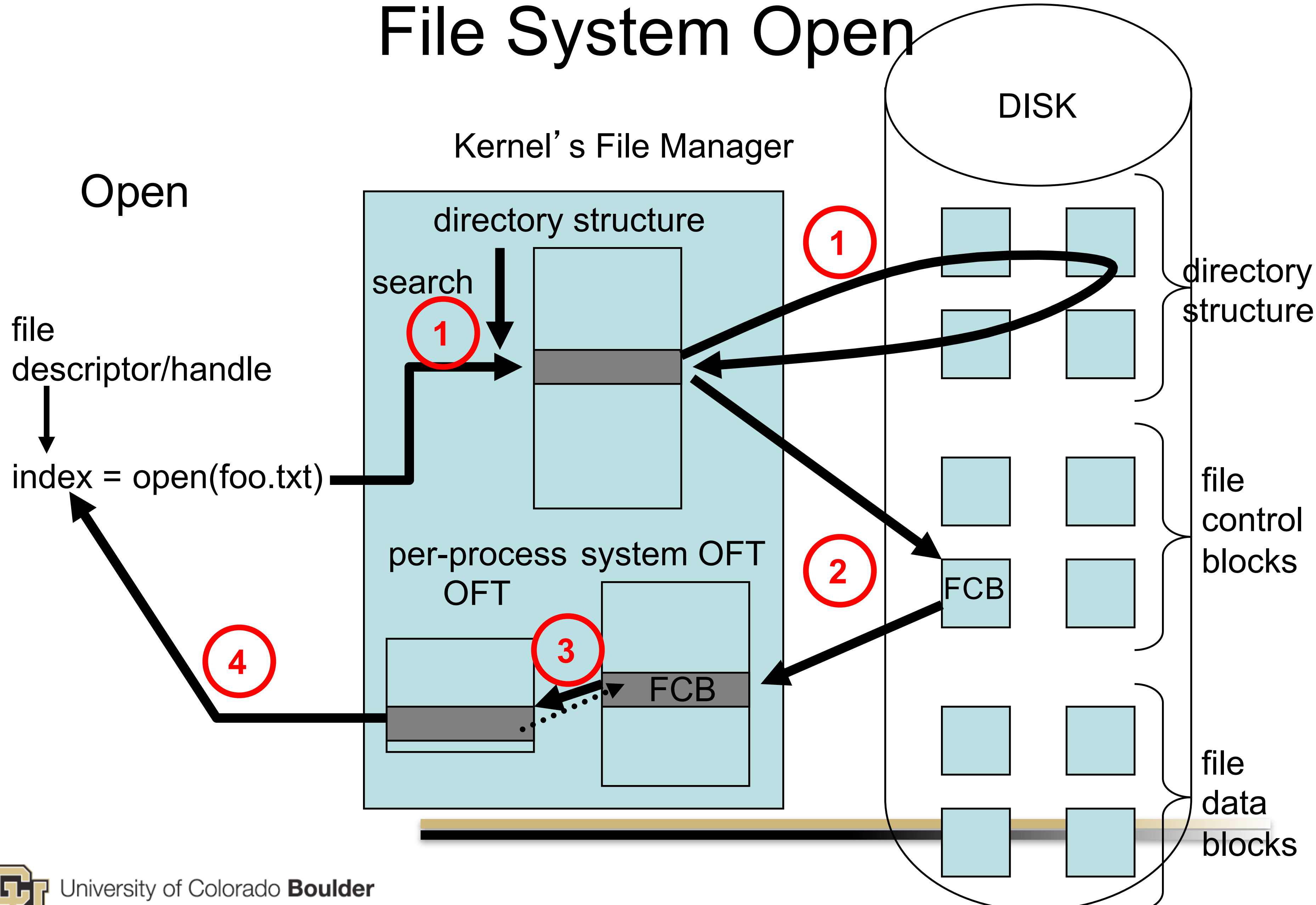
File System Implementation

- The four main file system components in memory are: (cont.)
  2. the OS also maintains a *system-wide open file table* (let's abbreviate it OFT) that tracks process-independent info of open files
     - the file header containing attributes about the open file is stored here
     - an open count of the number of processes that have a file open is stored here
  3. the OS also maintains a *per-process OFT* - tracks all files that have been opened by a particular process, may store access rights, etc.
     - Also keeps a current-file-position pointer, i.e. where in the file the process is currently reading/writing

File System Implementation

- The four main file system components in memory are: (cont.)
  4. OS keeps a mount table of devices with file systems that have been mounted as volumes
     - We'll use the terms "volumes" and "partitions" interchangeably, though technically a volume may be spread across different disk partitions on different disks, e.g. in RAID disk systems

File System Open



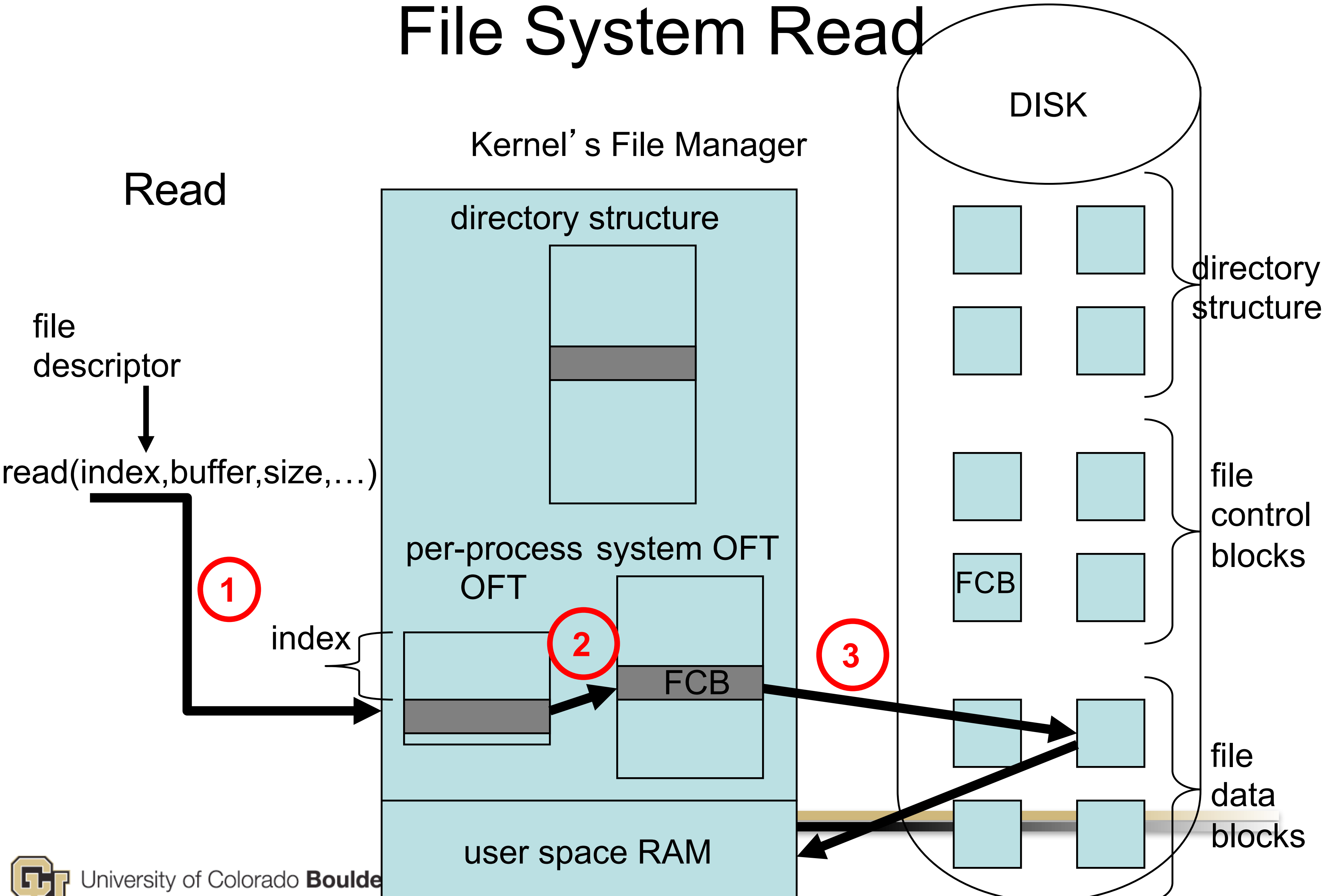
File System Open

- when a process calls open(foo.txt) to set up access to a file, the following procedural steps are followed:
  1. the directory structure is searched for the file name foo.txt
     - if the directory entries are in memory, then the search is fast
     - otherwise, directories and directory entries have to be retrieved from disk and cached for later accesses
  2. once the file name is found, the directory entry contains a pointer to the FCB on disk
     - retrieve the FCB from disk
     - copy the FCB into the system OFT. This acts as a cache for future file opens.
     - Increment the open file counter for this file in the system OFT
  3. add an entry to the per-process OFT that points to the file's FCB in the system OFT
  4. return a file descriptor or handle to the process that called open()

File System Open

- some OS’s employ a mandatory lock on an open file so that only one process at a time can use an open file, e.g. Windows
- other OS’s allow optional or advisory locks, e.g. UNIX. In this case, it’s up to users to synchronize access to files

File System Read



File System Close

- on a close(),
  1. remove the entry from the per-process OFT
  2. decrement the open file counter for this file in the system OFT
  3. if counter = 0, then write back to disk any metadata changes to the FCB, e.g. its modification date
     - Note: there may be a temporary inconsistency between the FCB stored in memory and the FCB on disk – designers of file systems need to be aware of this. A similar inconsistency occurred for modified memory-mapped file data in RAM that had not yet been written to disk.