CSCI 3753
Operating Systems

File System

Chapters 11 and 12

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder

Last Update: 11/02/16

File Systems

- What is a file?
  - Human perspective: a file is a logical storage unit to store some semantically related information
  - 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
    - 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

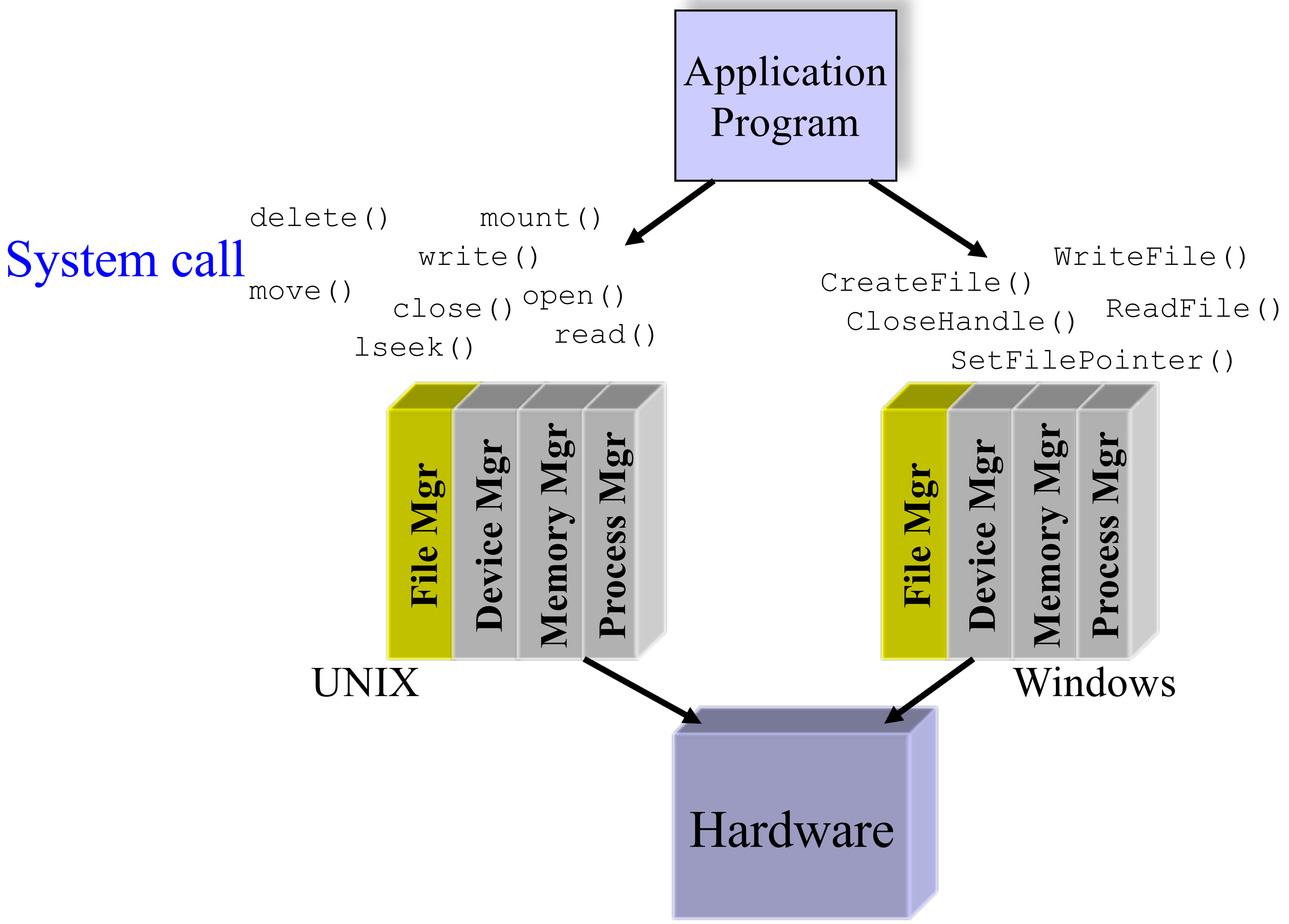
File Systems

- A file consists of data and attributes:
  - Attributes: Name, size, protection info (read/write/execute), timing info (when created, last modified, last accessed)
  - Attributes: location on permanent storage (disk or flash)
  - Attributes: some OS's support a "type"
    - In UNIX, an optional magic number is used to indicate a type. Not all applications adhere to this convention, and the OS does not enforce types, viewing them merely as hints
  - Attributes: 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*

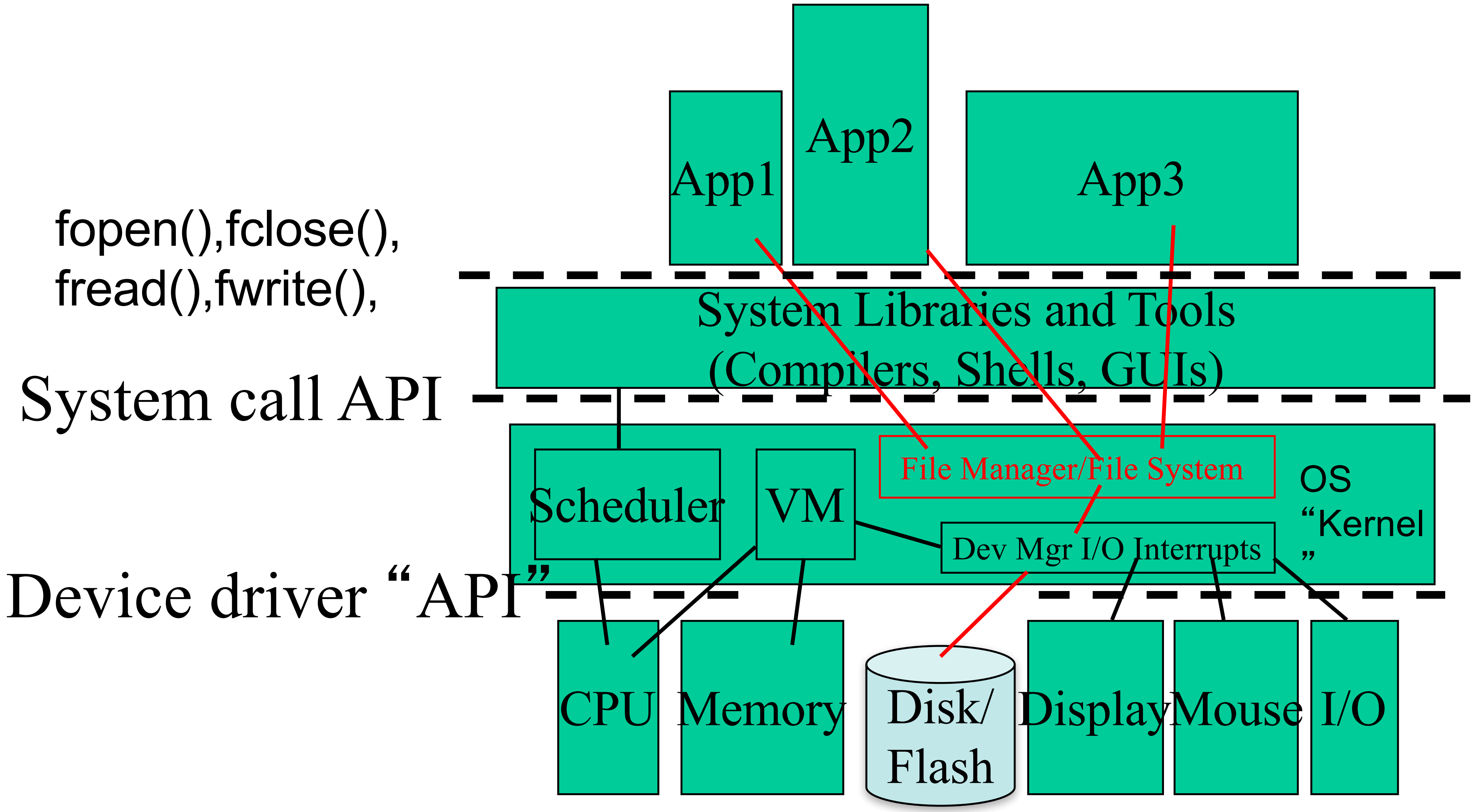
Directories

- A set of logically associated files and sub directories
- How should files be organized within directory?
  - Flat name space
    - All files appear in a single directory
  - Two level name space
    - Each user is given their own flat directory
  - Hierarchical name space
    - Directory contains files and subdirectories
    - Each file/directory appears as an entry in exactly one other directory -- a *tree*
    - Popular variant: All directories form a tree, but a file can have multiple parents.

High Level View



File Manager/File System



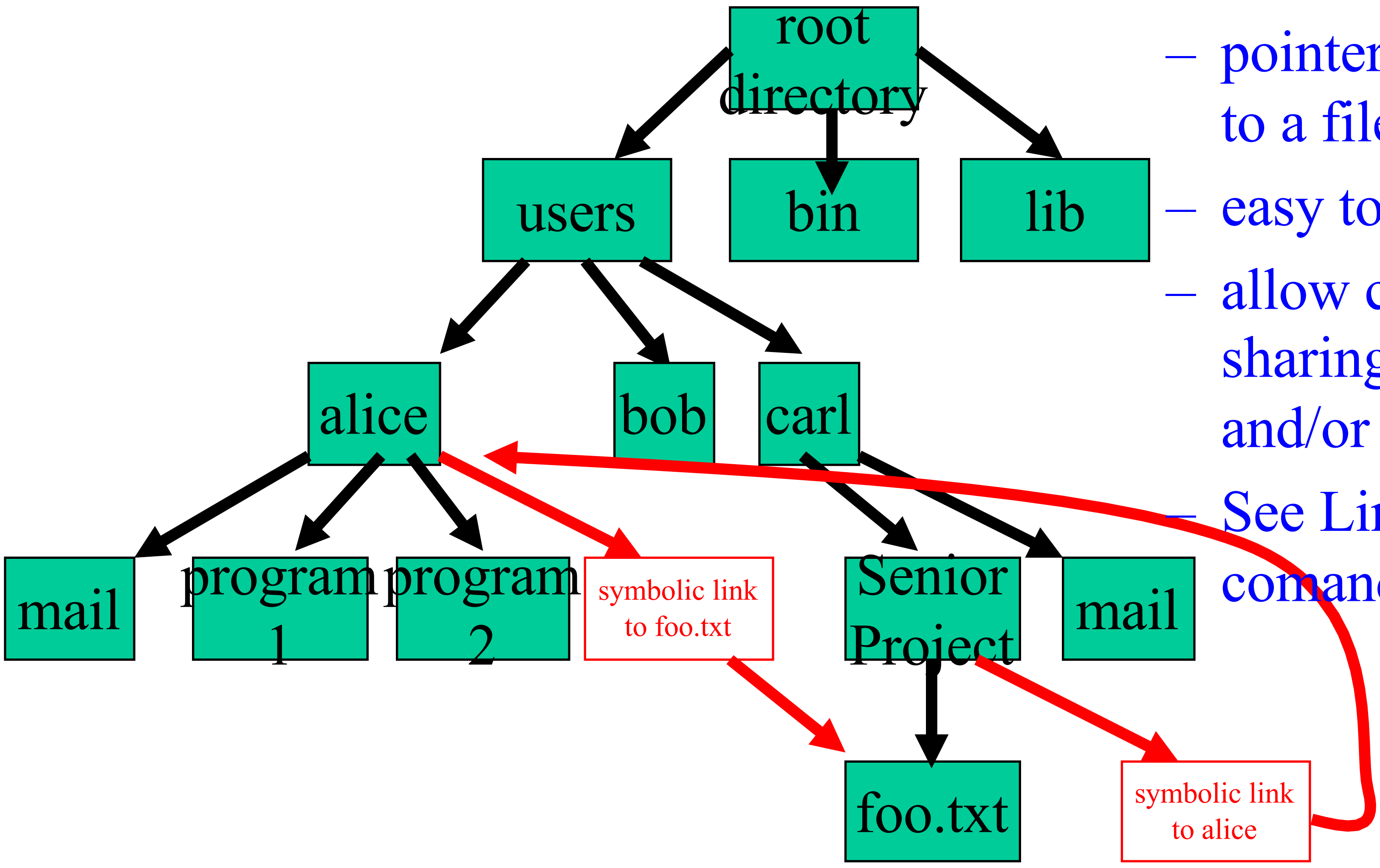
File System

- 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

Sharing Files/Directories

- Three options
  - Option 1: Symbolic links
    - A symbolic link is a pointer to a directory entry, which in turn points to a file or directory
  - Option 2: Duplicating directories
    - Hard to maintain consistencies
  - Option 3: Setting permissions

Symbolic links

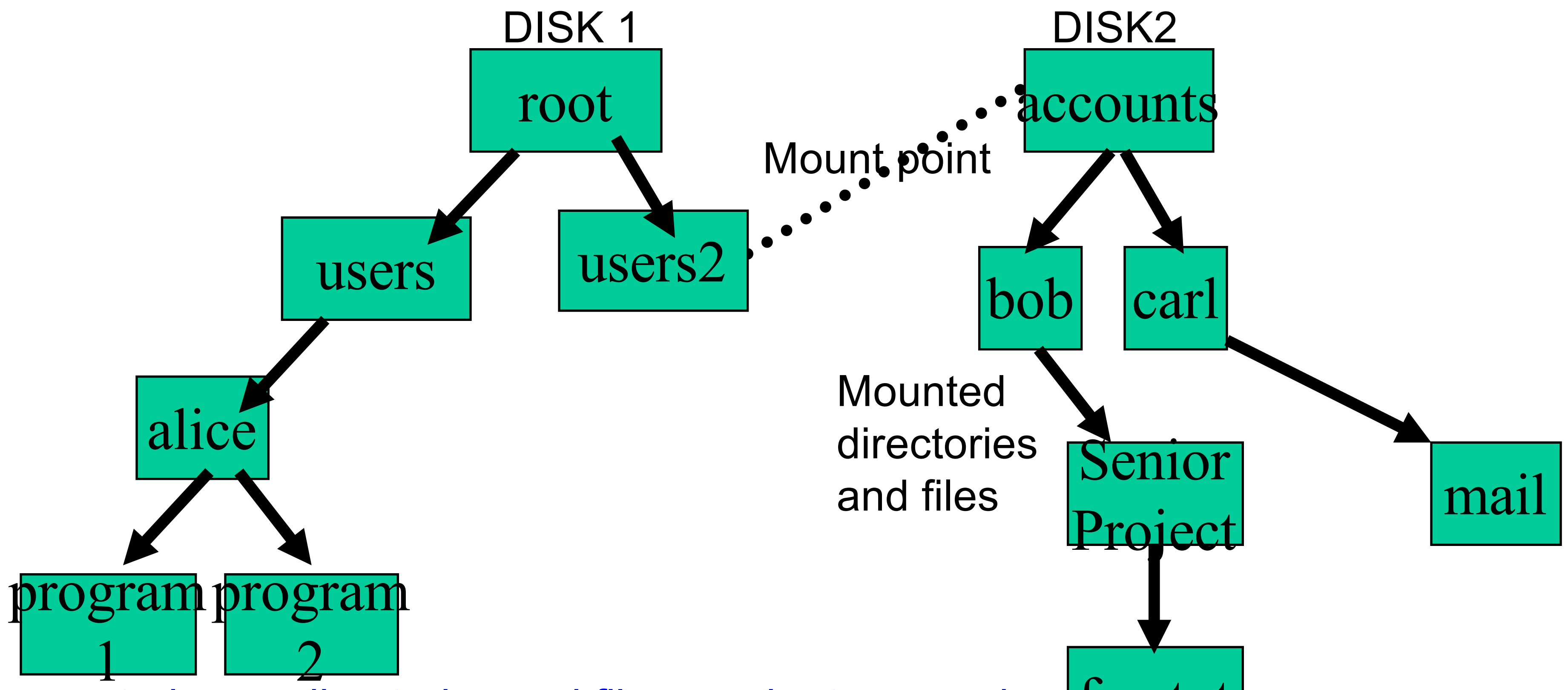


- pointer or reference to a file
- easy to implement
- allow convenient sharing of files and/or directories
- See Linux ln comand

Symbolic links can create loops, but these can be dealt with, because symlinks are easily identifiable as distinct from files

Mounting File Systems

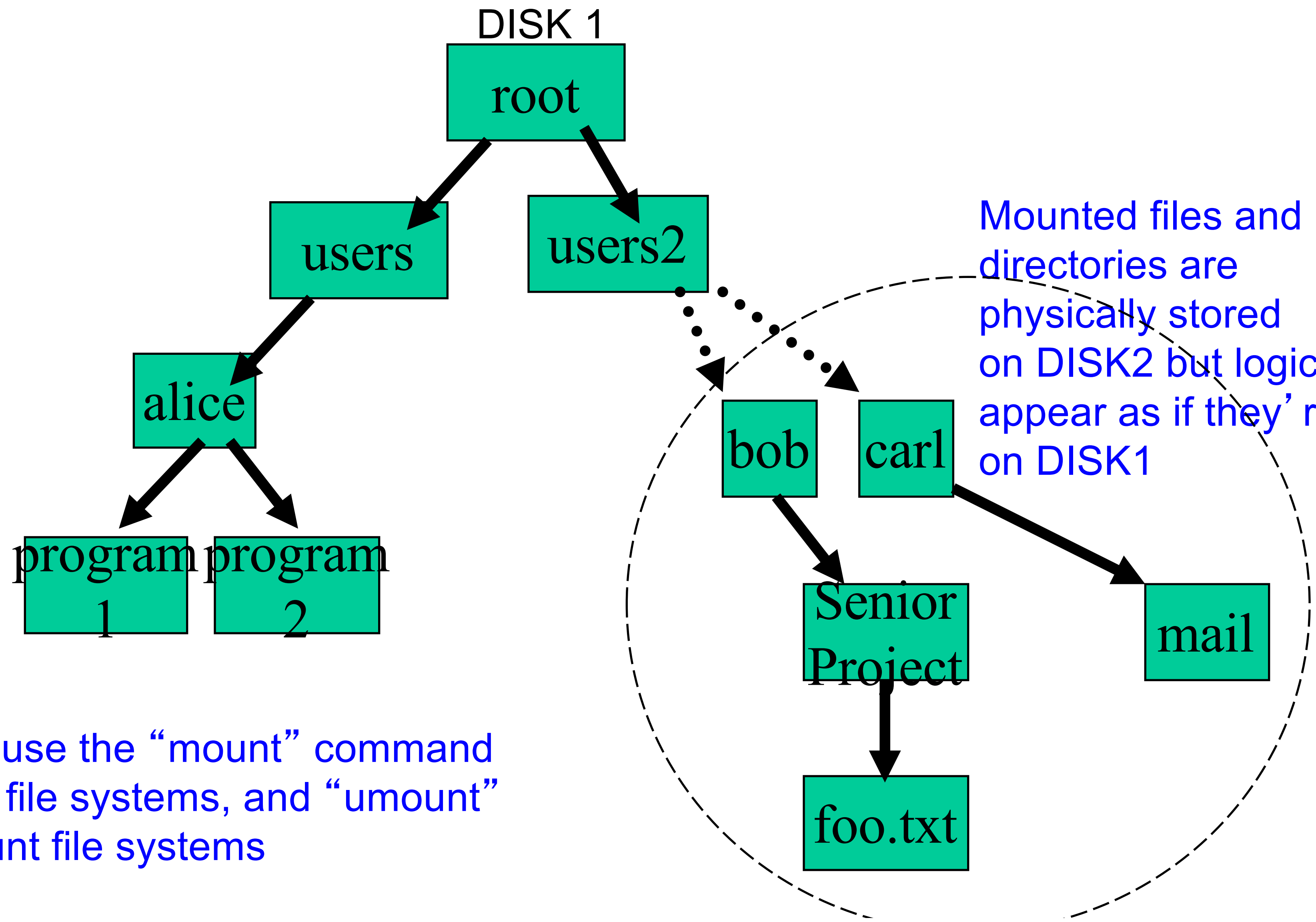
- Want to share files within the same directory structure though some files may be stored on different disks, or different partitions within a disk
  - *Mount* these new file systems so they appear within your current directory structure



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



Mounted files and directories are physically stored on DISK2 but logically appear as if they're on DISK1

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

Mounting File Systems

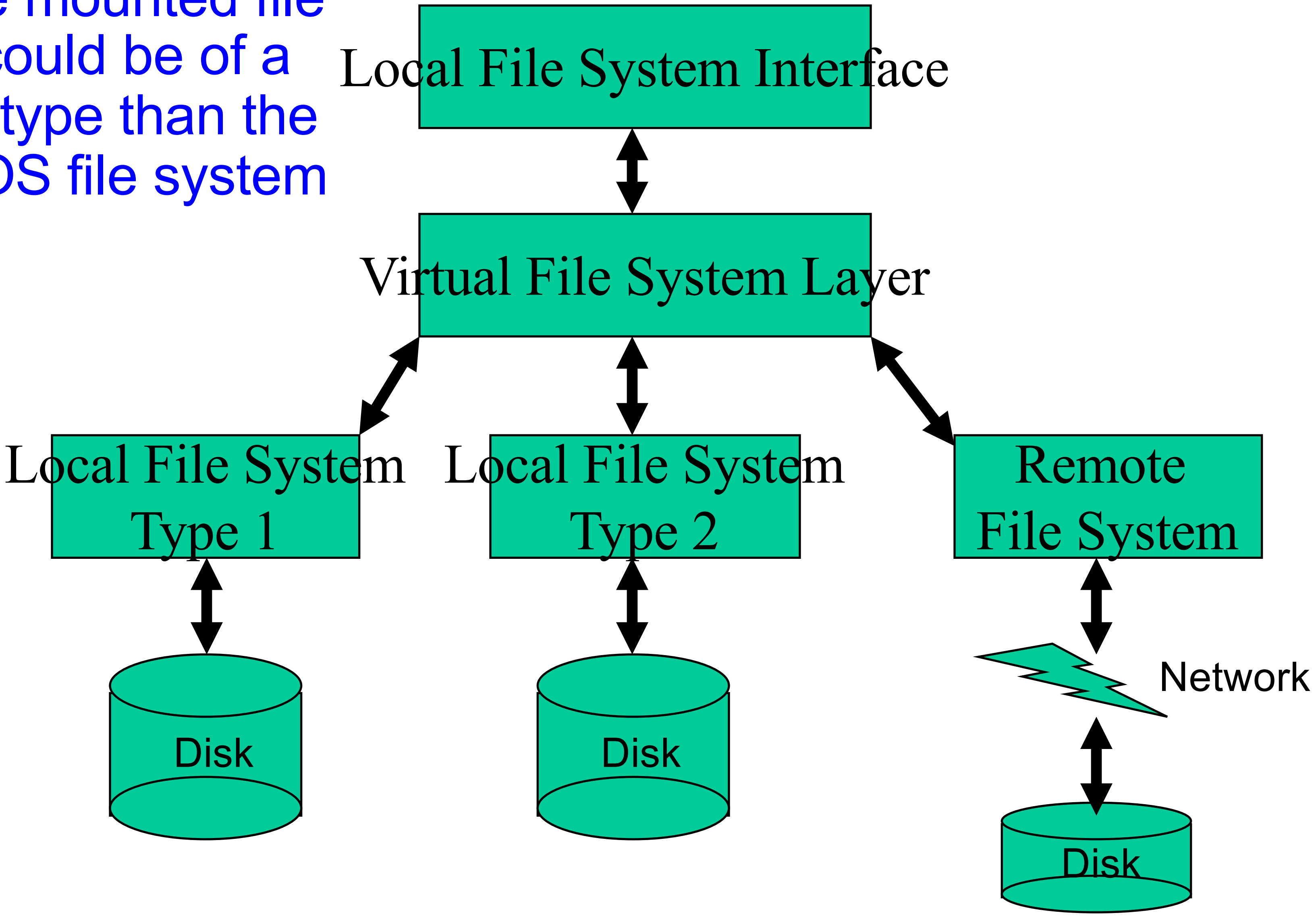
- To mount a remote directory, say the */xfs* filesystem at *home.colorado.EDU*, as a local directory */xfs*:

```
mkdir /xfs
/bin/mount  home.colorado.edu:/vol/xfs  /xfs
```

- In Unix, you can mount the new file system anywhere within the current directory tree
- 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
- When a file system is no longer needed, you can unmount the file system

Virtual File Systems

In the most general case, the mounted file system could be of a different type than the current OS file system

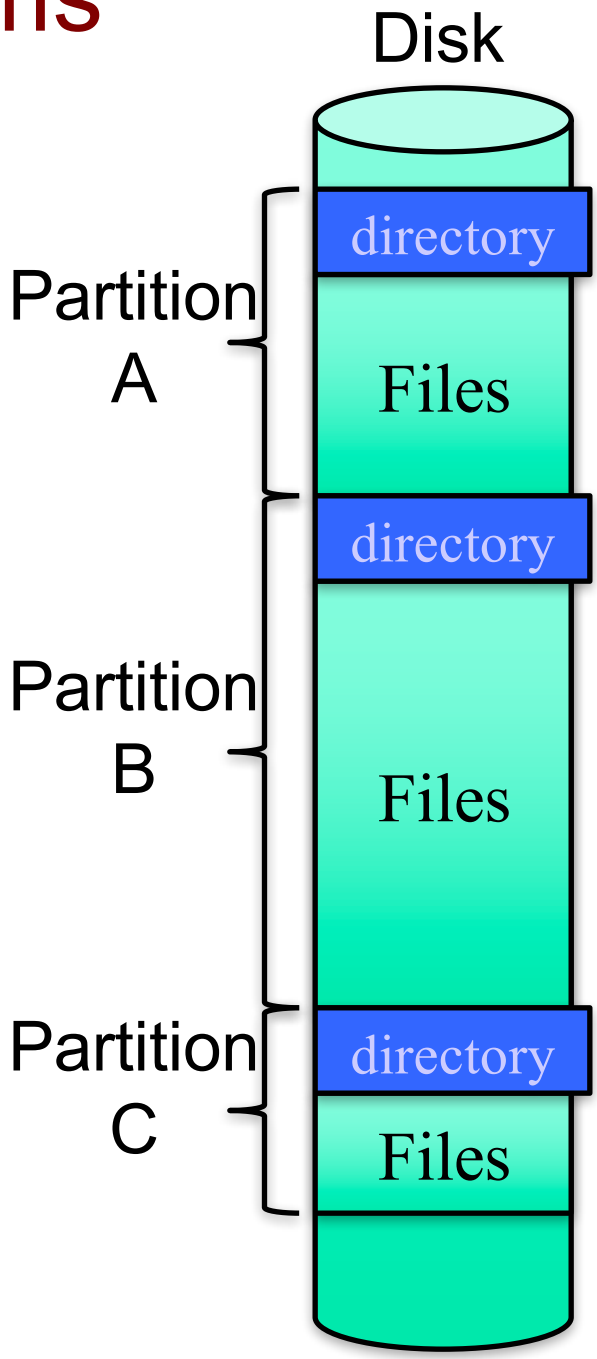


Virtual File Systems

- A *virtual file system* (VFS) layer abstracts file representation and manipulation
- 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

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…



Partitions and Volumes

- Disk partition
  - OS can partition a disk into one or more groups of cylinders
  - OS treats each partition as a separate (logical) disk
  - A partition editor can be used to create, resize, delete, and manipulate these partitions on the hard disk.
  - Different partitions may be used for different purposes
    - OS code, swap space, user files, etc.
  - Multi-boot systems

- Volume
  - A volume is a single accessible storage area with a single file system
  - One or more partitions make up a volume
  - A volume may be spread across different disk partitions on different disks, e.g. in RAID disk systems

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

2. A *system-wide open file table* (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. A *per-process OFT* - tracks all files that have been opened by a particular process
   - Access rights, a current-file-position pointer, …

4. A mount table of devices with file systems that have been mounted as volumes

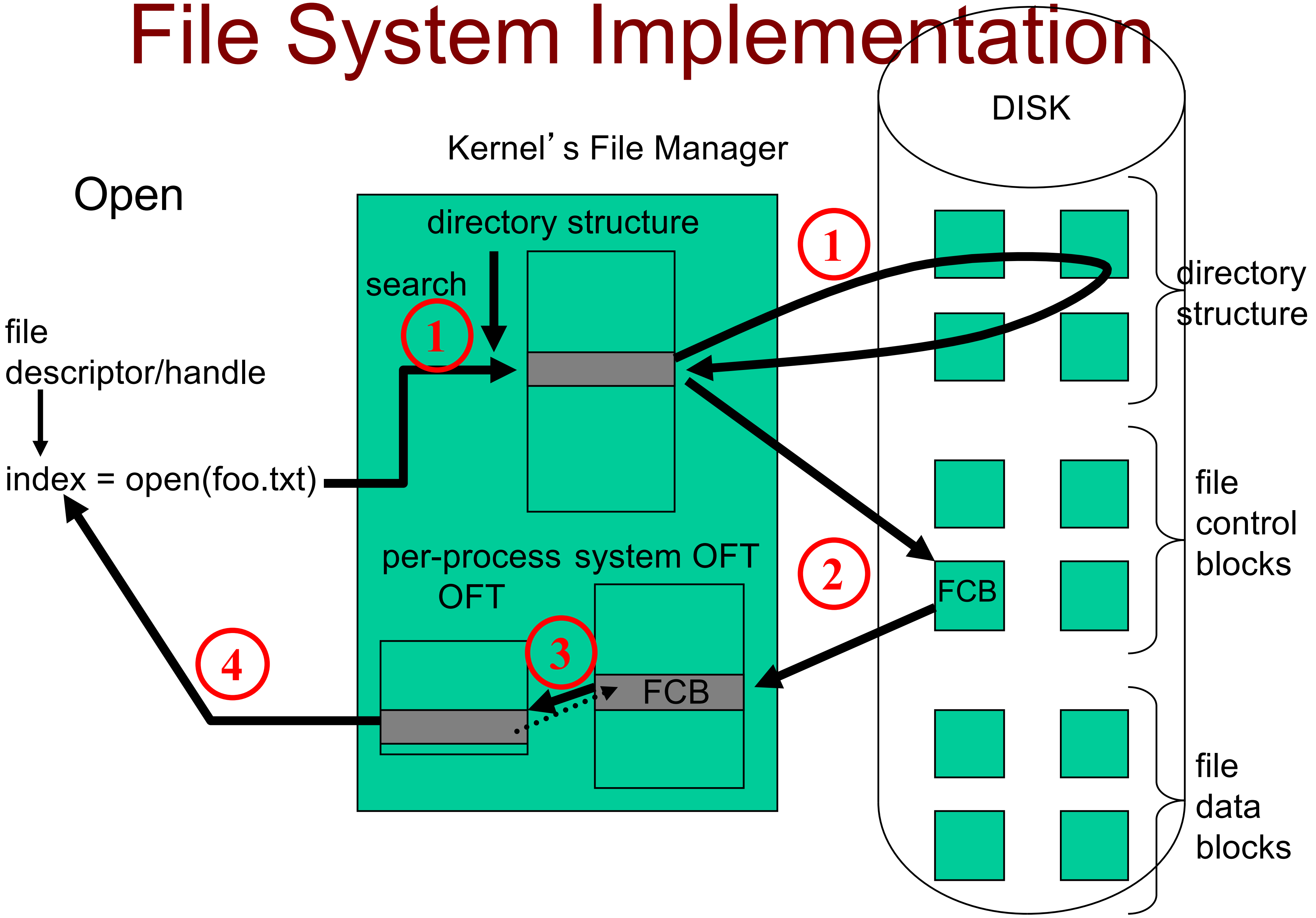
File System Implementation

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 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 Implementation



File System Implementation

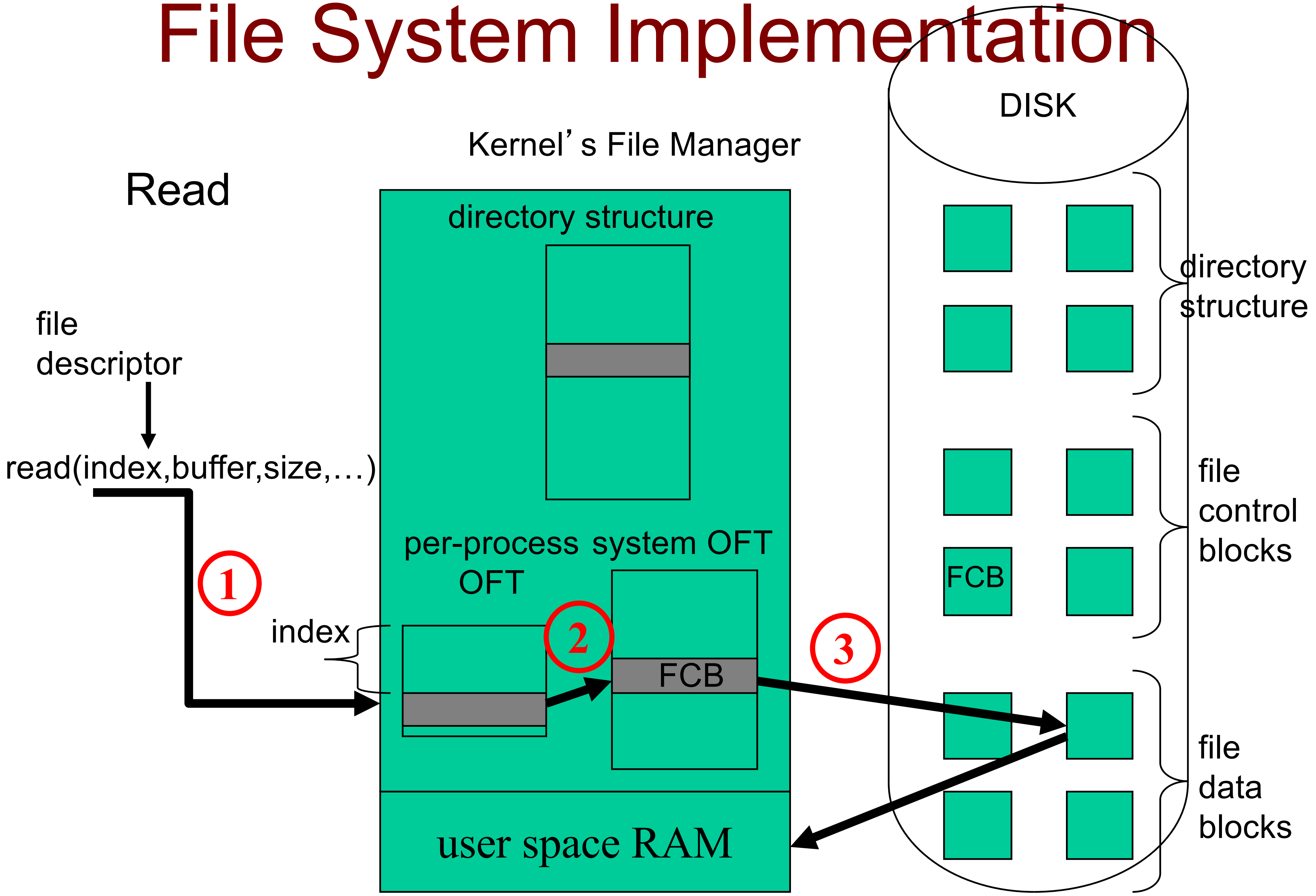
- Some OS’s employ a mandatory lock on an open file so that only one process at a time can use an open file, e.g. Windows
- Other OS’s allow optional or advisory locks, e.g. UNIX. In this case, it’s up to users to synchronize access to files.

File System Implementation

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 that 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.

File System Implementation



Directory Implementation

- Linear List
  - Searching for a file is slow because each directory requires a linear search
  - Also slow because creating and deleting a file requires a search through the linear list
  - Could keep a sorted list in memory
- Hash Table
  - Hash the file name, and in each directory search only the short linked list corresponding to that hash value
  - Greatly reduces search time
  - Linux ext3 and ext4 file systems use a variant called HTree, a hashed B-tree for fast lookup

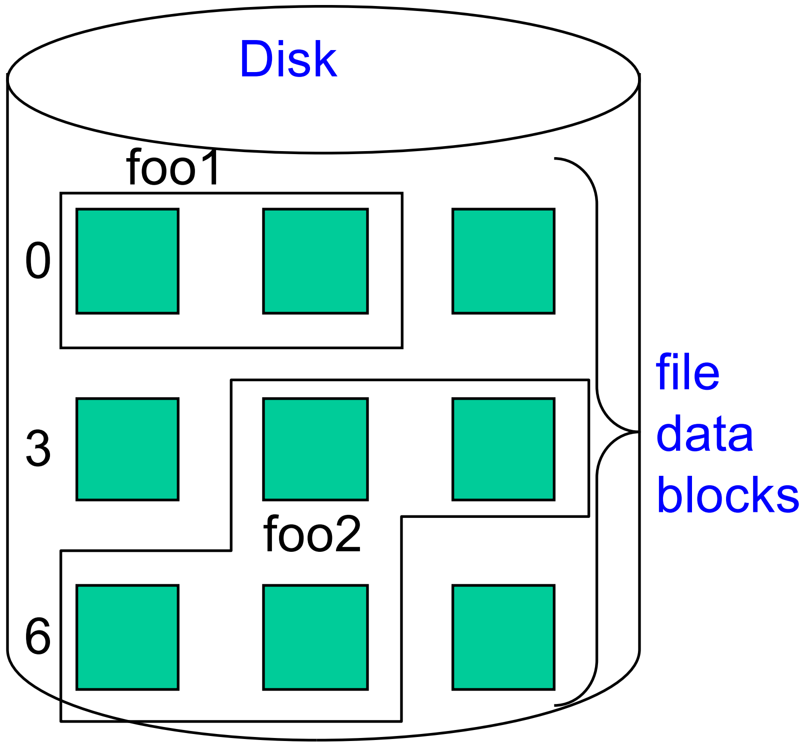
File Allocation

- File allocation concerns how file data is stored on disk
  - We divide up disk into equally sized blocks

1. Contiguous file allocation
   - A file is laid out contiguously, i.e. if a file is n blocks long, then a starting address b is selected and the file is allocated blocks b, b+1, b+2, ..., b+n-1

File headers

file	start	length
foo1	0	2
foo2	4	4



Contiguous File Allocation

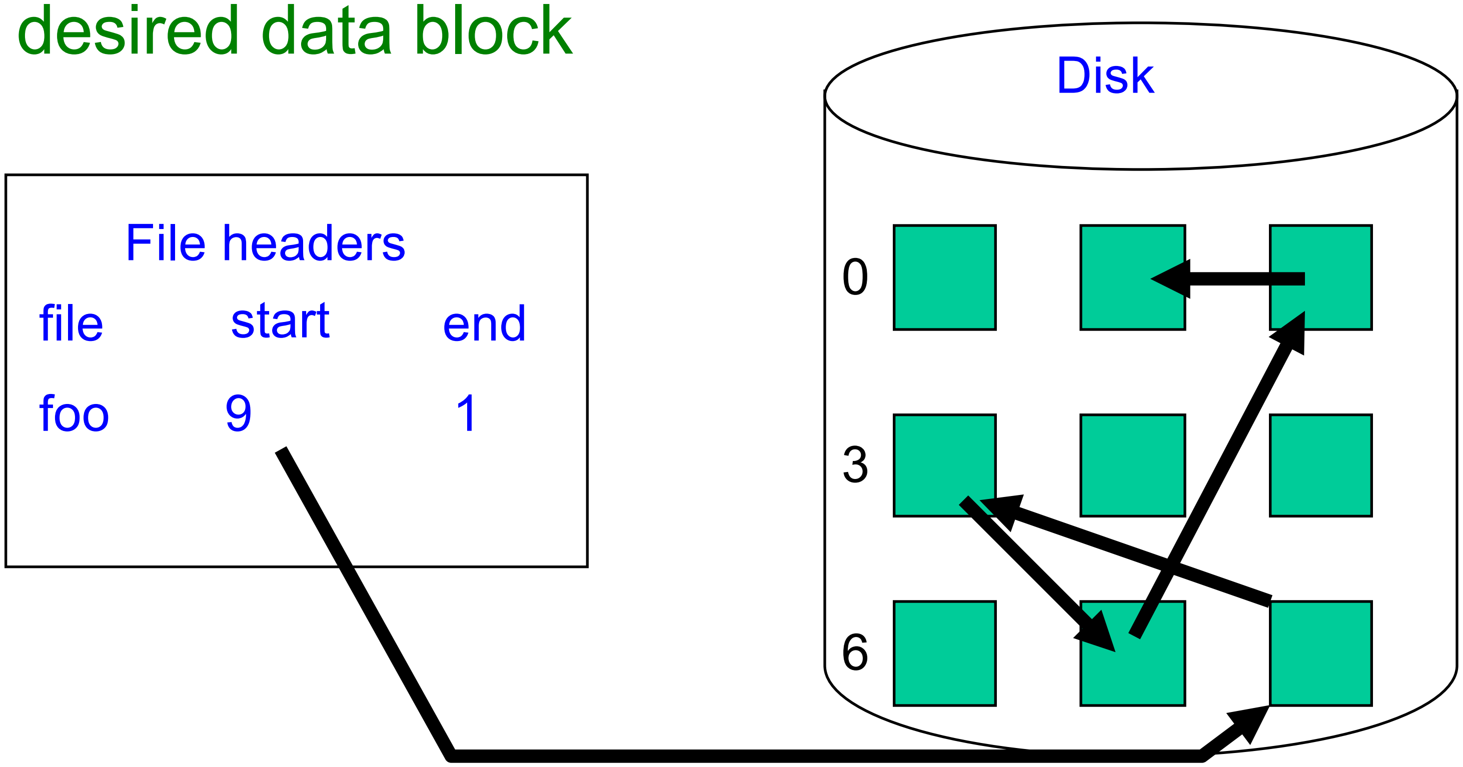
- Advantage: fast performance (low seek times because the blocks are all allocated near each other on disk)
- Problem 1: external fragmentation
  - Solutions: first fit, best fit, etc.
  - also compact memory/defragment disk
    - Can be performed in the background, late at night, etc.
- Problem 2: May not know the size of the file in advance
  - allocate a larger hole than estimated
  - if file exceeds allocation, have to copy file to a larger hole
- Problem 3: A file may eventually need 1 million bytes of space, but it may be growing at a very slow rate, e.g. 1 byte/sec, so that allocating 1 MB wastes allocation. This is a "slow growth" problem.

File Allocation

- Fixed-sized blocks have the following benefits
  - Solves *external fragmentation* problem
  - Addresses problem of not knowing in advance the size of a file – just allocate more blocks on an as-needed basis
  - Solves problem of "slow growth" – allocate blocks as needed
  - UNIX FS (UFS, = Berkeley FFS) uses 8 KB blocks. Linux' file system ext2fs uses default 1 KB blocks (though 2 and 4 KB supported (and much larger))
- Need a data structure to keep track of where file is laid out on disk
  - Must accommodate growth of the file

File Allocation

2. Linked Allocation
   - Each file is a linked list of disk blocks
   - To add to a file, just modify the linked list either in the middle or at the tail, depending on where you wish to add a block
   - To read from a file, traverse linked list until reaching the desired data block



Linked Allocation

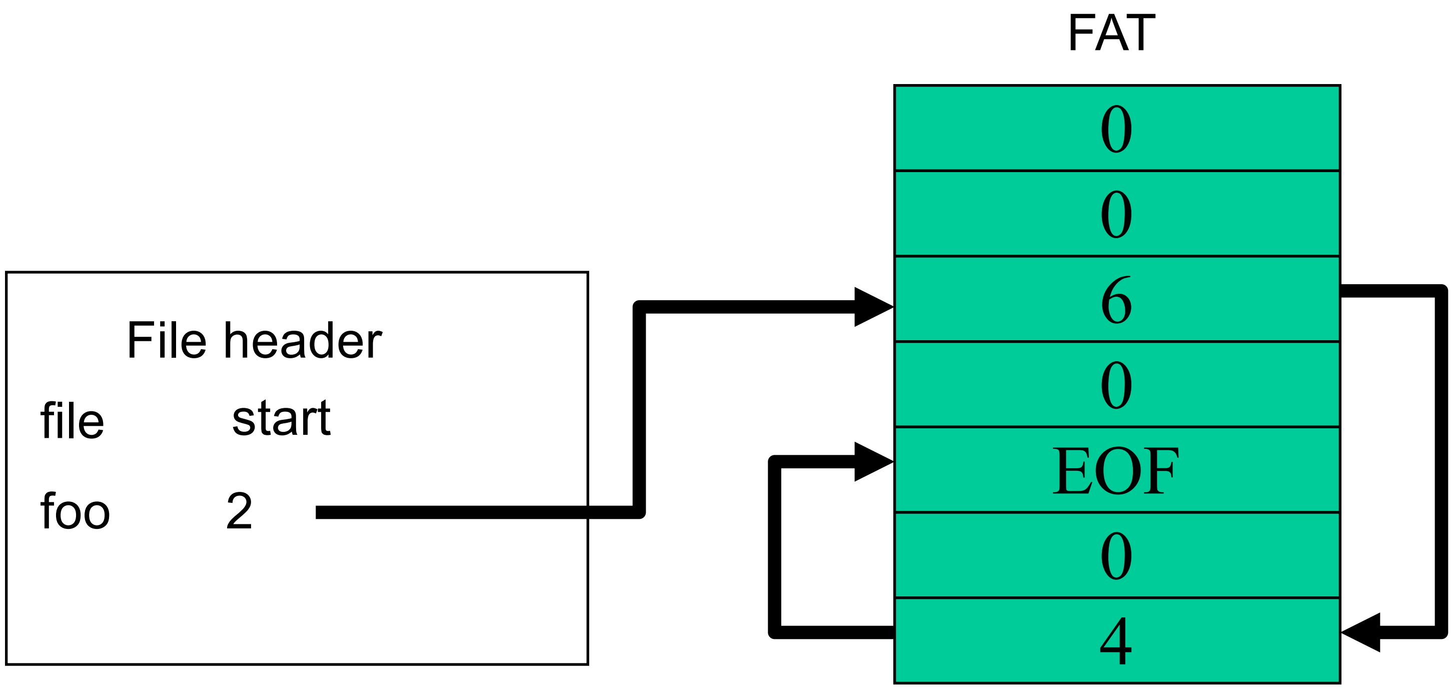
- Solves problems of contiguous allocation
  - no external fragmentation, no need to defragment
  - don’t need to know size of a file a priori
  - slow growth is not a problem
- Advantage: allocate only minimum bookkeeping overhead needed to record where a file is located on disk
  - Number of pointers in the linked list is the minimum bookkeeping info needed to recover the file (compare to indexed allocation later)
- Another advantage is support for inserting data into the middle of a file – insert blocks in the middle of a linked list
- Problems:
  - performance of random (direct) access is extremely slow for reads/writes, because you have to traverse the linked list until indexing into the correct disk block
  - reliability is fragile - if one pointer is in error or corrupted, then lose the rest of the file after that pointer

File Allocation

3. File Allocation Table (FAT) is an important variation of linked lists
   - Used in MS-DOS and Win95/98, before being replaced by NTFS (basis of Windows file systems from WinNT through Windows Vista/7)
   - Instead of embedding the pointers in the linked list with the file data blocks themselves, separate the pointers out and put them in a special table (FAT) located at a section of disk at the beginning of a volume
   - Random/direct access Read/Write times are faster than pure linked list because the pointers are all co-located in the FAT near each other at the beginning of disk volume
     - Still have to traverse the linked list though, which is a slow operation

File Allocation Table

- Entries in the FAT point to other entries in the FAT, but their values are interpreted as the disk block number
- Unused blocks in the FAT are initialized to 0
- Linked list for a file is terminated by EOF
- allocating a new block is simple - find the first 0-valued block
- FAT-16 and FAT-32 refer to the size of the address used in the FAT



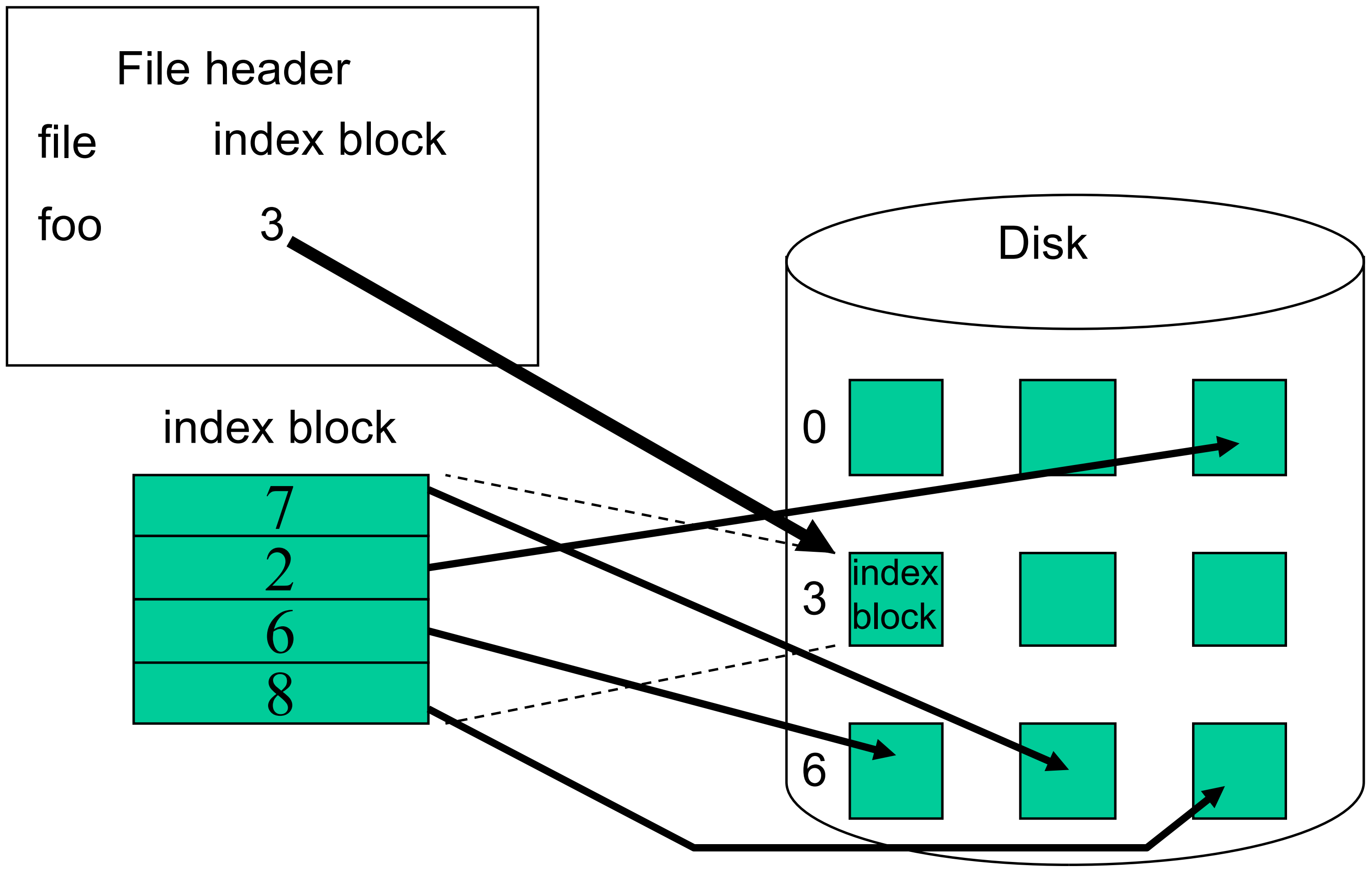
File Allocation

4. Indexed Allocation
   - Collect all pointers into a list or table called an *index block* (per file)
     - Index j into the list or index block retrieves a pointer to the j' th block on disk
   - Unlike FAT, index block can be stored in any block on disk, not just in a special section at the beginning of disk
   - unlike FAT, index is just a linear list of pointers

Indexed Allocation

- Advantages
  - No external fragmentation
  - Size of file not required a priori
  - Slow growth is efficiently supported
  - Don’t have to traverse linked list for random/direct reads/writes
    - just index quickly into the index block

Indexed Allocation



Indexed Allocation

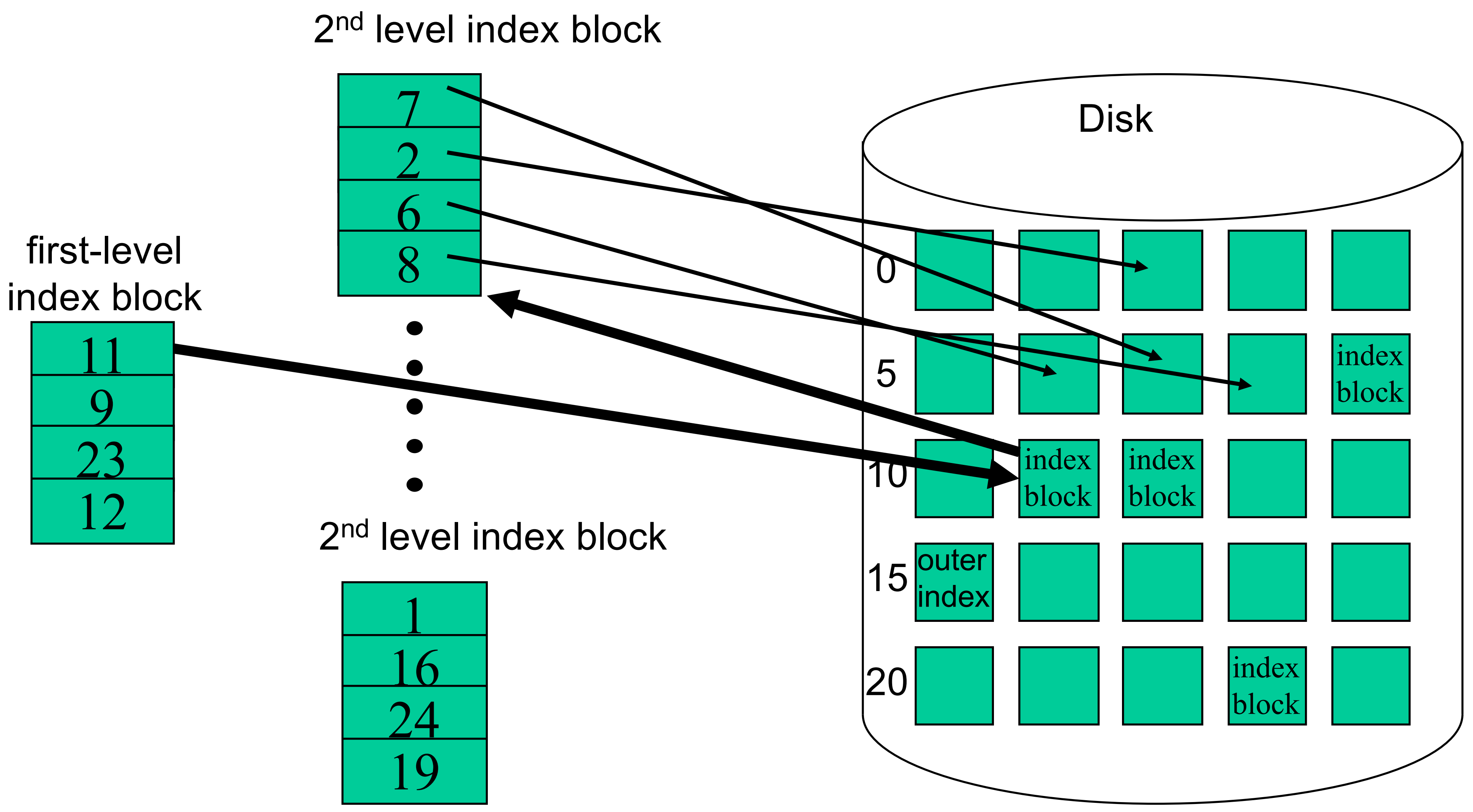
Problem: how large should the index block be?

- Too large → many wasted/empty entries for small files
- Too small → not enough entries for large files
- Solutions:
  - link together index blocks → multilevel index

File Allocation

5. multilevel index allocation
   - indexing into the first-level index block provides a pointer to second-level index blocks. Indexing into the second-level index block (using a different offset) retrieves a pointer to the actual file block on disk
   - Don't have to allocate unused second-level index blocks!
   - Example: two levels of index blocks, 4 KB blocks → 1024 pointer entries/block => 1 million addressable data blocks → the largest file size is 4 GB.

Multilevel Index Allocation



Multilevel Indexed Allocation

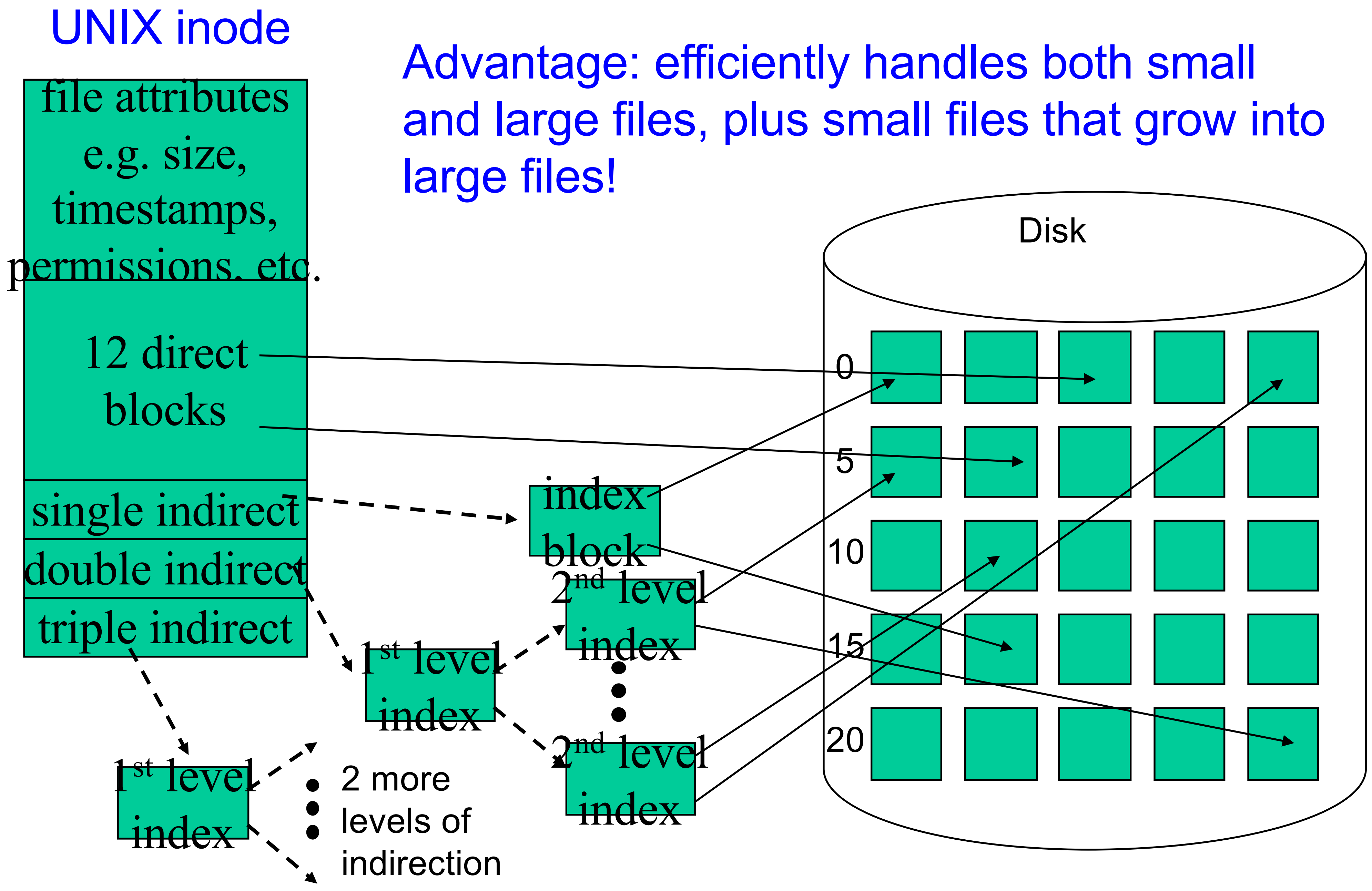
- Problem: Accessing small files takes just as long as large files, i.e. have to go through the same # of levels of indexing, hence disk operations.

File Allocation

6. UNIX (and Linux ext2fs, ext3fs, etc.) uses the following variation of multi-level indexing to accommodate large and small files
   - Suppose there are 15 entries in the index block
   - First 12 entries point to direct blocks on disk
   - 13th entry points to a 1-level indirect block
   - 14th entry points a 2-level indirect block
   - 15th entry points to a 3-level indirect block
   - For small files, this approach only uses a small index block of 15 entries, so there is very little wasted memory
   - For large files, the indirect pointers allow expansion of the index block to span a large number of disk blocks
   - UNIX stores this hybrid index block with the file inode

Unix File Allocation

Advantage: efficiently handles both small and large files, plus small files that grow into large files!



Free Space Management

Another aspect of managing a file system is managing free space

•File system needs to keep track of what blocks of disk are free/unallocated

•keep a free-space "list"

Approaches:

1. Bit Vector or Bit Map
2. Linked List
3. Grouping
4. Counting

Free Space Management

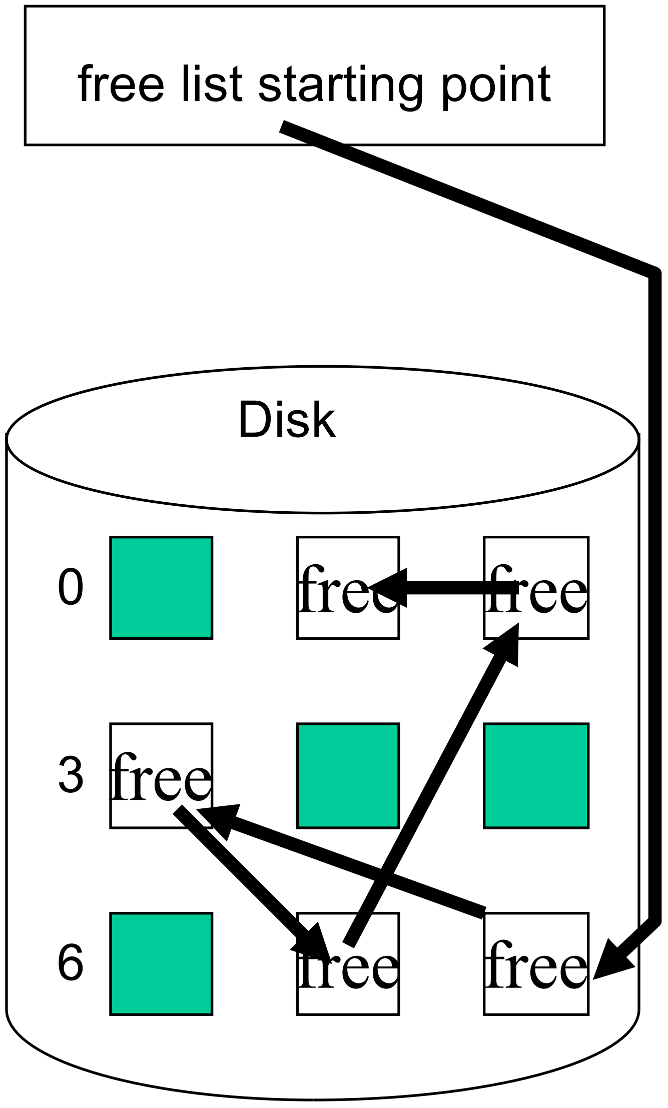
1. Bit Vector or Bitmap

- Each block is represented by a bit. Concatenate all such bits into an array of bits, namely a bit vector. The j'th bit indicates whether the j'th block has been allocated.
- if bit = 1, then a block is free, else if bit = 0, then block is allocated
- simple, easy to implement and search
  - to search for a free block, find the 1st non-zero word in bitmap
- Linux ext2fs uses a version of this bit map approach
- but its size can become quite large
  - Might take awhile to search and find first free block (bit)

Free Space Management

2. Linked List

- Link together all free blocks
- Keeps track of only the free blocks
  - Bitmap has the overhead of tracking both free and allocated blocks - this is wasteful if memory is mostly allocated
  - Faster than bitmap – find 1st free block immediately
- Problem: traversing the free list is slow if you want to allocate a large number of free blocks all at once
  - hopefully this occurs infrequently



Free Space Management

3. Grouping

- Linked list, except store n-1 pointers to free blocks in each list block, while the last block points to the next list block containing more free pointers
- Allows faster allocation of larger numbers of free blocks all at once

4. Counting

- Grouped linked list, except add a field to each pointer entry that indicates the number of free blocks immediately after the block pointed to
- Even faster allocation of large # of free blocks

File System Performance

So far, we've seen the following approaches to improve performance in a file system:

- In memory:
  - Caching FCB information about open files in memory improves performance (faster access)
  - Caching directory entries in memory improves access speed.
  - And hash the directory tree to quickly find an entry and see if it' s in memory.
- On disk:
  - file data: indexed allocation is generally faster than traversing linked list allocation
  - free block list: counting, grouped, linked list allows fast allocation of large # of files

File System Performance

Some other potential optimizations

- The disk controller can also have its own cache that stores file data/FCBs/etc. for fast access
- *read ahead*: if the OS knows this is sequential access, then read the requested block and several subsequent blocks into main memory cache in anticipation of future reads

File System Performance

- *Asynchronous writes*: delay writing of file data
  - Removes disk I/O wait time from the critical path of execution
  - This allows a disk to schedule writes efficiently, grouping nearby writes together
  - May avoid a disk write if the data has been changed again soon
  - Note that in certain cases, you may prefer to enforce *synchronous writes*, e.g. when modifying file metadata in the FCB on an open() call
- Cache file data in memory
- Smarter layout on disk: keep an inode/FCB near file data to reduce disk seeks, and/or file data blocks near each other

Reliability/Fault Recovery

- In general, if there is a hardware or software failure that causes the OS to crash, the system should be able to recover gracefully
  - The file system needs to be engineered to ensure reliability/fault recovery
- Example: asynchronous writes introduce the problem of maintaining consistency between the file system on disk and the writes cached in RAM
  - Directory info in RAM can be more up to date than disk
  - If there is a system failure, e.g. power loss, the cached writes may be lost
  - in this case, the promised writes will not be executed
  - UNIX caches directory entries for reads, but any data write causing changes in metadata or free space allocation is written synchronously (immediately) to disk, before the data blocks are written

Reliability/Fault Recovery

- Example: even if all writes are synchronous, there is still a consistency problem - e.g. a file create() involves many operations on the file system, and may be interrupted at any time in mid-execution
  - file create() updates the directory, FCB, file data blocks, and free space management
  - if there is a failure after creating the FCB, then the file system is in an inconsistent state because the file data has not yet been saved on disk, i.e. the directory says there is a file and points to the FCB, but the FCB is incomplete because its index block hasn’t been fully allocated

Reliability/Fault Recovery

- Approach: file systems often run a *consistency checker* like fsck in UNIX or chkdsk in MSDOS
  - In linked allocation, would check each linked list and all FCB's to see if they are consistent with the directory structure. similar checks for indexed allocation
  - Check each allocated file data block to see that its checksum is valid
  - Disadvantages:
    - This is heavyweight, and takes a long time to check the entire file system.
    - This can help detect an error, but doesn't necessarily help you correct or recover from the error.

Reliability/Fault Recovery

- Want a solution that helps you recover from file system failures: *log-based recovery*
  - OS maintains a *log* or journal on disk of each operation on the file system
    - called *log-based or journaling* file systems,
    - The log on disk is consulted after a failure to reconstruct the file system

Log-Based Recovery

- Each operation on the file system is written as a record to the log on disk *before* the operation is actually performed on data on disk
  - This is called *write-ahead logging*
  - Thus, the file system has a sequence of records of operations in the log about what was intended in case of a crash
  - The log contains a sequence of statements like "I' m about to write this directory entry/file header/file data block", and "I just finished writing this directory/FH/data".
- Some file systems only write changes to the metadata of a filesystem to the log, e.g. file headers and directory entries only (NTFS), and not any changes to file data

Log-Based Recovery

Linux ext3fs can be parameterized to operate in:

•Journal mode: both metadata and file data are logged. This is the safest mode, but there is the latency cost of two disk writes for every write.

•Ordered mode: only metadata is logged, not file data, and it's guaranteed that file contents are written to disk before associated metadata is marked as committed in the journal. This way, you don't have say a file header pointing to a new block of data, yet that data has not yet been written to disk. This is the default on many Linux distributions.

•Writeback mode: only metadata is logged, not file data, and no guarantee file data written before metadata, so files can become corrupted. This is riskiest mode/least reliable.