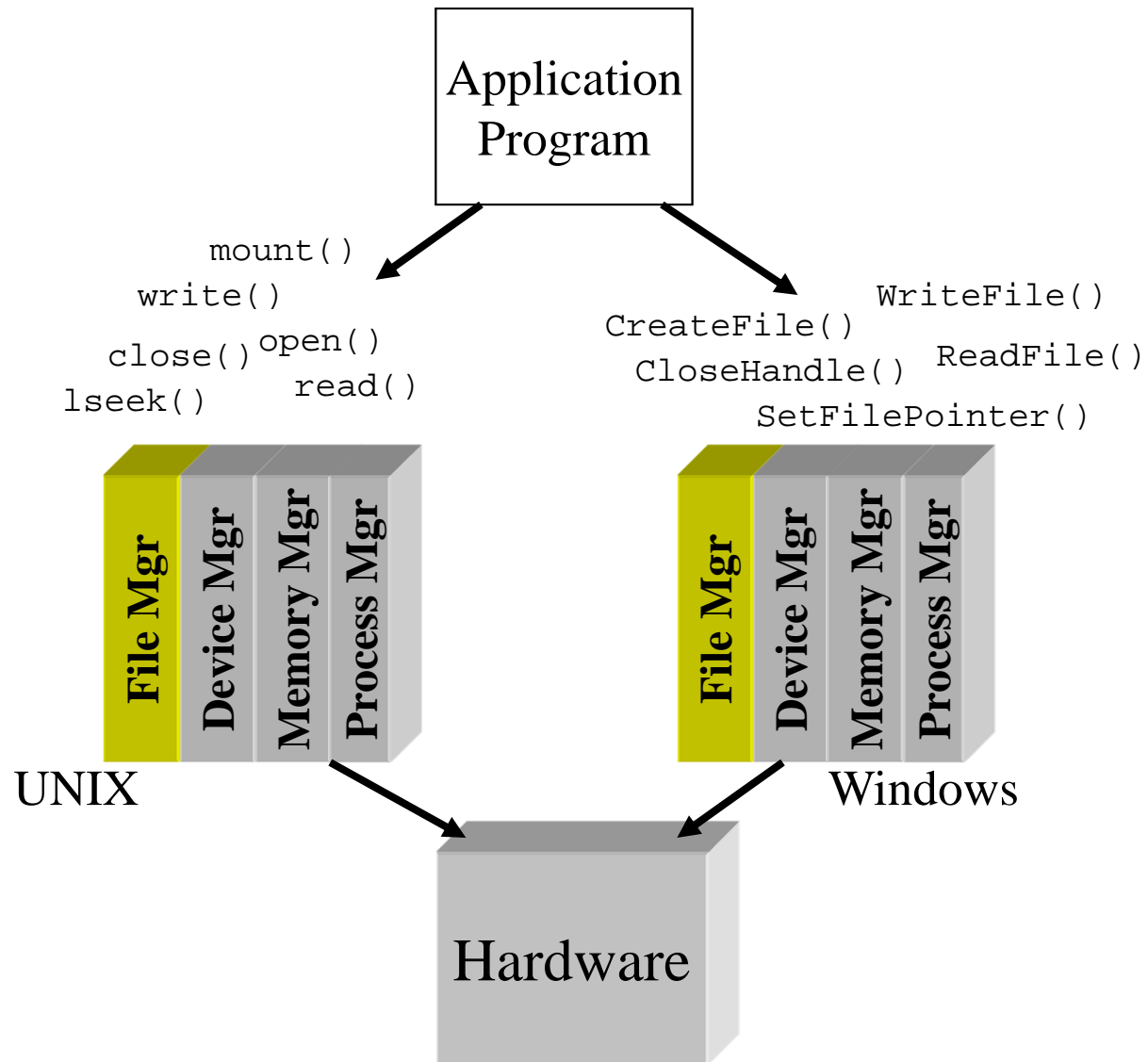


File System

Raju Pandey
Department of Computer Sciences
University of California, Davis
Spring 2011

Overall view

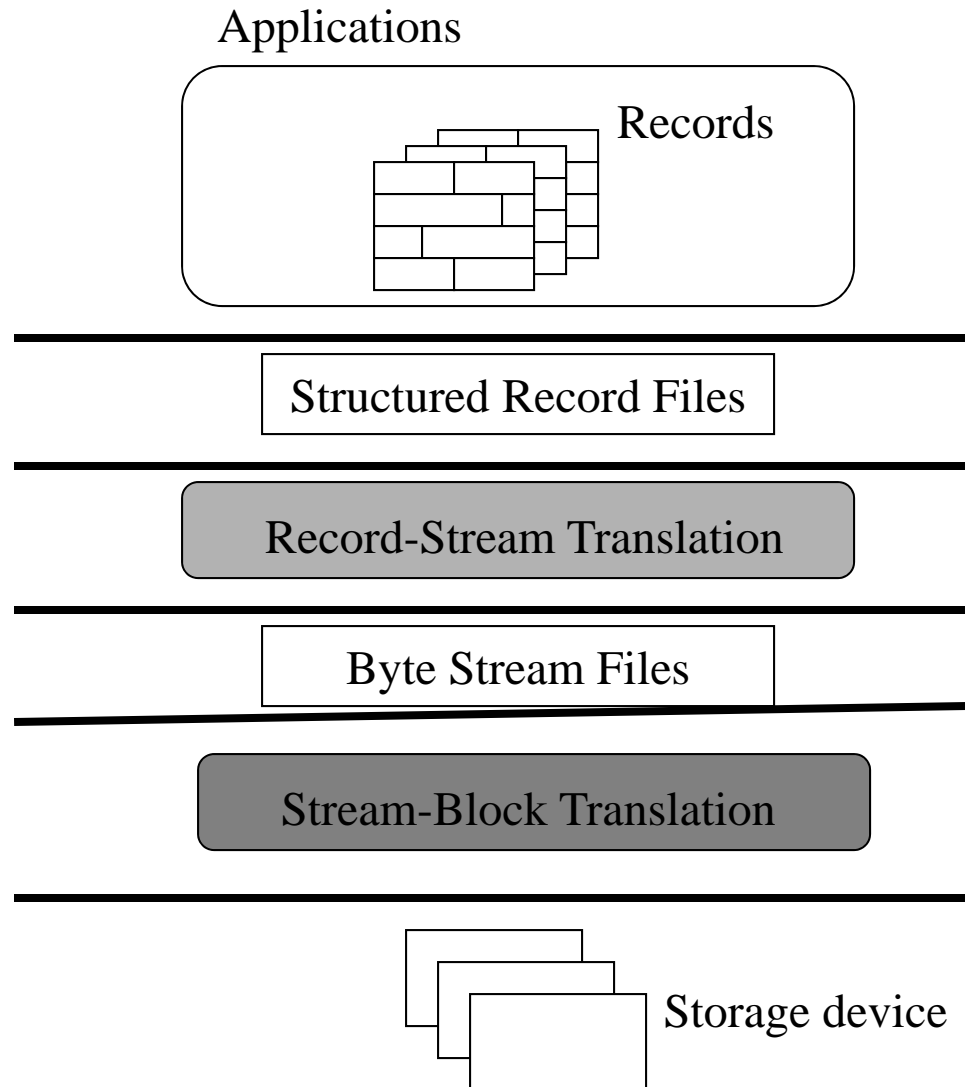


Files

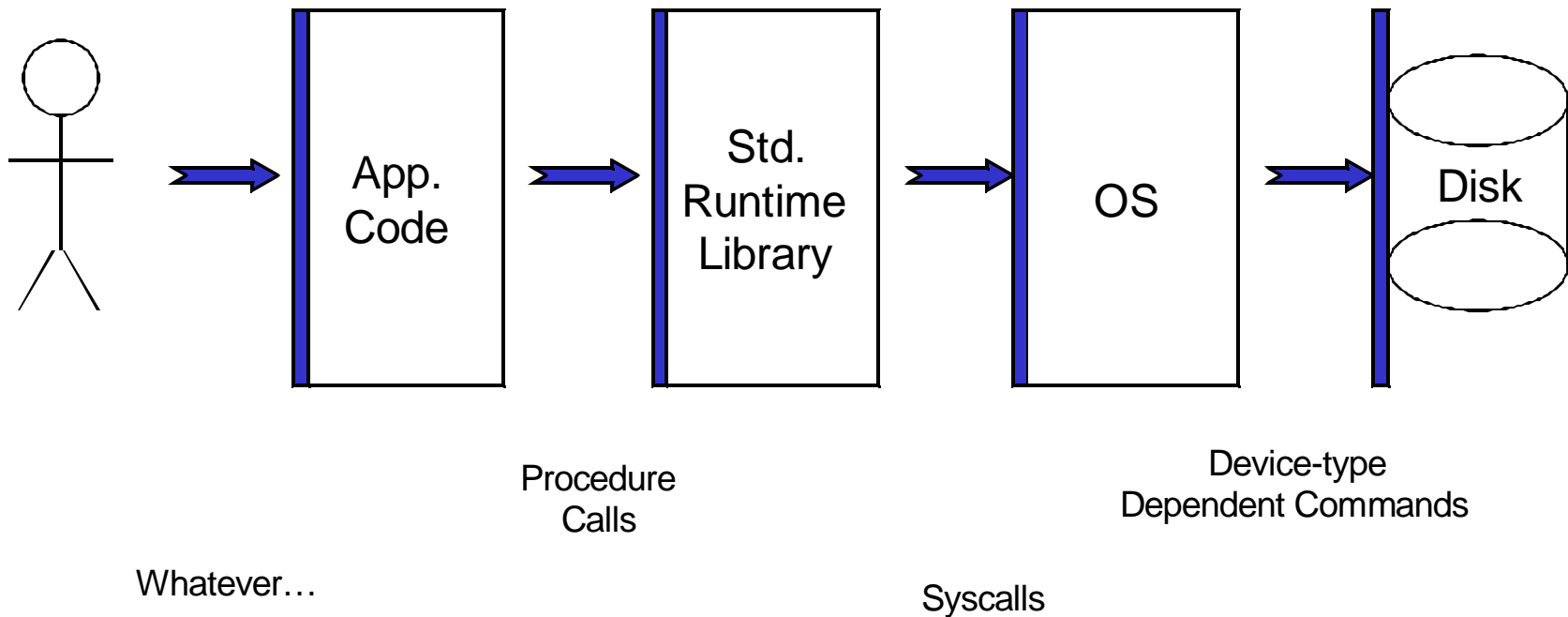
- A file is a collection of data with some properties
 - contents, size, owner, last read/write time, protection ...
- Files may also have types
 - understood by file system
 - o device, directory, symbolic link
 - understood by other parts of OS or by runtime libraries
 - o executable, dll, source code, object code, text file, ...
- Type can be encoded in the file's name or contents
 - windows encodes type in name
 - o .com, .exe, .bat, .dll, .jpg, .mov, .mp3, ...
 - old Mac OS stored the name of the creating program along with the file
 - unix has a smattering of both
 - o in content via magic numbers or initial characters (e.g., #!)

File Management

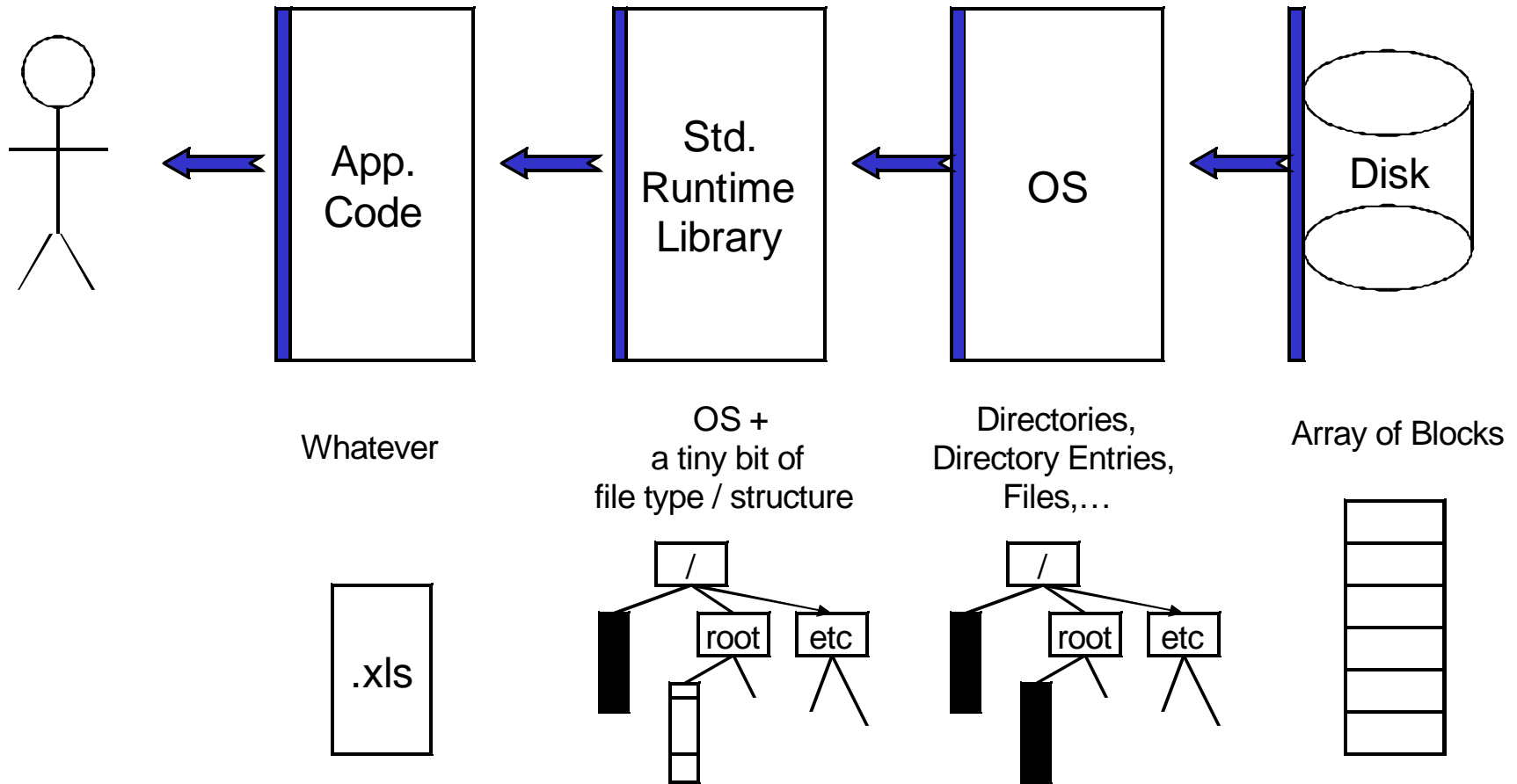
- The file manager administers the collection by:
 - Storing the information on a device
 - Mapping the block storage to a logical view
 - Allocating/deallocating storage
 - Providing file directories
- What abstraction should be presented to programmer?



Interface Layers

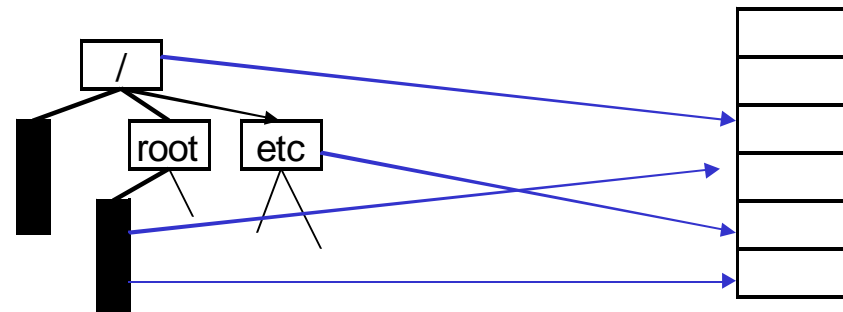
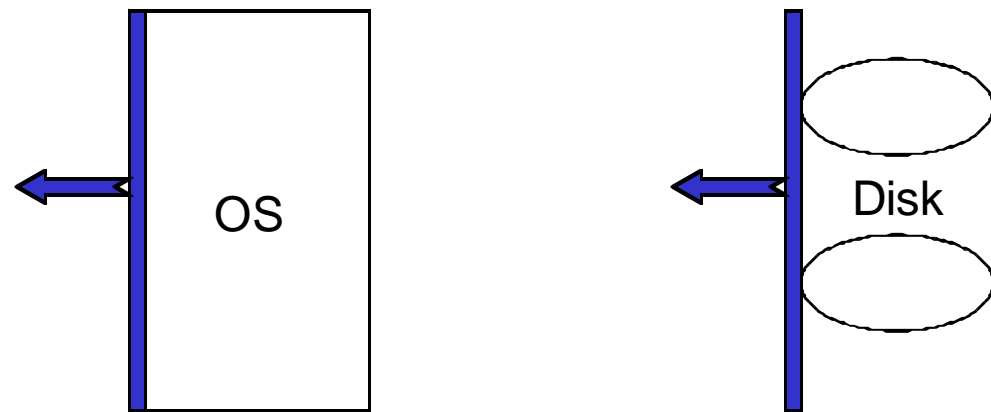


Exported Abstractions



Primary Roles of the OS (file system)

1. Hide hardware specific interface
2. Allocate disk blocks
3. Check permissions
4. Understand directory file structure
5. Maintain *metadata*
6. Performance
7. Flexibility



File access methods

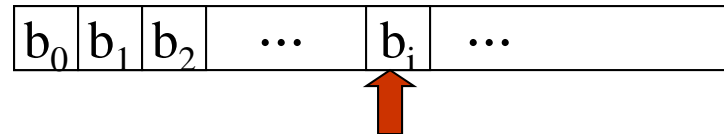
- Some file systems provide different **access methods** that specify ways the application will access data
 - sequential access
 - o read bytes one at a time, in order
 - direct access
 - o random access given a block/byte #
 - record access
 - o file is array of fixed- or variable-sized records
 - indexed access
 - o FS contains an index to a particular field of each record in a file
 - o apps can find a file based on value in that record (similar to DB)
- Why do we care about distinguishing sequential from direct access?
 - what might the FS do differently in these cases?

Byte Stream File Interface

```
fileID = open(fileName)
close(fileID)
read(fileID, buffer, length)
write(fileID, buffer, length)
seek(fileID, filePosition)
```

Low Level Files

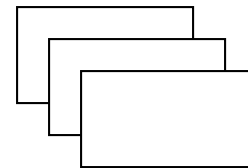
```
fid = open("fileName", ...);  
...  
read(fid, buf, buflen);  
...  
close(fid);
```



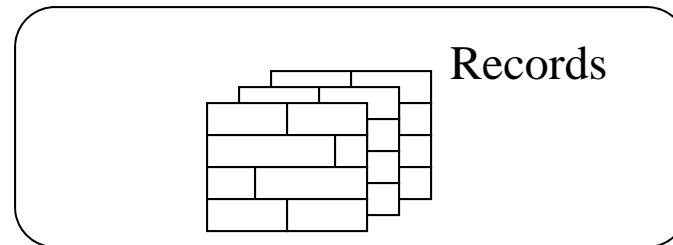
```
int open(...) {...}  
int close(...) {...}  
int read(...) {...}  
int write(...) {...}  
int seek(...) {...}
```

Stream-Block Translation

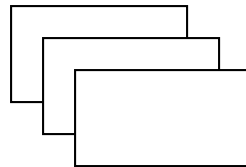
Storage device response to commands



Structured Files

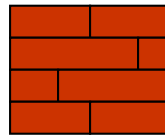


Record-Block Translation



Record-Oriented Sequential Files

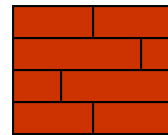
Example: Mail system



Logical Record

```
fileID = open(fileName)
close(fileID)
getRecord(fileID, record)
putRecord(fileID, record)
seek(fileID, position)
```

Record-Oriented Sequential Files



Logical Record



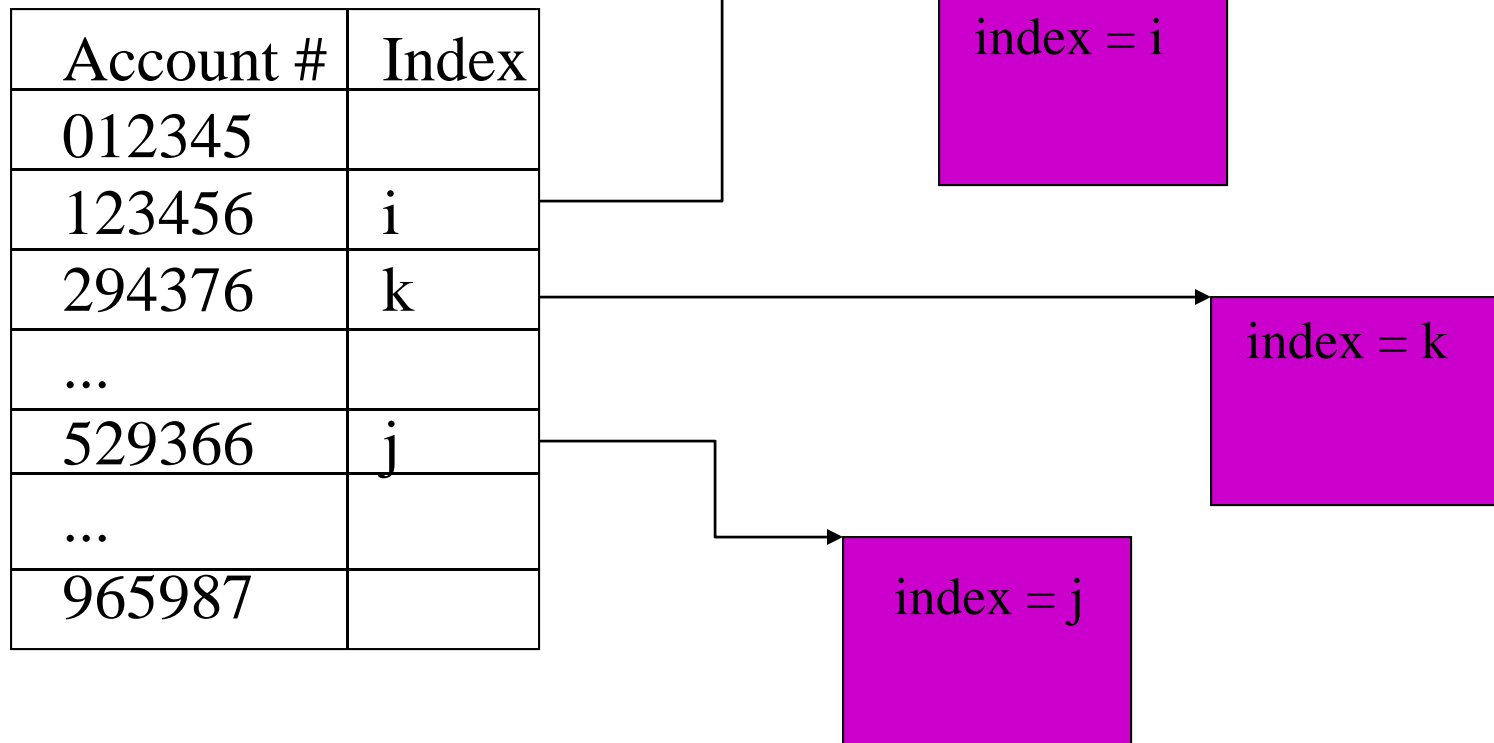
Indexed Sequential File

- Suppose we want to directly access records, rather than sequentially
- Add an index to the file
- Each record includes an index field

```
fileID = open(fileName)
close(fileID)
getRecord(fileID, index)
index = putRecord(fileID, record)
deleteRecord(fileID, index)
```

Indexed Sequential File (cont)

Application structure

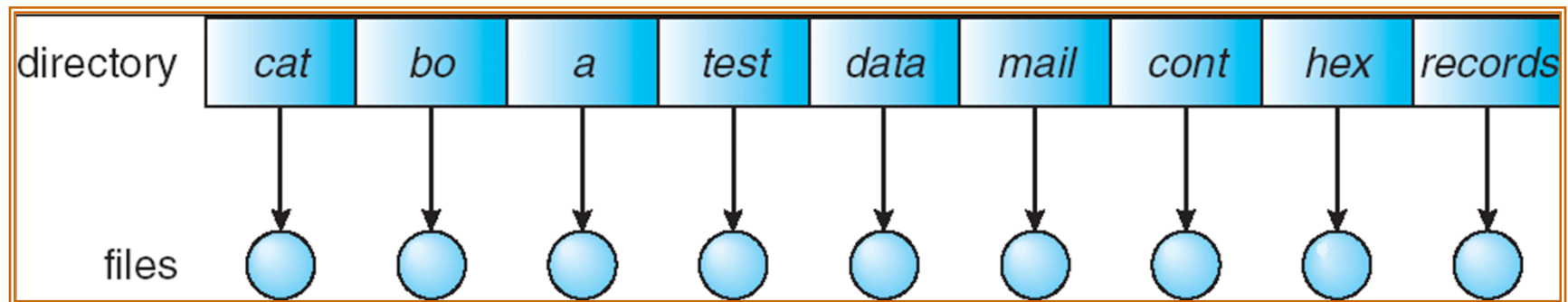


Directories

- Directories provide:
 - a way for users to organize their files
 - a convenient file name space for both users and FS's
- Most file systems support multi-level directories
 - naming hierarchies (`/`, `/usr`, `/usr/local`, `/usr/local/bin`, ...)
- Most file systems support the notion of current directory
 - absolute names: fully-qualified starting from root of FS
`bash$ cd /usr/local`
 - relative names: specified with respect to current directory
`bash$ cd /usr/local` (absolute)
`bash$ cd bin` (relative, equivalent to `cd /usr/local/bin`)

Single-Level Directory

- A single directory for all users

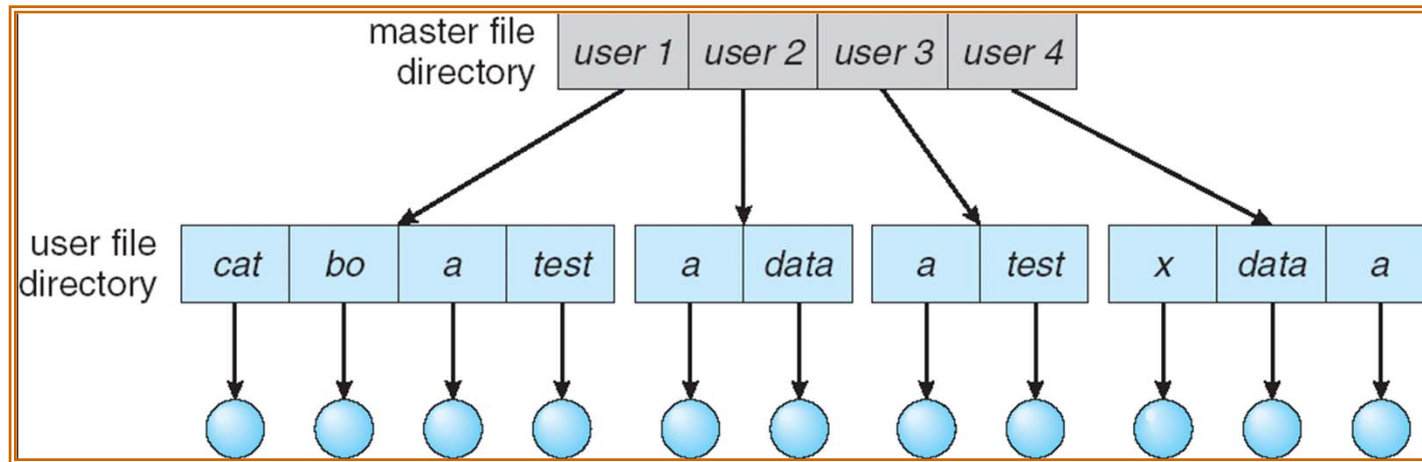


Naming problem

Grouping problem

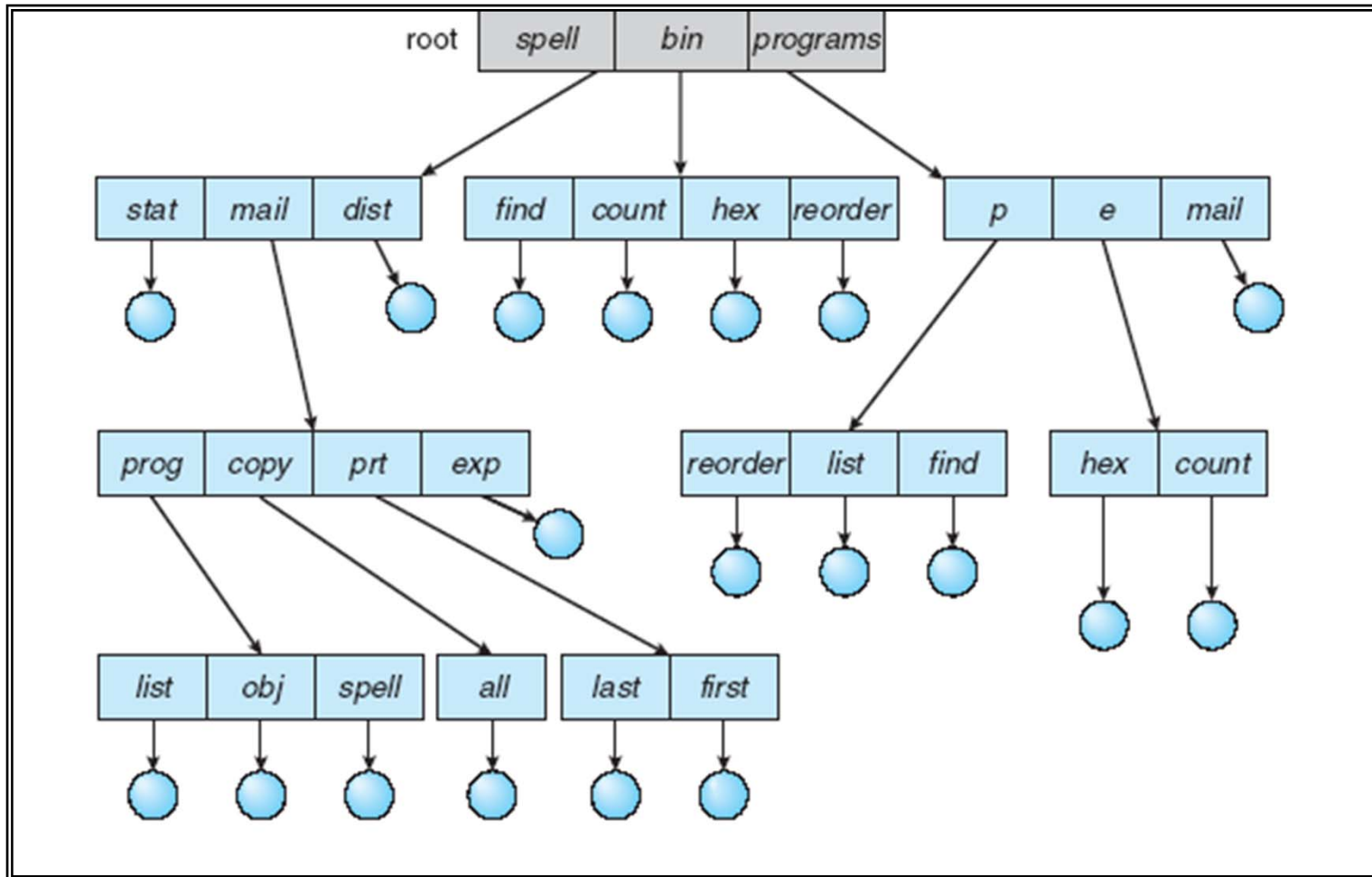
Two-Level Directory

- Separate directory for each user



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories



Tree-Structured Directories (Cont)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
 - `cd /spell/mail/prog`
 - `type list`

Tree-Structured Directories (Cont)

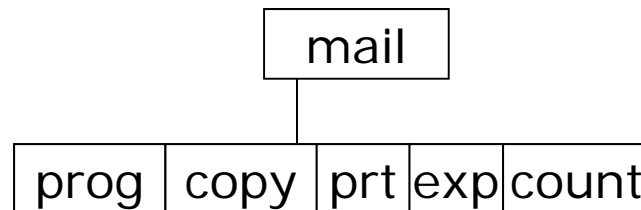
- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file

`rm <file-name>`

- Creating a new subdirectory is done in current directory

`mkdir <dir-name>`

Example: if in current directory `/mail`
`mkdir count`



Deleting "mail" \Rightarrow deleting the entire subtree rooted by "mail"

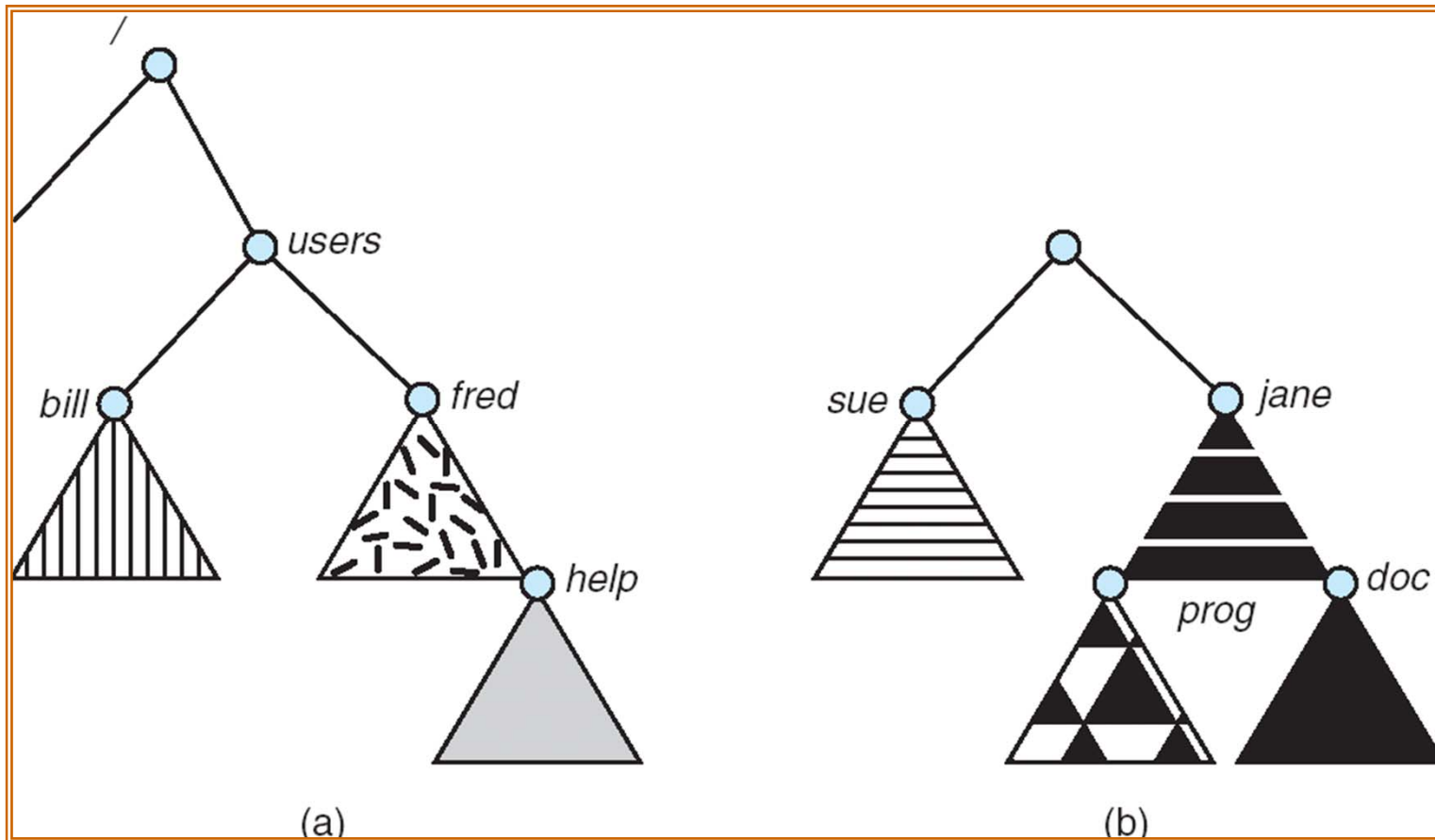
Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If *dict* deletes *list* \Rightarrow dangling pointer
Solutions:
 - Backpointers, so we can delete all pointers
Variable size records a problem
 - Backpointers using a daisy chain organization
 - Entry-hold-count solution
- New directory entry type
 - **Link** – another name (pointer) to an existing file
 - **Resolve the link** – follow pointer to locate the file

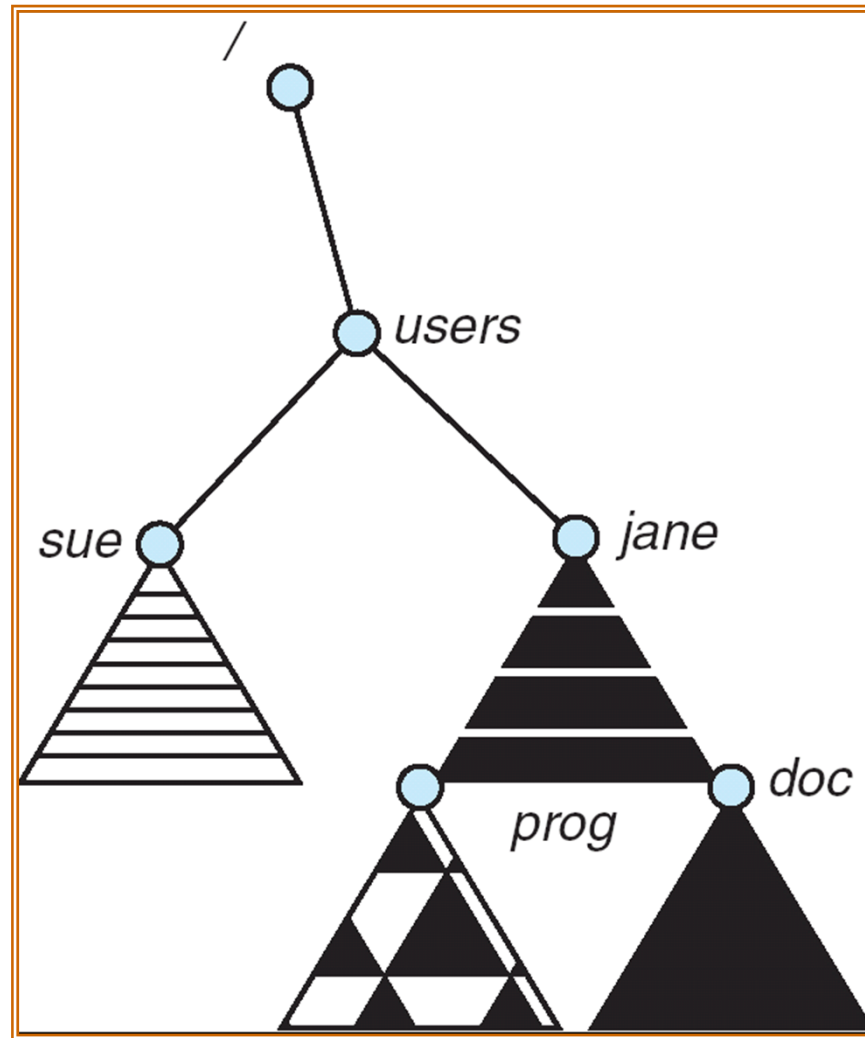
File System Mounting

- A file system must be **mounted** before it can be accessed
- A file system is mounted at a **mount point** (a directory)

(a) Existing. (b) Unmounted Partition



Mount Point



Directory internals

- A directory is typically just a file that happens to contain special metadata
 - directory = list of (name of file, file attributes)
 - attributes include such things as:
 - size, protection, location on disk, creation time, access time, ...
 - the directory list is usually unordered (effectively random)
 - when you type "ls", the "ls" command sorts the results for you

Path name translation

- Let's say you want to open `"/one/two/three"`
`fd = open("/one/two/three", O_RDWR);`
- What goes on inside the file system?
 - open directory `"/"` (well known, can always find)
 - search the directory for `"one"`, get location of `"one"`
 - open directory `"one"`, search for `"two"`, get location of `"two"`
 - open directory `"two"`, search for `"three"`, get loc. of `"three"`
 - open file `"three"`
 - (of course, permissions are checked at each step)
- FS spends lots of time walking down directory paths
 - this is why open is separate from read/write (session state)
 - OS will cache prefix lookups to enhance performance
 - o `/a/b`, `/a/bb`, `/a/bbb` all share the `"/a"` prefix

File protection

- FS must implement some kind of protection system
 - to control who can access a file (user)
 - to control how they can access it (e.g., read, write, or exec)
- More generally:
 - generalize files to **objects** (the “what”)
 - generalize users to **principals** (the “who”, user or program)
 - generalize read/write to **actions** (the “how”, or operations)
- A protection system dictates whether a given action performed by a given principal on a given object should be allowed
 - e.g., you can read or write your files, but others cannot
 - e.g., you can read `/etc/motd` but you cannot write to it

Model for representing protection

- Two different ways of thinking about it:
 - access control lists (ACLs)
 - for each object, keep list of principals and principals' allowed actions
 - capabilities
 - for each principal, keep list of objects and principal's allowed actions
- Both can be represented with the following matrix:

	objects		
	/etc/passwd	/home/gribble	/home/guest
principals	rw	rw	rw
gribble	r	rw	r
guest			r

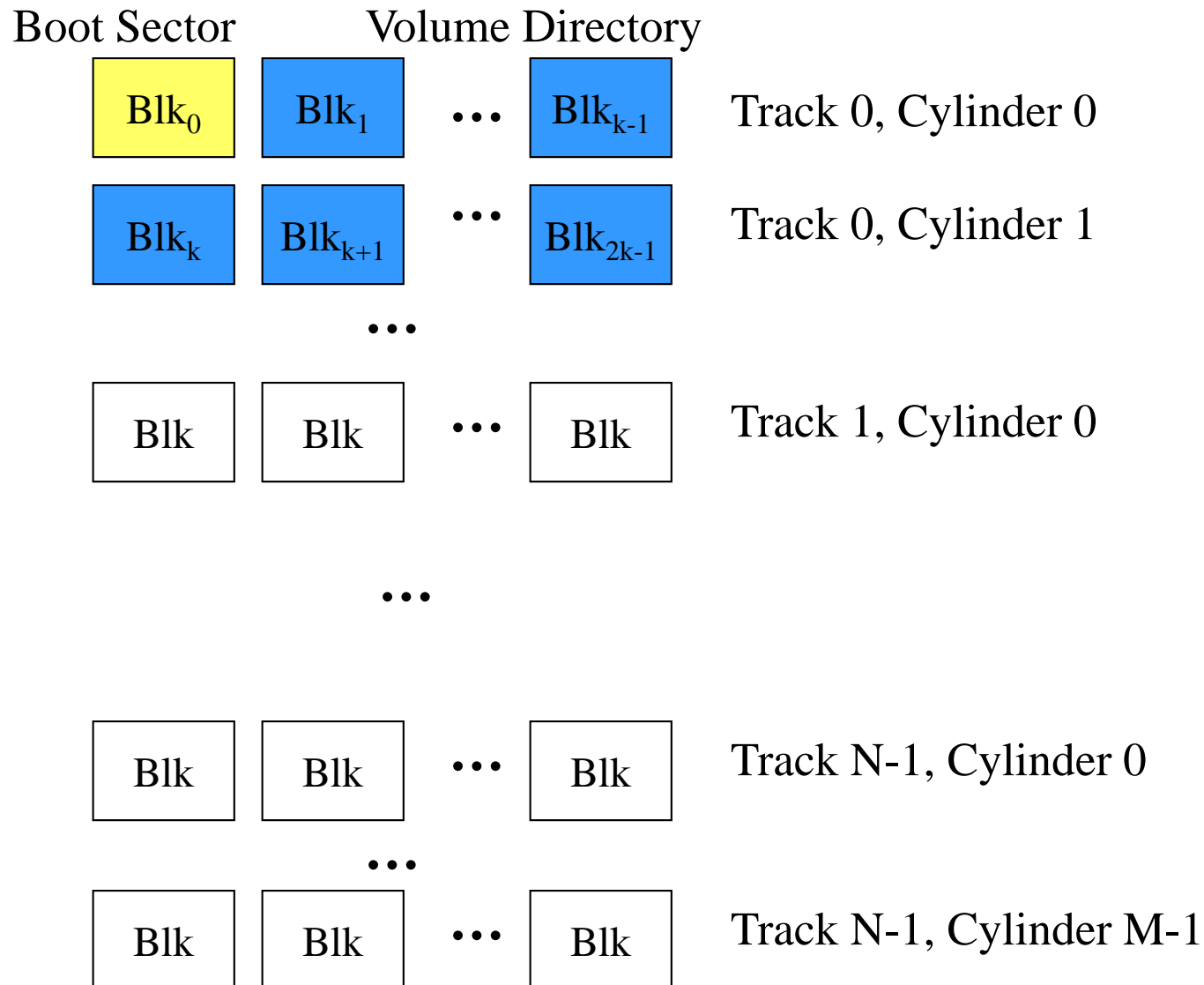
ACLs vs. Capabilities

- Capabilities are easy to transfer
 - they are like keys: can hand them off
 - they make sharing easy
- ACLs are easier to manage
 - object-centric, easy to grant and revoke
 - to revoke capability, need to keep track of principals that have it
 - hard to do, given that principals can hand off capabilities
- ACLs grow large when object is heavily shared
 - can simplify by using “groups”
 - put users in groups, put groups in ACLs
 - you are all in the “VMware powerusers” group on Win2K
 - additional benefit
 - change group membership, affects ALL objects that have this group in its ACL

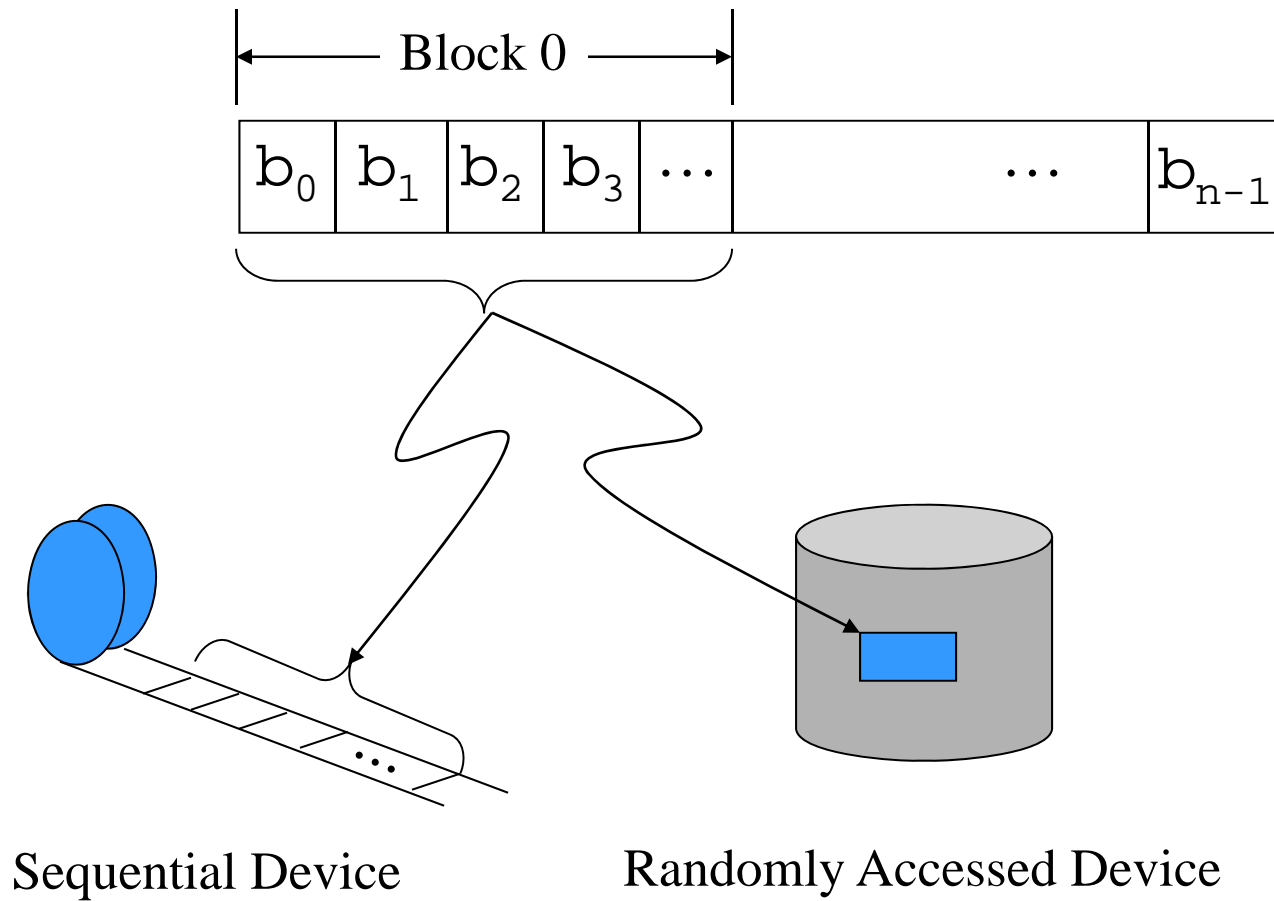
Implementing Low Level Files

- Secondary storage device contains:
 - Volume directory (sometimes a root directory for a file system)
 - External file descriptor for each file
 - The file contents
- Manages blocks
 - Assigns blocks to files (descriptor keeps track)
 - Keeps track of available blocks
- Maps to/from byte stream

Disk Organization



Low-level File System Architecture



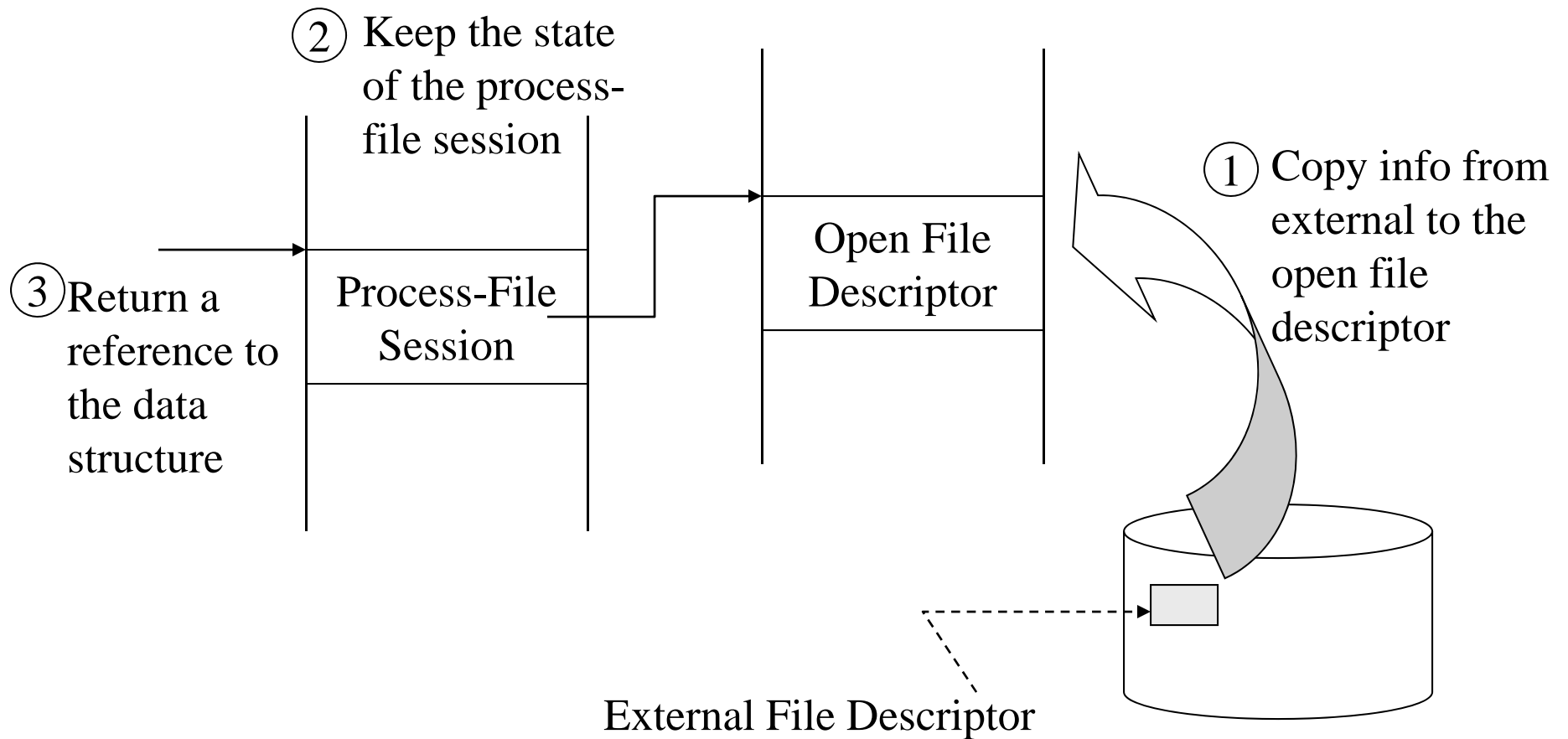
File Descriptors

- External name
- Current state
- Sharable
- Owner
- User
- Locks
- Protection settings
- Length
- Time of creation
- Time of last modification
- Time of last access
- Reference count
- Storage device details

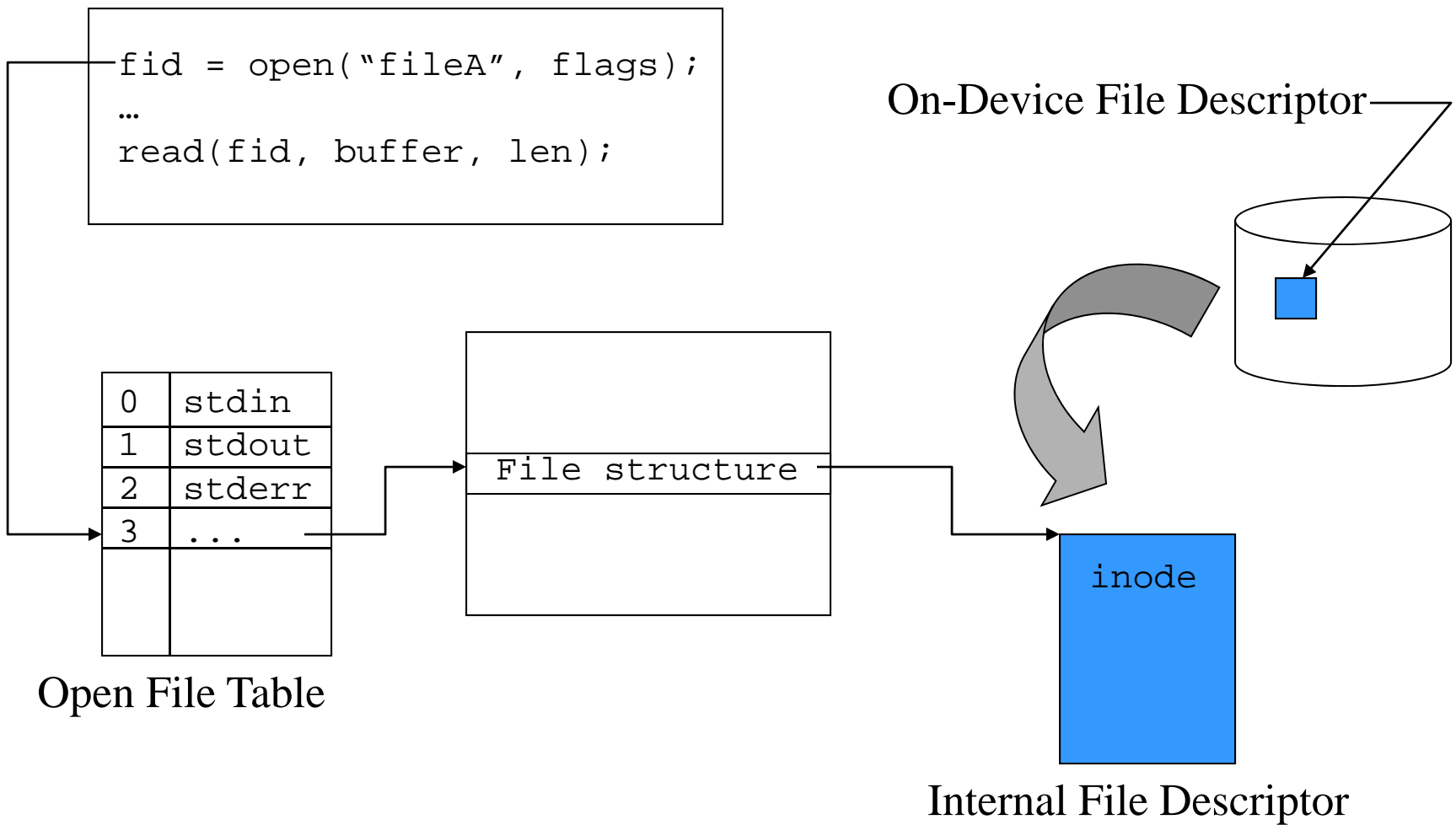
An `open()` Operation

- Locate the on-device (external) file descriptor
- Extract info needed to read/write file
- Authenticate that process can access the file
- Create an internal file descriptor in primary memory
- Create an entry in a “per process” open file status table
- Allocate resources, e.g., buffers, to support file usage

File Manager Data Structures



Opening a UNIX File



Block Management

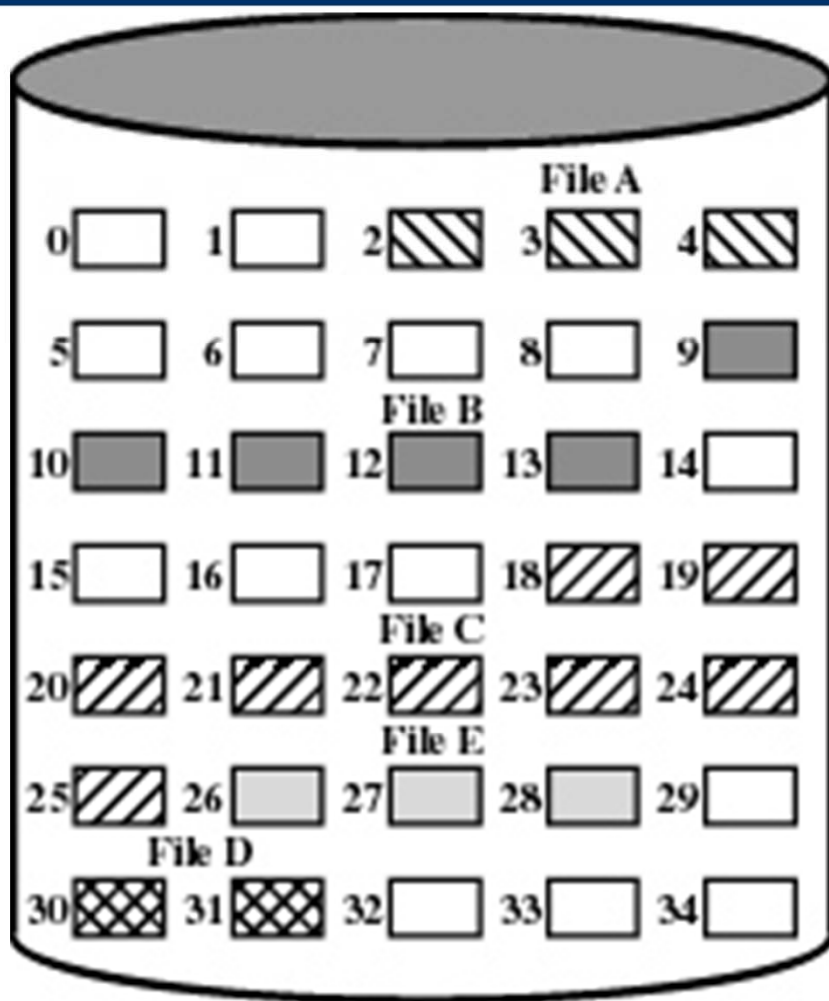
- The job of selecting & assigning storage blocks to the file
- For a fixed sized file of k blocks
 - File of length m requires $N = \lceil m/k \rceil$ blocks
 - Byte b_i is stored in block $\lfloor i/k \rfloor$
- Three basic strategies:
 - Contiguous allocation
 - Linked lists
 - Indexed allocation

Contiguous Allocation

- Maps the N blocks into N contiguous blocks on the secondary storage device
- Difficult to support dynamic file sizes

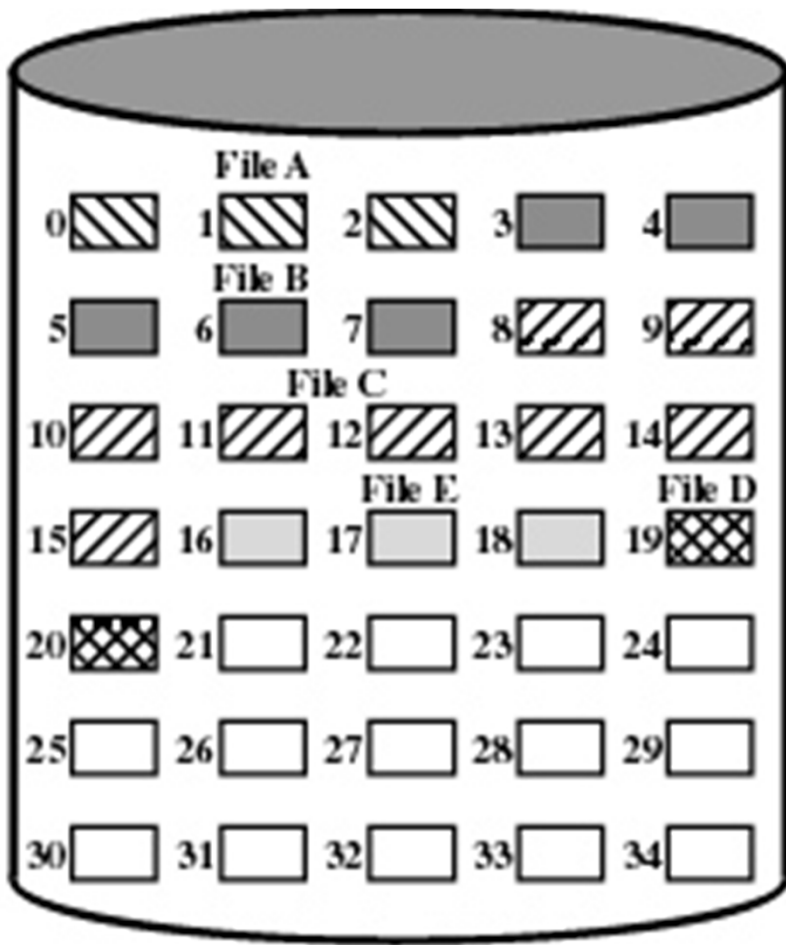
File descriptor

Head position	237
...	
First block	785
Number of blocks	25



File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3



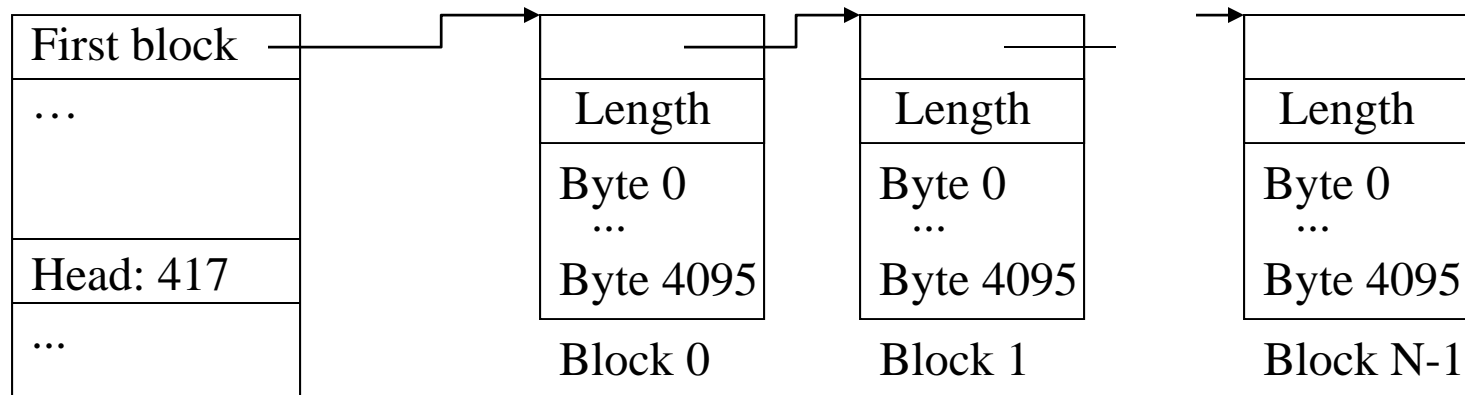
File Allocation Table

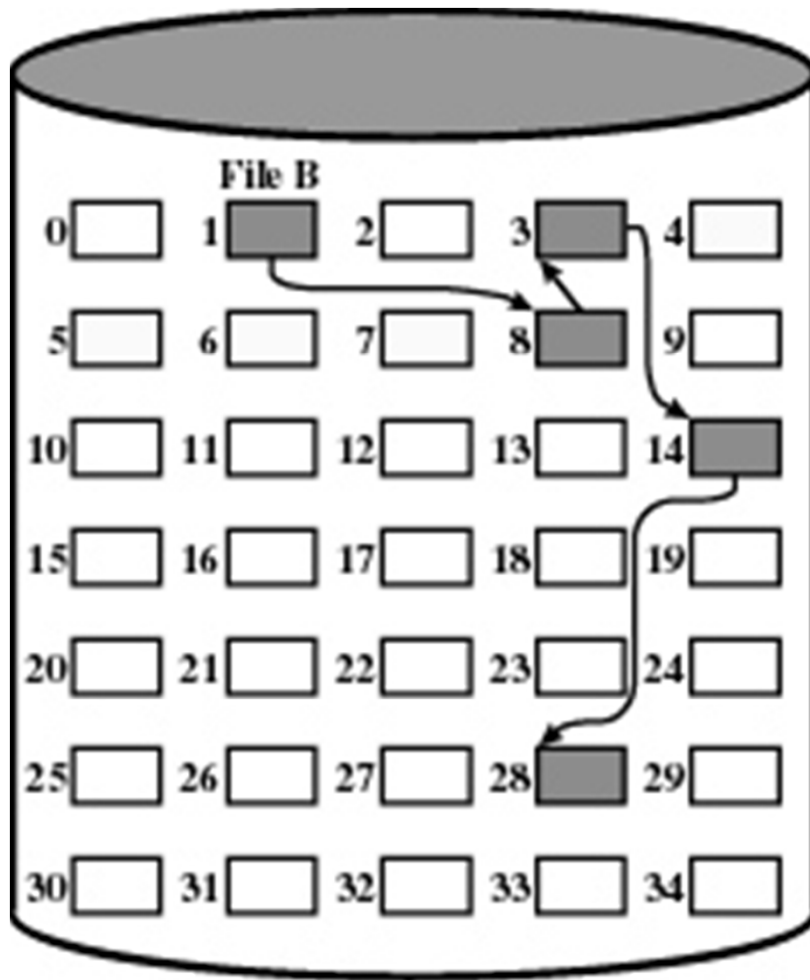
File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

After Compaction

Linked Lists or Chained Allocation

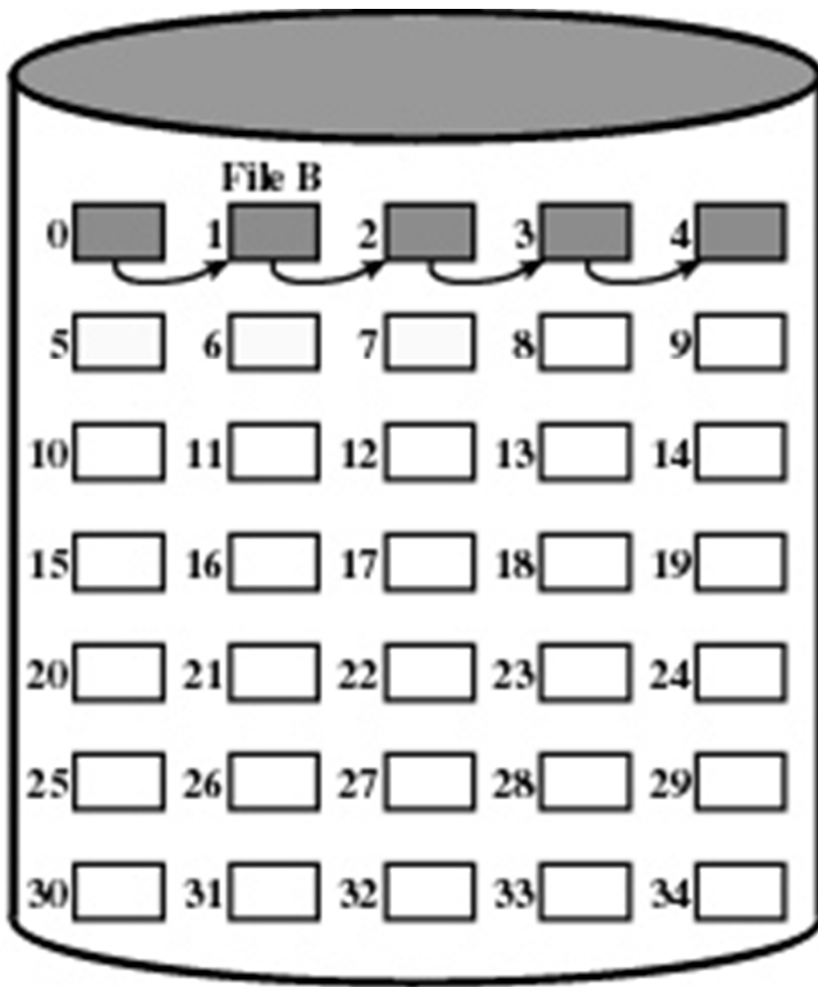
- Each block contains a header with
 - Number of bytes in the block
 - Pointer to next block
- Blocks need not be contiguous
- Files can expand and contract
- Seeks can be slow





File Allocation Table

File Name	Start Block	Length
...
File B	1	5
...



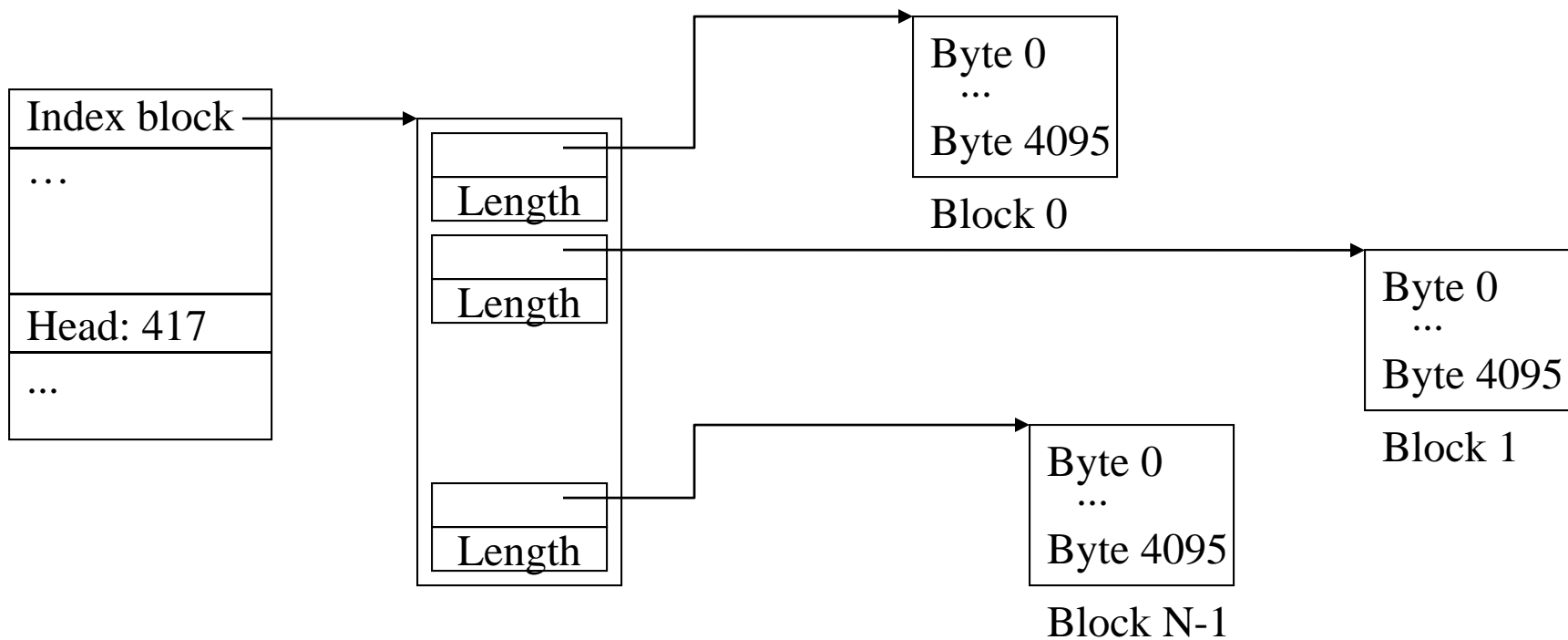
File Allocation Table

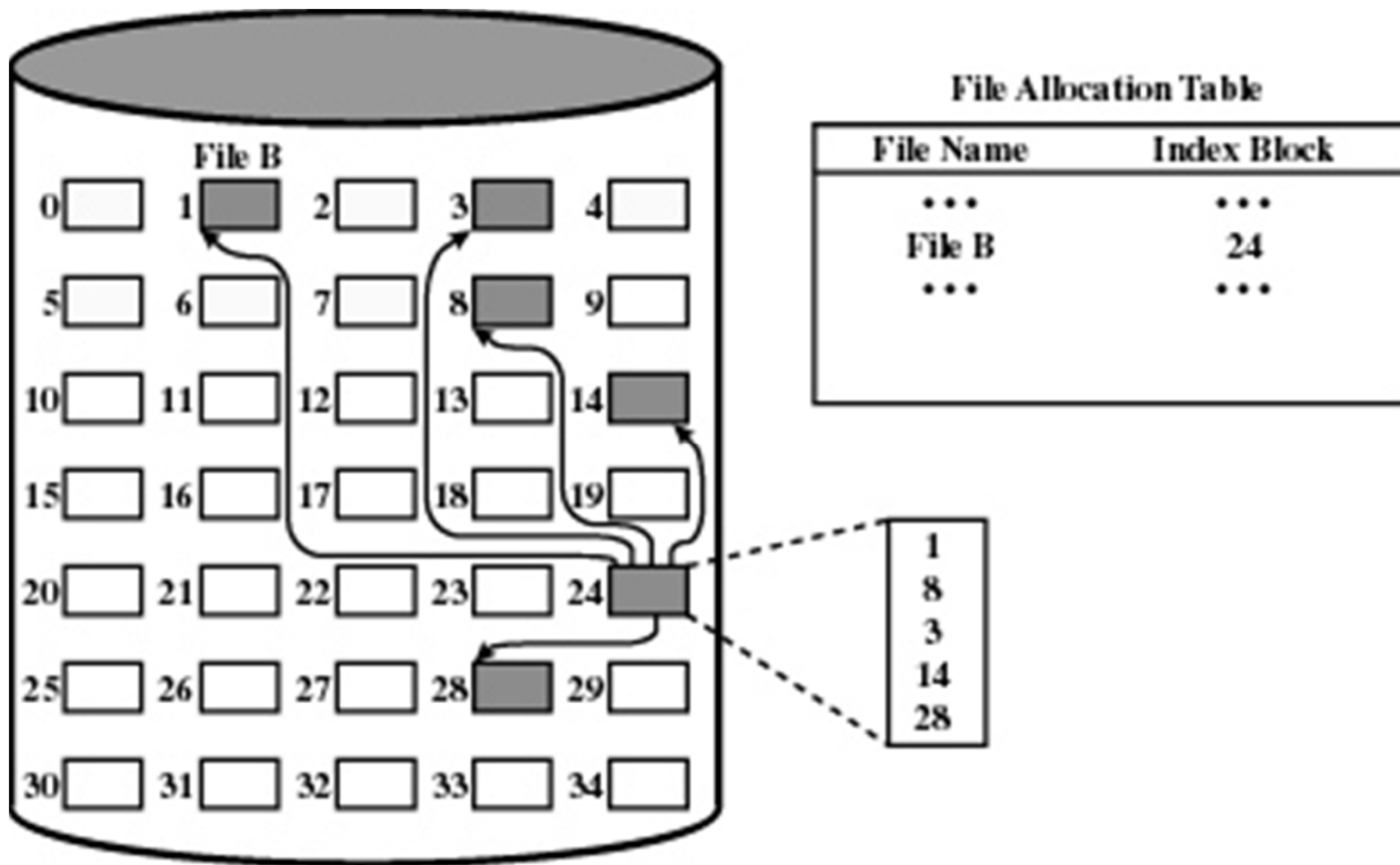
File Name	Start Block	Length
...
File B	0	5
...

After Consolidation

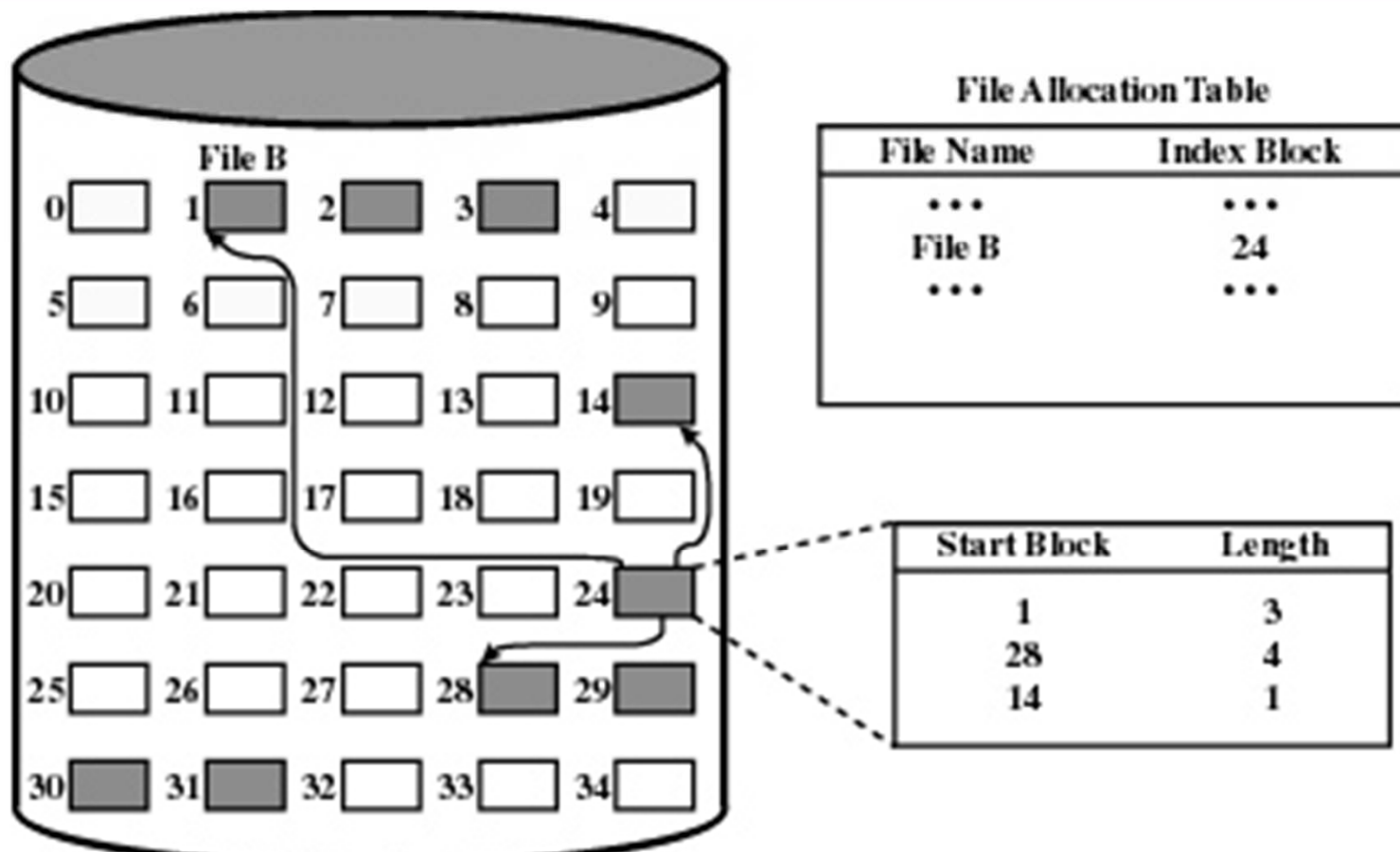
Indexed Allocation

- Extract headers and put them in an index
- Simplify seeks
- May link indices together (for large files)





Indexed allocation with Block portions



Indexed allocation with variable length portions

The original Unix file system

- Dennis Ritchie and Ken Thompson, Bell Labs, 1969
- “UNIX rose from the ashes of a multi-organizational effort in the early 1960s to develop a dependable timesharing operating system” – Multics
- Designed for a “workgroup” sharing a single system
- Did its job exceedingly well
 - Although it has been stretched in many directions and made ugly in the process
- A wonderful study in engineering tradeoffs



(Old) Unix disks are divided into five parts ...

- Boot block
 - can boot the system by loading from this block
- Superblock
 - specifies boundaries of next 3 areas, and contains head of freelists of inodes and file blocks
- i-node area
 - contains descriptors (i-nodes) for each file on the disk; all i-nodes are the same size; head of freelist is in the superblock
- File contents area
 - fixed-size blocks; head of freelist is in the superblock
- Swap area
 - holds processes that have been swapped out of memory

So ...

- You can attach a disk to a dead system ...
- Boot it up ...
- Find, create, and modify files ...
 - because the superblock is at a fixed place, and it tells you where the i-node area and file contents area are
 - by convention, the second i-node is the root directory of the volume

i-node format

- User number
- Group number
- Protection bits
- Times (file last read, file last written, inode last written)
- File code: specifies if the i-node represents a directory, an ordinary user file, or a “special file” (typically an I/O device)
- Size: length of file in bytes
- Block list: locates contents of file (in the file contents area)
 - more on this soon!
- Link count: number of directories referencing this i-node

The flat (i-node) file system

- Each file is known by a number, which is the number of the i-node
 - seriously – 1, 2, 3, etc.!
 - why is it called “flat”?
- Files are created empty, and grow when extended through writes

The tree (directory, hierarchical) file system

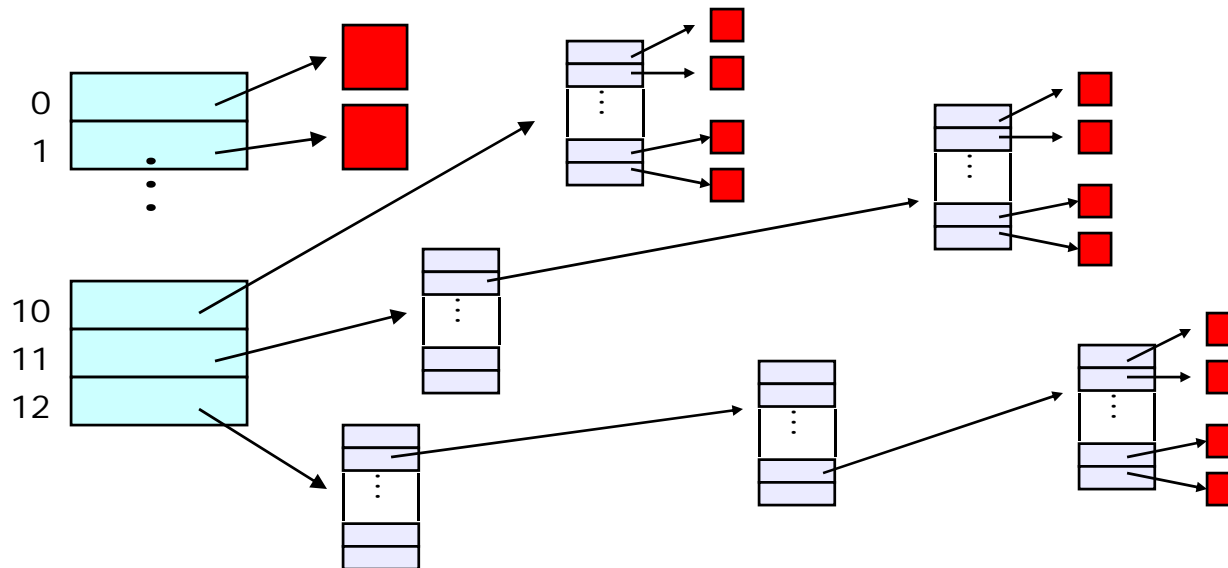
- A directory is a flat file of fixed-size entries
- Each entry consists of an i-node number and a file name

i-node number	File name
152	.
18	..
216	my_file
4	another_file
93	oh_my_god
144	a_directory

- It's as simple as that!

The “block list” portion of the i-node (Unix Version 7)

- Points to blocks in the file contents area
- Must be able to represent very small and very large files.
- Each inode contains 13 block pointers
 - first 10 are “direct pointers” (pointers to 512B blocks of file data)
 - then, single, double, and triple indirect pointers



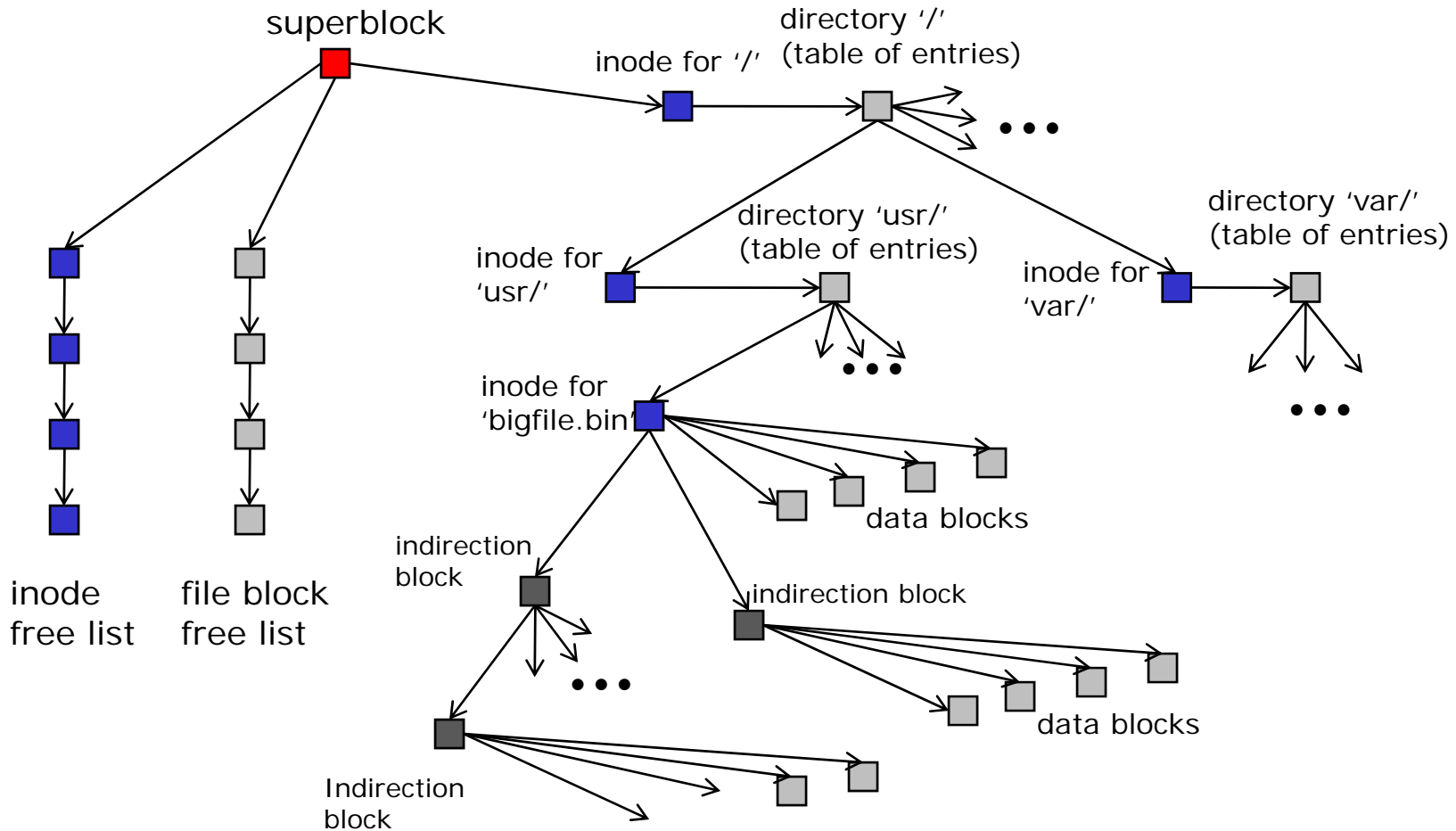
So ...

- Only occupies 13 x 4B in the i-node
- Can get to 10 x 512B = a 5120B file directly
 - (10 direct pointers, blocks in the file contents area are 512B)
- Can get to 128 x 512B = an additional 65KB with a single indirect reference
 - (the 11th pointer in the i-node gets you to a 512B block in the file contents area that contains 128 4B pointers to blocks holding file data)
- Can get to 128 x 128 x 512B = an additional 8MB with a double indirect reference
 - (the 12th pointer in the i-node gets you to a 512B block in the file contents area that contains 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks holding file data)

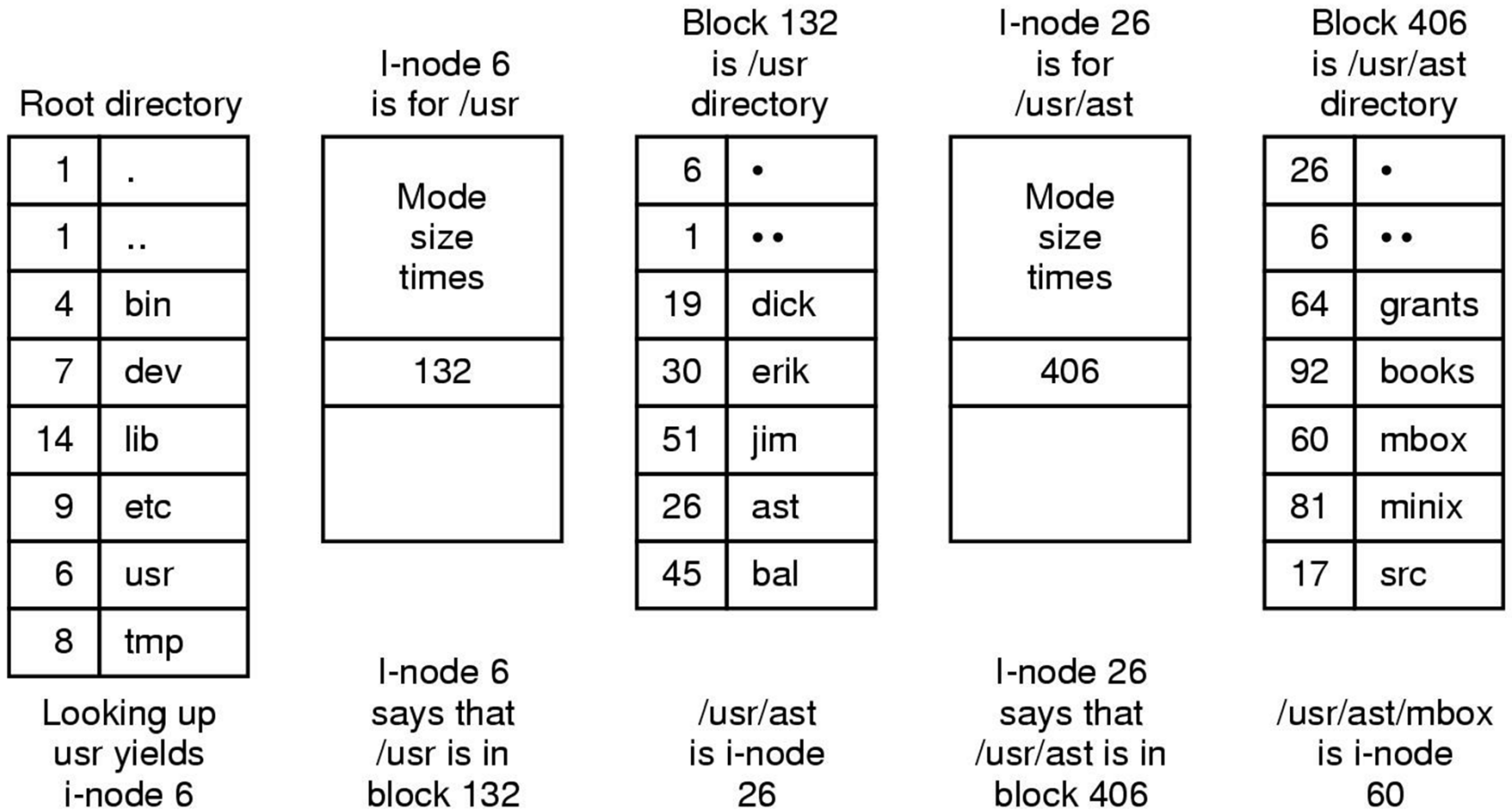
-
- Can get to $128 \times 128 \times 128 \times 512\text{B} =$ an additional 1GB with a triple indirect reference
 - (the 13th pointer in the i-node gets you to a 512B block in the file contents area that contains 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks holding file data)
 - Maximum file size is 1GB + a smidge
 - Berkeley Unix went to 1KB block sizes
 - What's the effect on the maximum file size?
 - $256 \times 256 \times 256 \times 1\text{K} = 17 \text{ GB} +$ a smidge
 - What's the price?
 - Subsequently went to 4KB blocks
 - $1\text{K} \times 1\text{K} \times 1\text{K} \times 4\text{K} = 4\text{TB} +$ a smidge

Putting it all together

- The file system is just a huge data structure



The UNIX File System



The steps in looking up `/usr/ast/mbox`

File system layout

- One important goal of a file system is to lay this data structure out on disk
 - have to keep in mind the physical characteristics of the disk itself (seeks are expensive)
 - and the characteristics of the workload (locality across files within a directory, sequential access to many files)
- Old UNIX's layout is very inefficient
 - constantly seeking back and forth between inode area and data block area as you traverse the file system, or even as you sequentially read files
- Newer file systems are smarter
- Newer storage devices (SSDs) change the constraints, but not the basic data structure

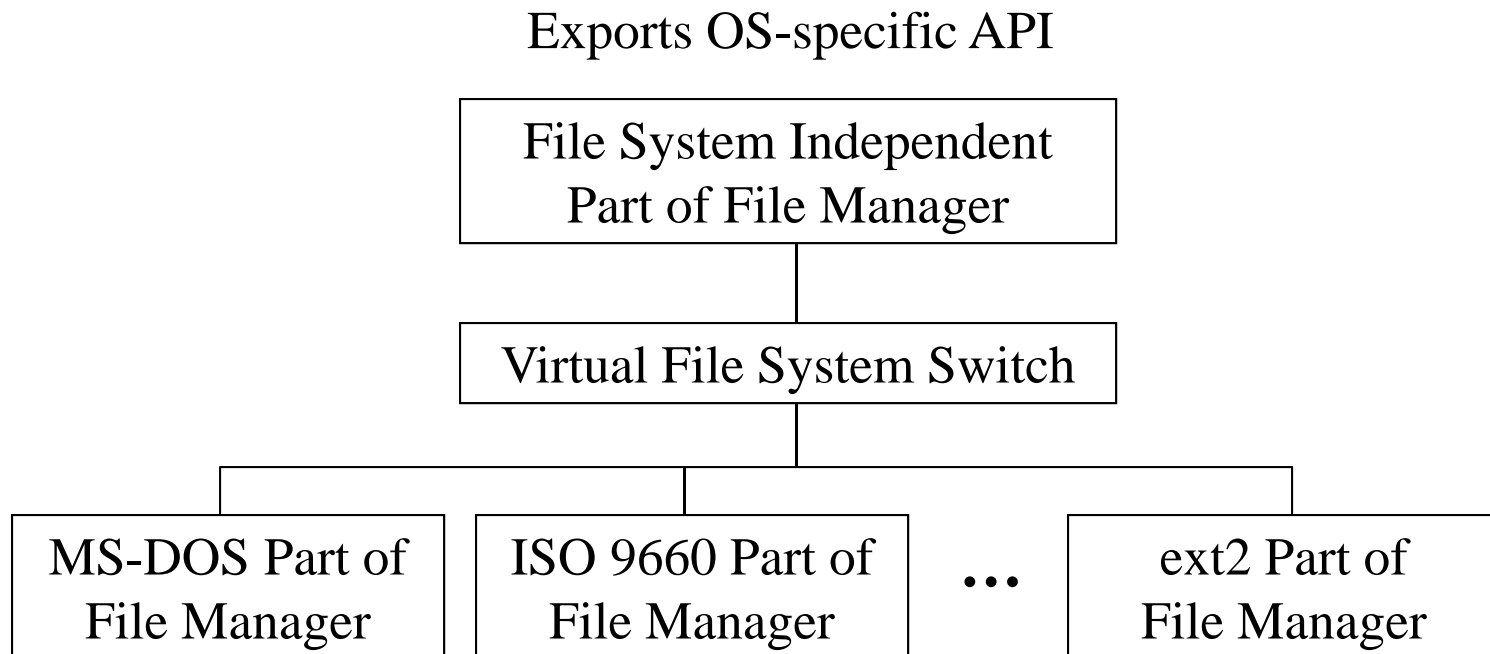
File system consistency

- Both i-nodes and file blocks are cached in memory
- The “sync” command forces memory-resident disk information to be written to disk
 - system does a sync every few seconds
- A crash or power failure between sync’s can leave an inconsistent disk
- You could reduce the frequency of problems by reducing caching, but performance would suffer big-time

What do you do after a crash?

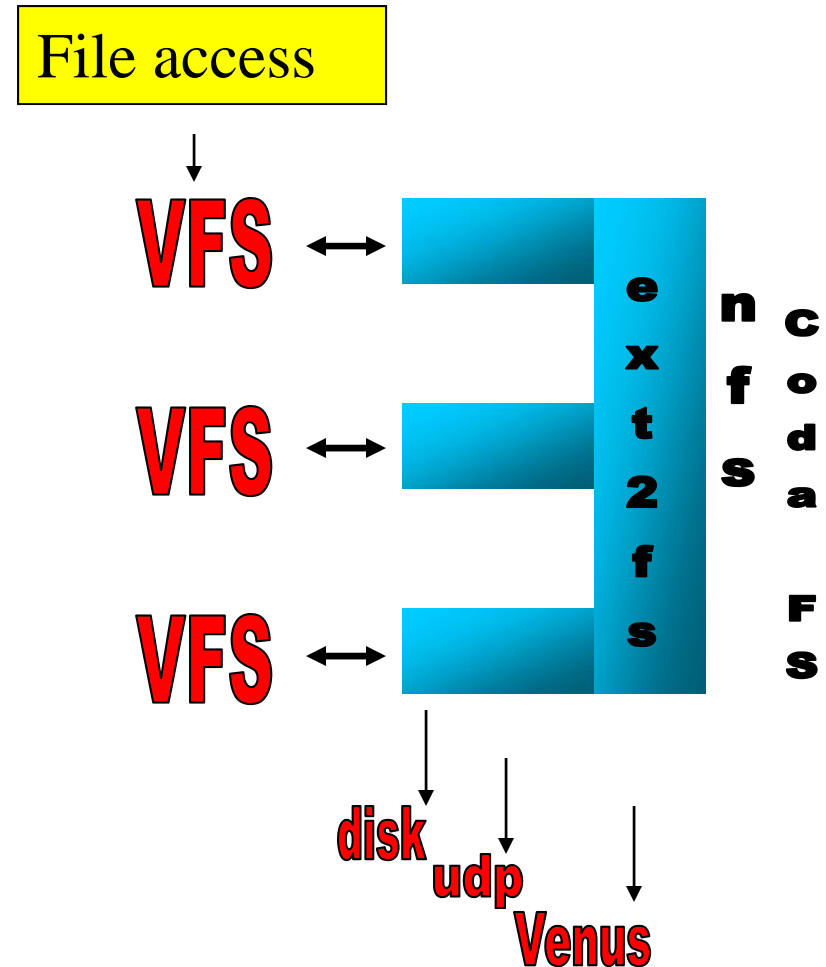
- Run a program called “fsck” to try to fix any consistency problems
- fsck has to scan the entire disk
 - as disks are getting bigger, fsck is taking longer and longer
 - modern disks: fsck can take a full day!
- Newer file systems try to help here
 - are more clever about the order in which writes happen, and where writes are directed
 - o e.g., Journaling file system: collect recent writes in a log called a journal. On crash, run through journal to replay against file system.

VFS-based File Manager



Linux VFS

- Multiple interfaces build up VFS:
 - files
 - dentries
 - inodes
 - superblock
 - quota
- VFS can do all caching & provides utility fctns to FS
- FS provides methods to VFS; many are optional



User level file access

- **Typical user level types and code:**
 - **pathnames:** `"/myfile"`
 - **file descriptors:** `fd = open("/myfile"...)`
 - **attributes** in struct stat: `stat("/myfile", &mybuf), chmod, chown...`
 - **offsets:** `write, read, lseek`
 - **directory handles:** `DIR *dh = opendir("/mydir")`
 - **directory** entries: `struct dirent *ent = readdir(dh)`

VFS

- Manages kernel level file abstractions in one format for all file systems
- Receives system call requests from user level (e.g. write, open, stat, link)
- Interacts with a specific file system based on mount point traversal
- Receives requests from other parts of the kernel, mostly from memory management

File system level

- **Individual File Systems**

- responsible for managing **file & directory data**
- responsible for managing **meta-data**: timestamps, owners, protection etc
- translates data between
 - o **particular FS data**: e.g. disk data, NFS data, Coda/AFS data
 - o **VFS data**: attributes etc in standard format
- e.g. `nfs_getattr(...)` returns attributes in VFS format, acquires attributes in NFS format to do so.

Anatomy of **stat** system call

```
sys_stat(path, buf) {  
    dentry = namei(path);  
    if ( dentry == NULL ) return -  
ENOENT;
```

Establish VFS data

```
    inode = dentry->d_inode;  
    rc = inode->i_op-  
>i_permission(inode);  
    if ( rc ) return -EPERM;
```

Call into inode
layer of filesystem

```
    rc = inode->i_op-  
>i_getattr(inode, buf);  
    dput(dentry);  
    return rc;  
}
```

Call into inode
layer of filesystem

Anatomy of `fstatfs` system call

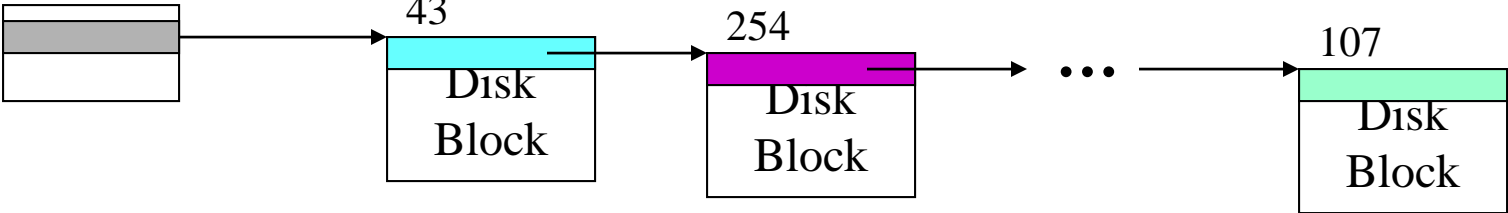
```
sys_fstatfs(fd, buf) {           /* for things
like "df" */
    file = fget(fd);             Translate fd to
    if ( file == NULL ) return -EBADF; VFS data
                                    structure

    superb = file->f_dentry->d_inode- Call into
    >i_super;                      superblock layer
                                    of filesystem

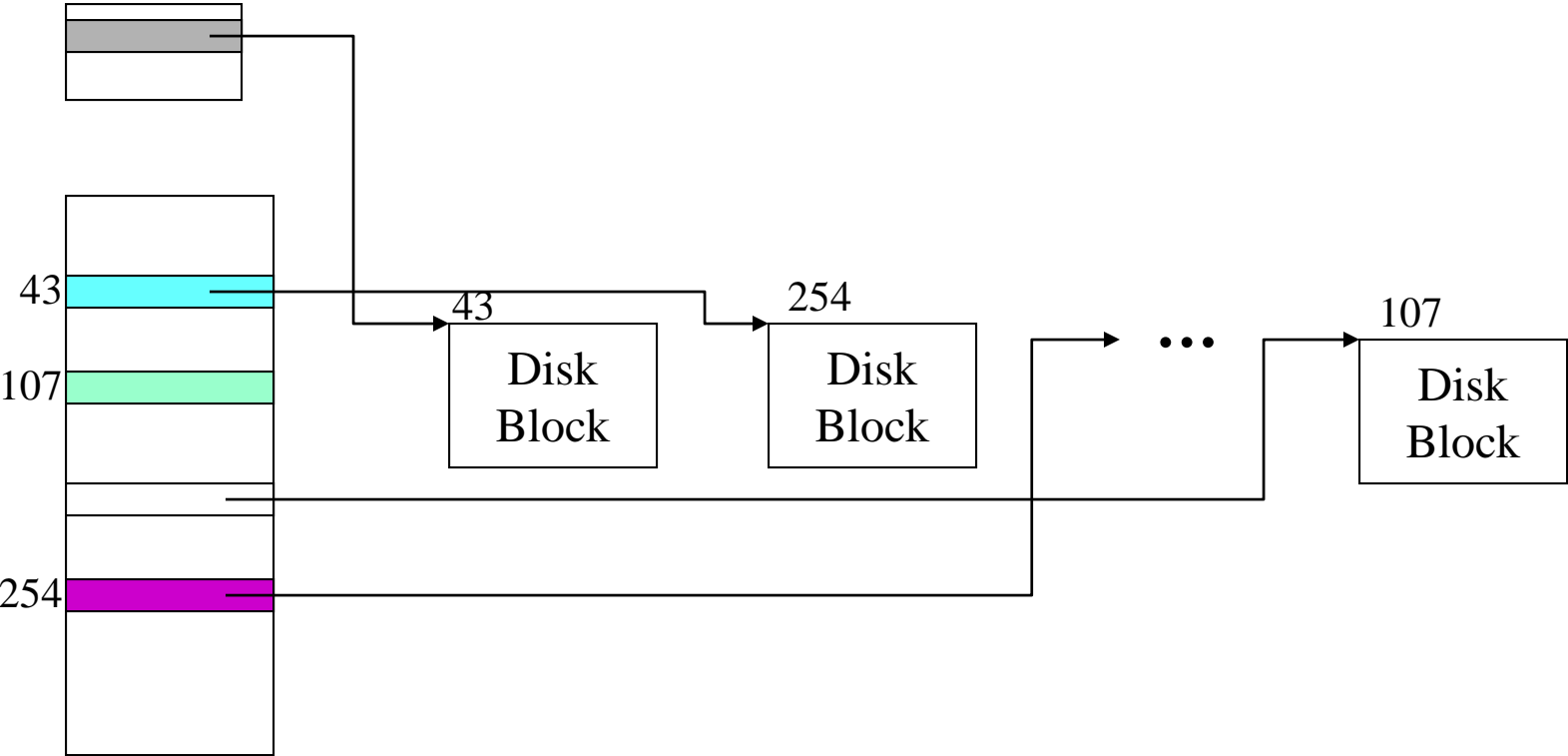
    rc = superb->sb_op->sb_statfs(sb, buf);
    return rc;
}
```

DOS FAT Files

File Descriptor



File Descriptor



File Access Table (FAT)