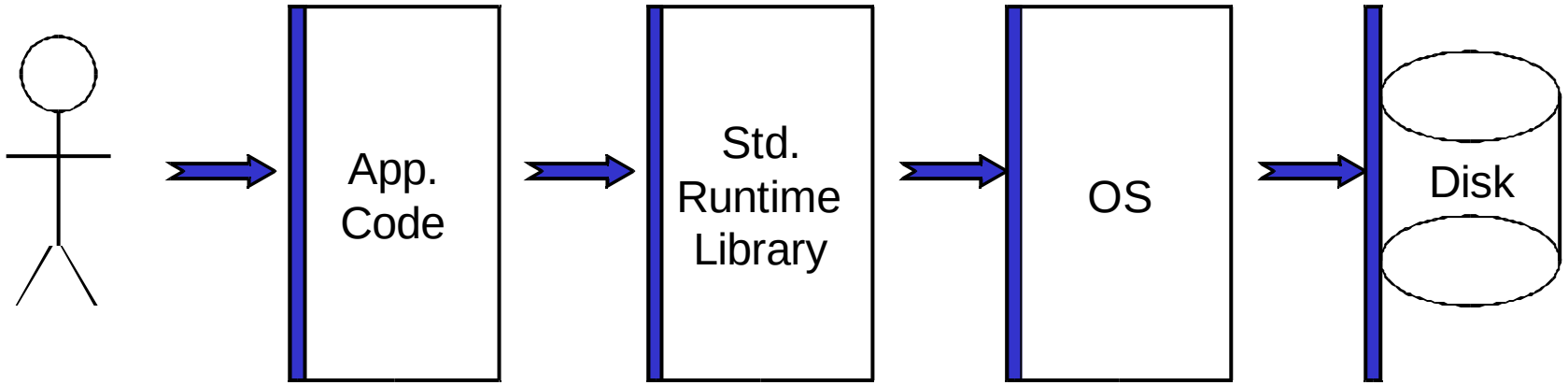
CSE 451: Operating Systems
Spring 2017

Module 13
File Systems

John Zahorjan

Interface Layers



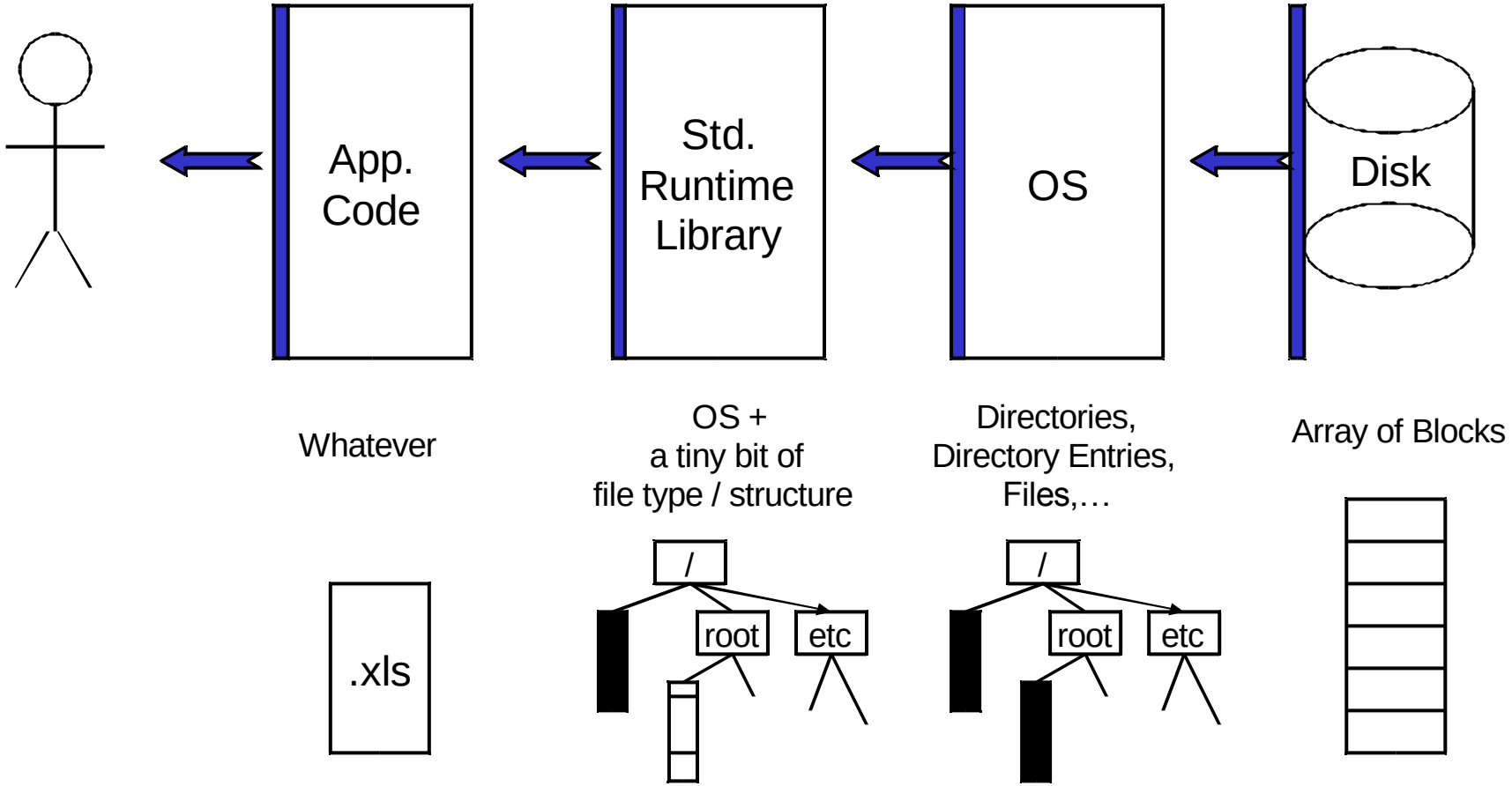
Procedure
Calls

Device-type
Dependent Commands

Whatever…

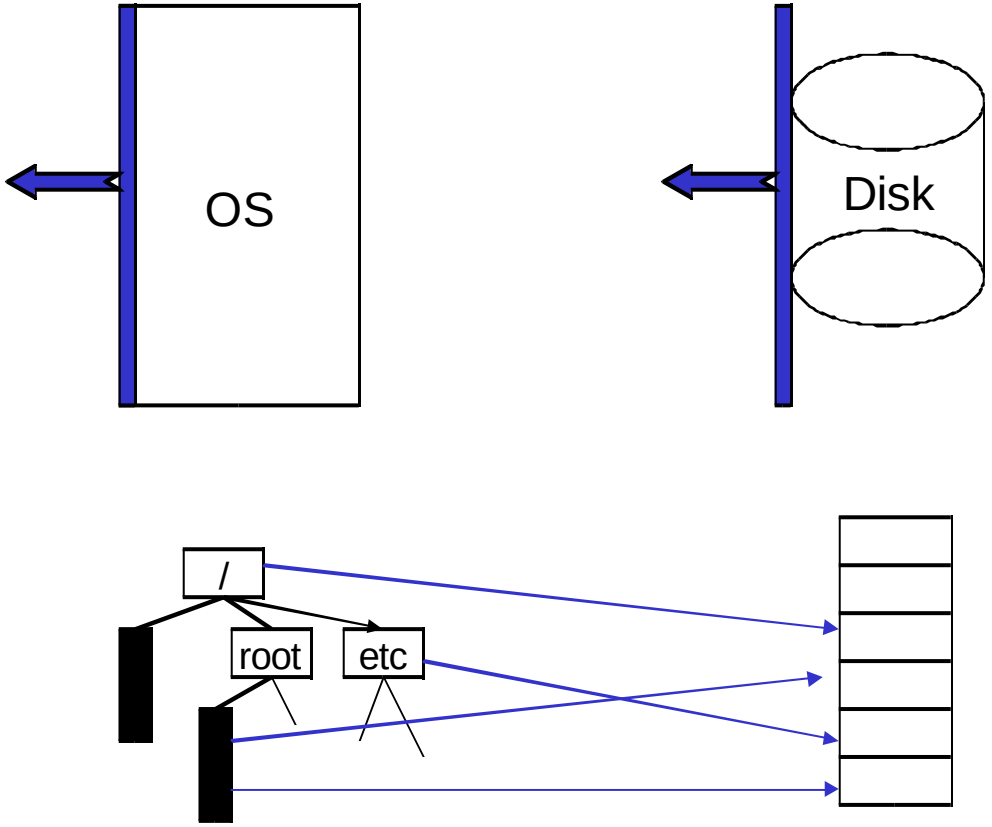
Syscalls

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 systems

- The concept of a file system is simple
  - the implementation of the abstraction for secondary storage
    - abstraction = files
  - logical organization of files into directories
    - the directory hierarchy
  - sharing of data among processes, people and machines
    - access control, consistency, …

Files

- A file is a collection of data with some properties (*metadata*)
  - contents, size, owner, last read/write time, protection …
- Files may also have types
  - understood by file system
    - device, directory, symbolic link
  - understood by other parts of OS or by runtime libraries
    - executable, dll, source code, object code, text file, …
- Type can be encoded in the file's name or contents
  - windows encodes type in name
    - .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
    - in content via magic numbers or initial characters (e.g., #!)

Brief Aside

- Is there a difference between a file's and the file itself?
  - Can a file have more than one name?
- File systems typically provide a namespace
  - Directories contain information that maps from a name for a file to the contents of the file
- The file system provided namespace has some scope
  - E.g., that system; that workgroup; that LAN; …
- Sometimes the namespace is borrowed for things that aren't "files"
  - /proc/cpuinfo

Basic operations

Unix

- create(name)
- open(name, mode)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)
- rename(old, new)

Windows

- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, …)
- WriteFile(handle, …)
- FlushFileBuffers(handle, …)
- SetFilePointer(handle, …)
- CloseHandle(handle, …)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)

File access methods

- Some file systems provide different access methods that specify ways the application will access data
  - sequential access
    - read bytes one at a time, in order
  - direct access
    - random access given a block/byte #
  - record access
    - file is array of fixed- or variable-sized records
  - indexed access
    - FS contains an index to a particular field of each record in a file
    - 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?

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)
    ```

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);
```

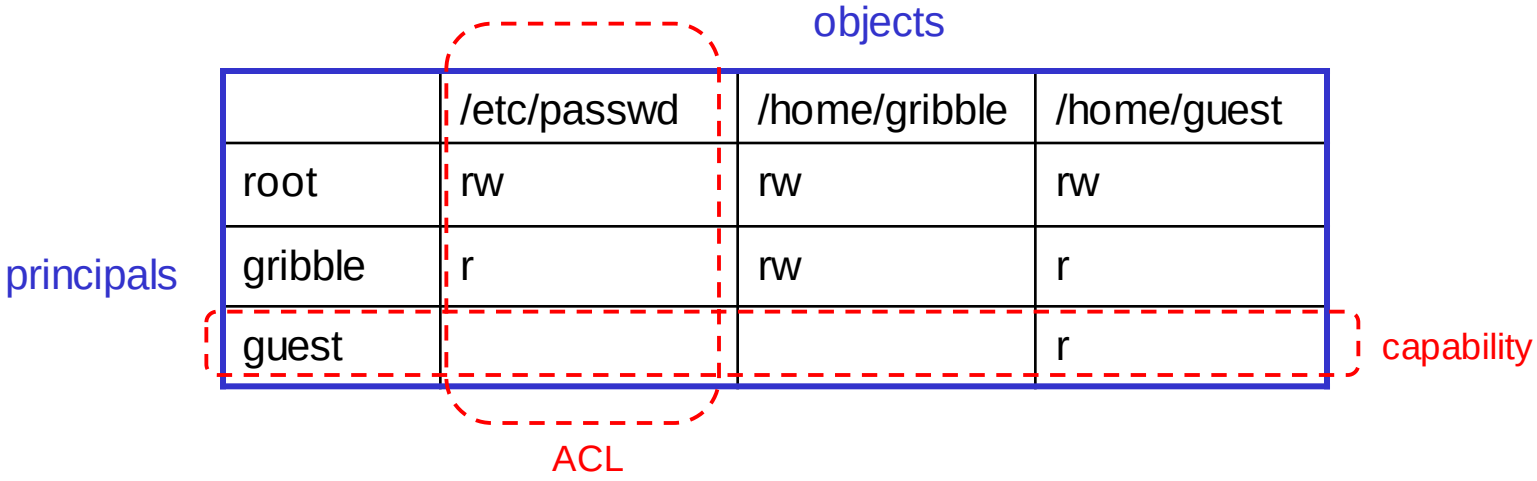
- What goes on inside the file system?
  - open directory "/" (well known, can always find)
  - search the directory for "one", get location of "one"
  - open directory "one", search for "two", get location of "two"
  - open directory "two", search for "three", get loc. of "three"
  - open file "three"
  - (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
    - /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., your 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:



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

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

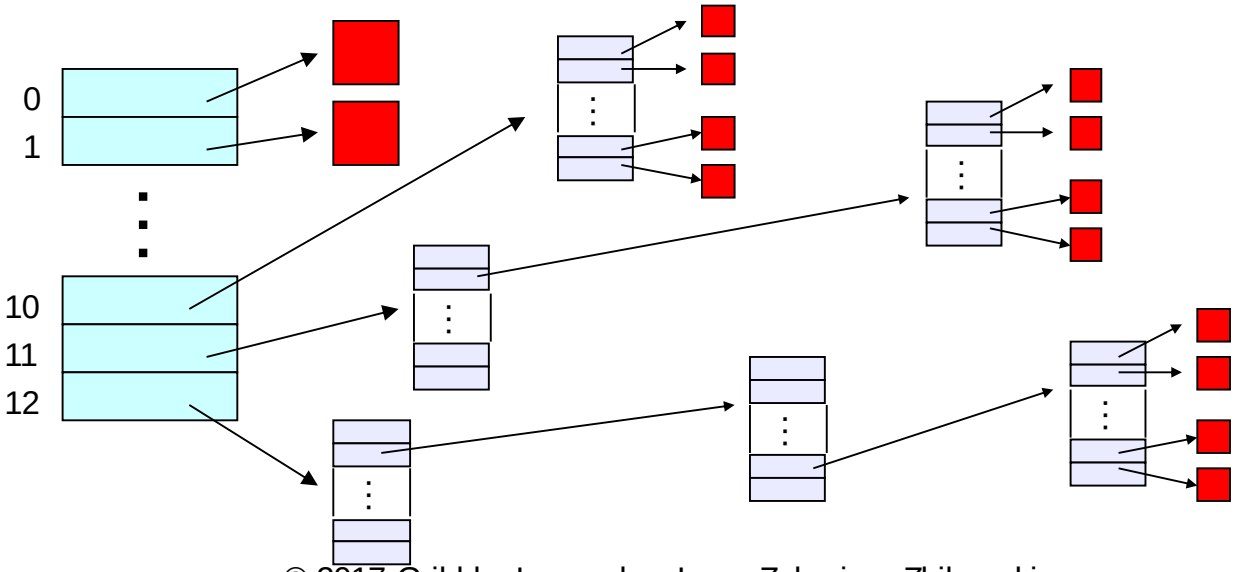
- A directory is a flat file of fixed-size entries
- Each entry consists of an i-node number and a file name

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. How?
- 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 …

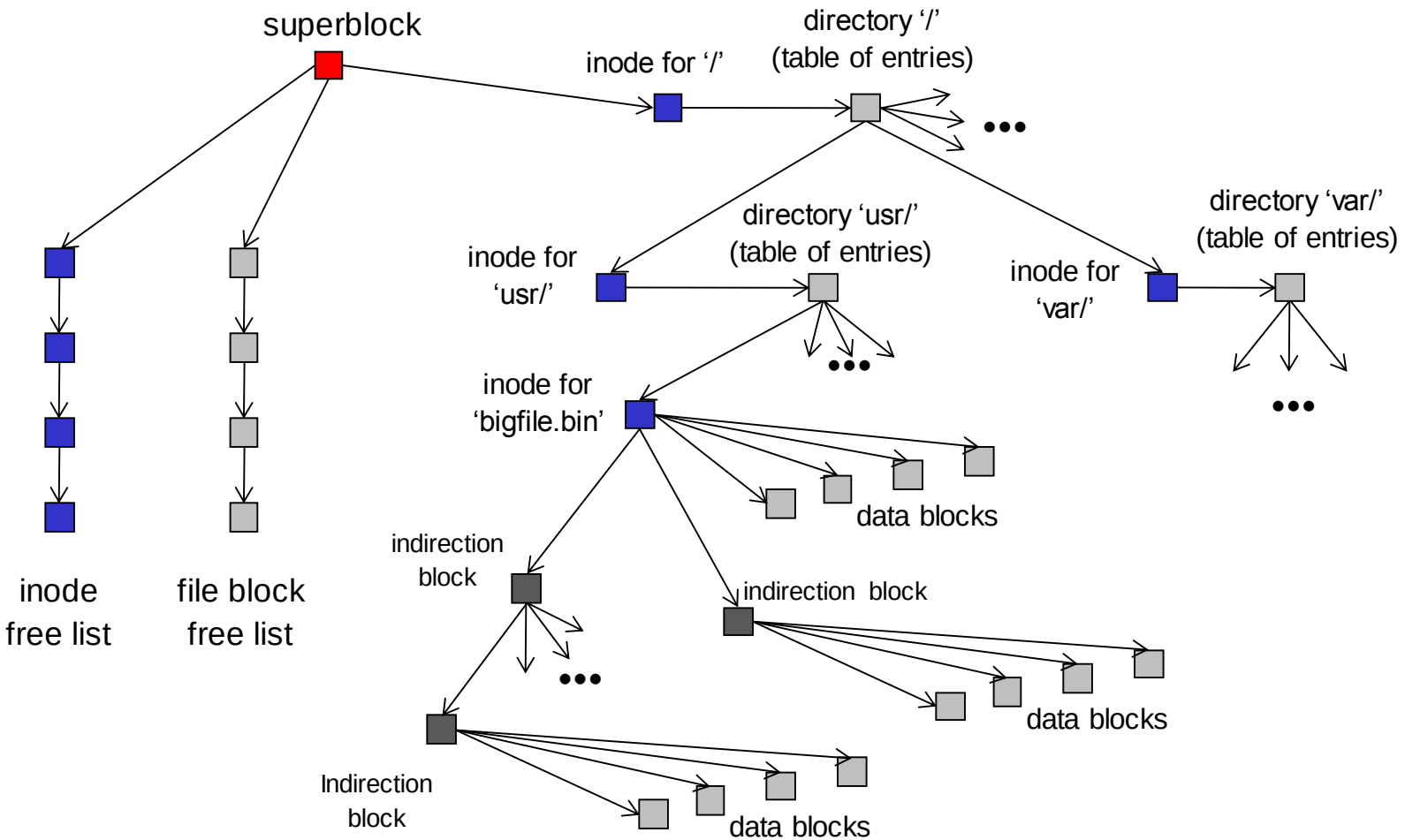
- Only occupies 13 x 4B in the i-node
- Can get to 10 x 512B = a 5120B file directly
  - (10 direct pointers, blocks in the file contents area are 512B)
- Can get to 128 x 512B = an additional 65KB with a single indirect reference
  - (the 11th pointer in the i-node gets you to a 512B block in the file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

- A later version of Bell Labs Unix utilized 12 direct pointers rather than 10
  - Why?
- Berkeley Unix went to 1KB block sizes
  - What's the effect on the maximum file size?
    - 256x256x256x1K = 17 GB + a smidge
  - What's the price?
- Subsequently went to 4KB blocks
  - 1Kx1Kx1Kx4K = 4TB + a smidge

Putting it all together

- The file system is just a huge data structure



File system layout

- One important goal of a filesystem 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 filesystem, 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 were getting bigger, fsck was 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
    - e.g., Journaling file system: collect recent writes in a log called a journal. On crash, run through journal to replay against file system.

fsck i-check (consistency of the flat file system)

- Is each block on exactly one list?
  - create a bit vector with as many entries as there are blocks
  - follow the free list and each i-node block list
  - when a block is encountered, examine its bit
    - If the bit was 0, set it to 1
    - if the bit was already 1
      - if the block is both in a file and on the free list, remove it from the free list and cross your fingers
      - if the block is in two files, call support!
  - if there are any 0's left at the end, put those blocks on the free list

fsck d-check (consistency of the directory file system)

- Do the directories form a tree?
- Does the link count of each file equal the number of directories linked to it?
  - I will spare you the details
    - uses a zero-initialized vector of counters, one per i-node
    - walk the tree, then visit every i-node

Protection

- Objects: individual files
- Principals: owner/group/world
- Actions: read/write/execute

- This is pretty simple and rigid, but it has proven to be about what we can handle!

File sharing

- Each user has a "channel table" (or "per-user open file table")
- Each entry in the channel table is a pointer to an entry in the system-wide "open file table"
- Each entry in the open file table contains a file offset (file pointer) and a pointer to an entry in the "memory-resident i-node table"
- If a process opens an already-open file, a new open file table entry is created (with a new file offset), pointing to the same entry in the memory-resident i-node table
- If a process forks, the child gets a copy of the channel table (and thus the same file offset)