

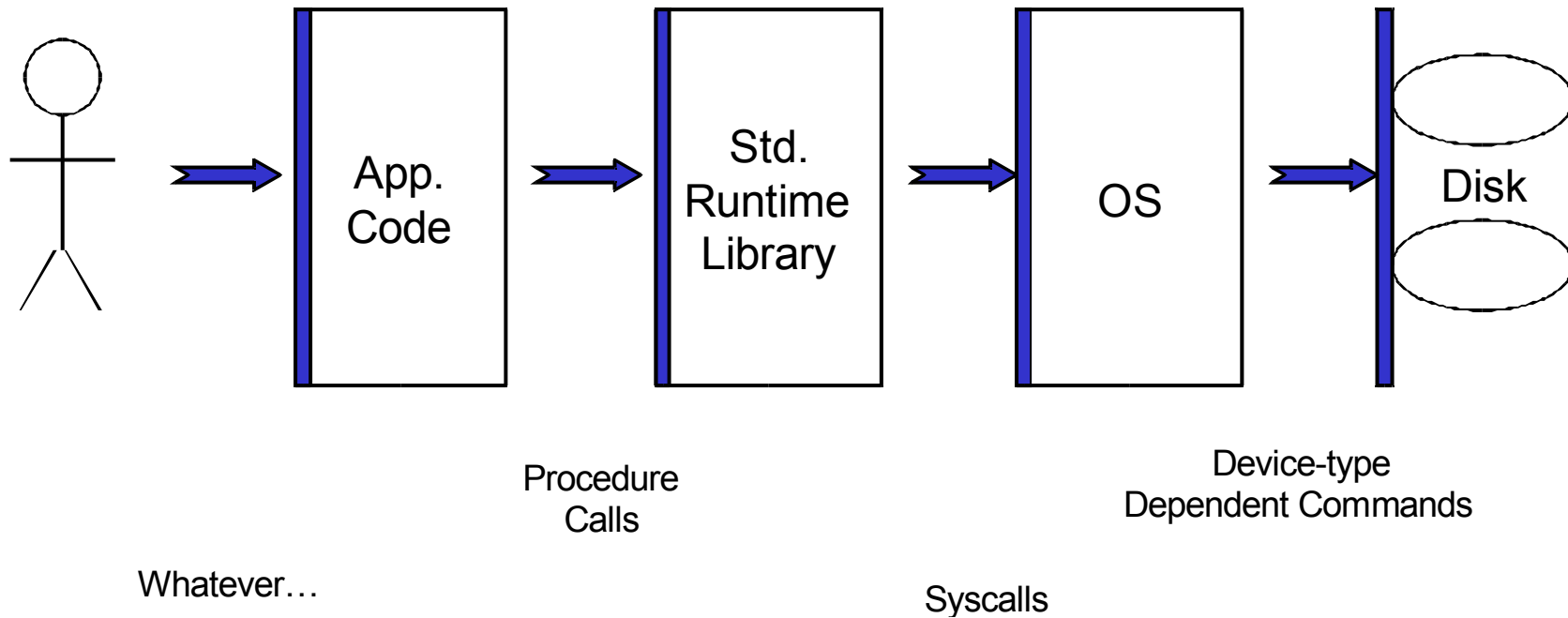
Operating Systems

Fall 2014

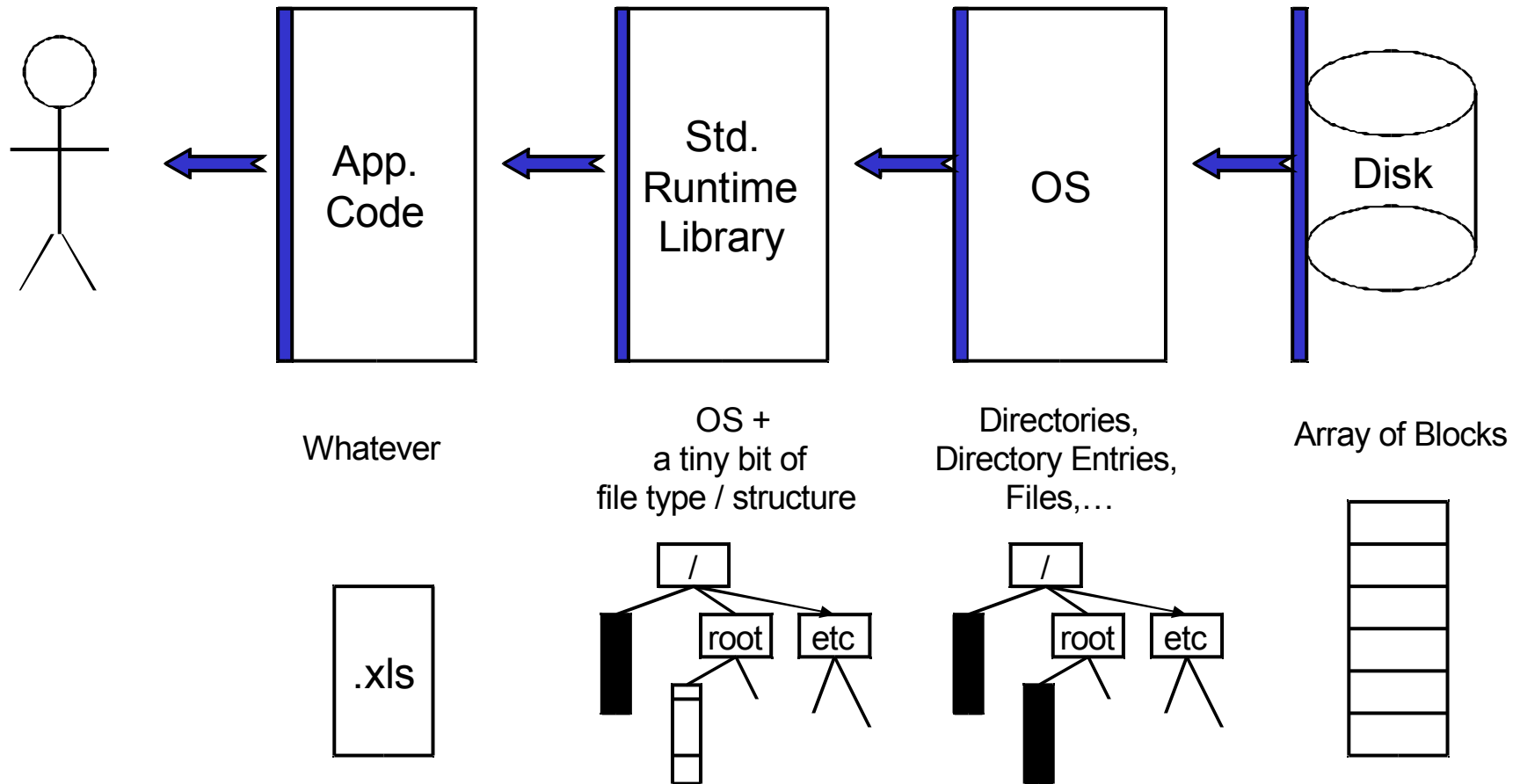
File Systems

Myungjin Lee
myungjin.lee@ed.ac.uk

Interface Layers

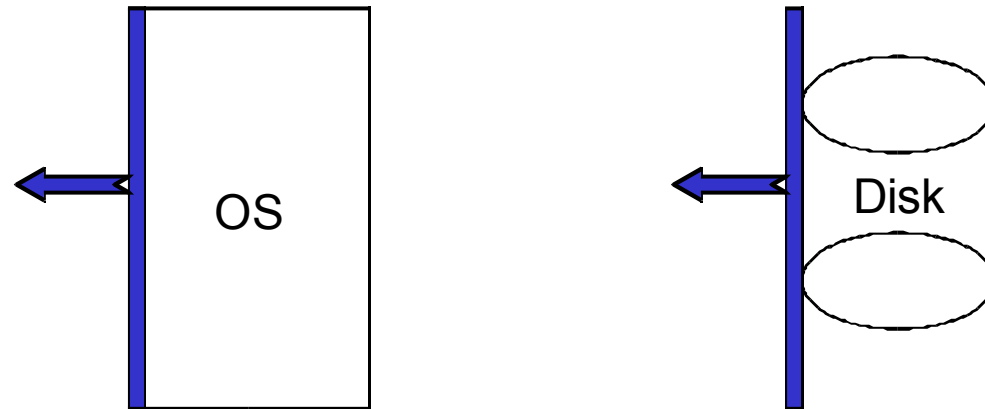


Exported Abstractions



Primary Roles of the OS (file system)

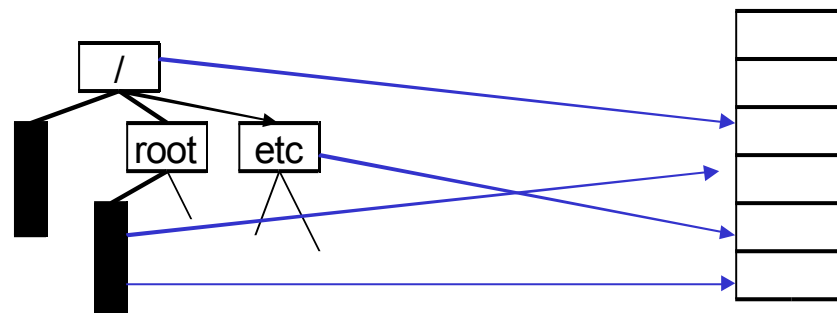
1. Hide hardware specific interface
2. Allocate disk blocks
3. Check permissions
4. Understand directory file structure
5. Maintain *metadata*
6. Performance
7. Flexibility



Why does the OS define directories?

Why not leave that to the library/application layer?

(Why would you want to leave it to the app/library?)



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 between processes, people and machines
 - access control, consistency, ...

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
 - 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., #!)

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., you can read `/group/teaching/cs3` 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

ACL

capability

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 ...

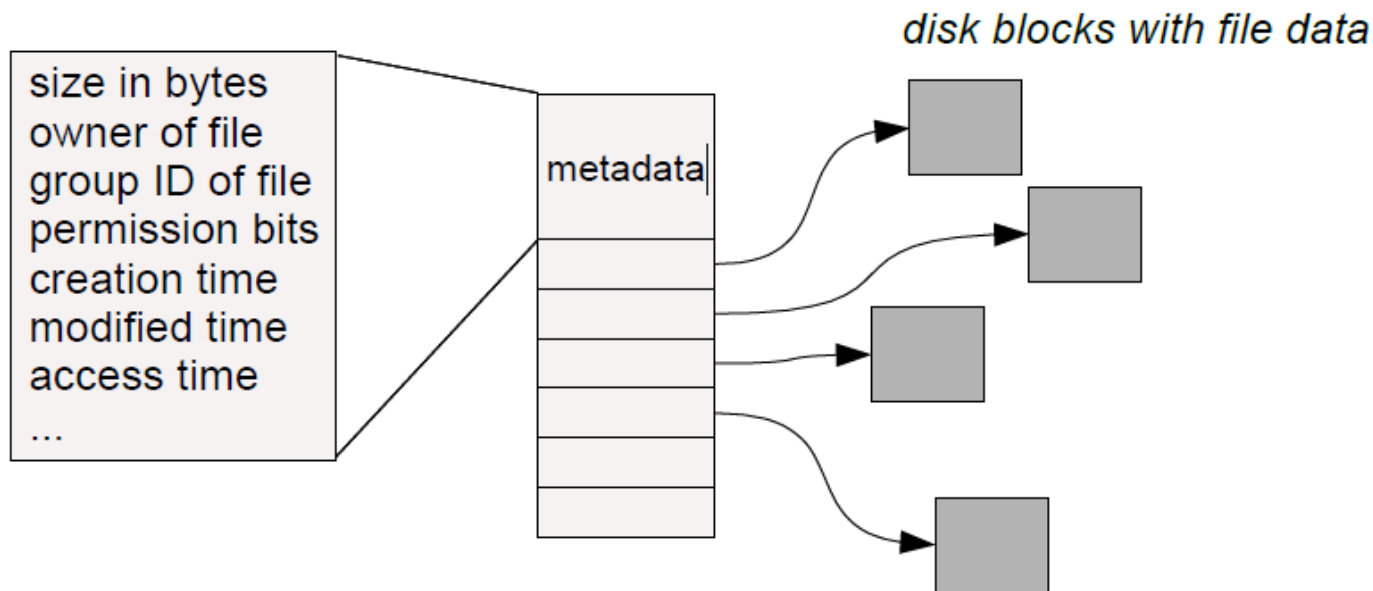
- **Boot block**
 - can boot the system by loading from this block
- **Superblock**
 - specifies boundaries of next 3 areas, and contains head of freelists of inodes and file blocks
- **inode area**
 - contains descriptors (inodes) for each file on the disk; all inodes are the same size; head of freelist 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 inode area and file contents area are

Basic file system structures

- Every file and directory is represented by an inode
 - Stands for “index node”
- Contains two kinds of information:
 - 1) Metadata describing the file's owner, access rights, etc.
 - 2) Location of the file's blocks on disk



inode format

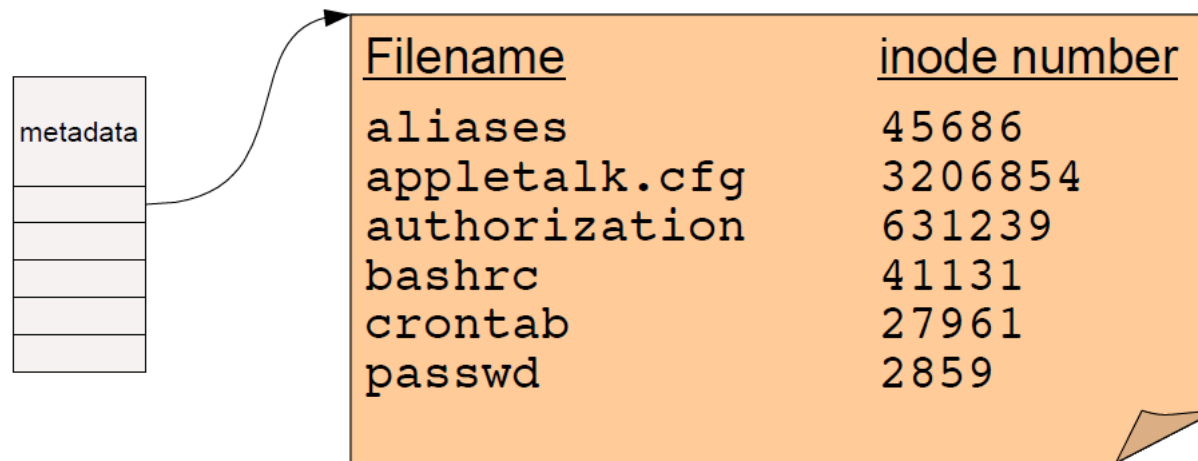
- User number
- Group number
- Protection bits
- Times (file last read, file last written, inode last written)
- File code: specifies if the inode represents a directory, an ordinary user file, or a “special file” (typically an I/O device)
- Size: length of file in bytes
- Block list: locates contents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

The flat (inode) file system

- Each file is known by a number, which is the number of the inode
 - 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

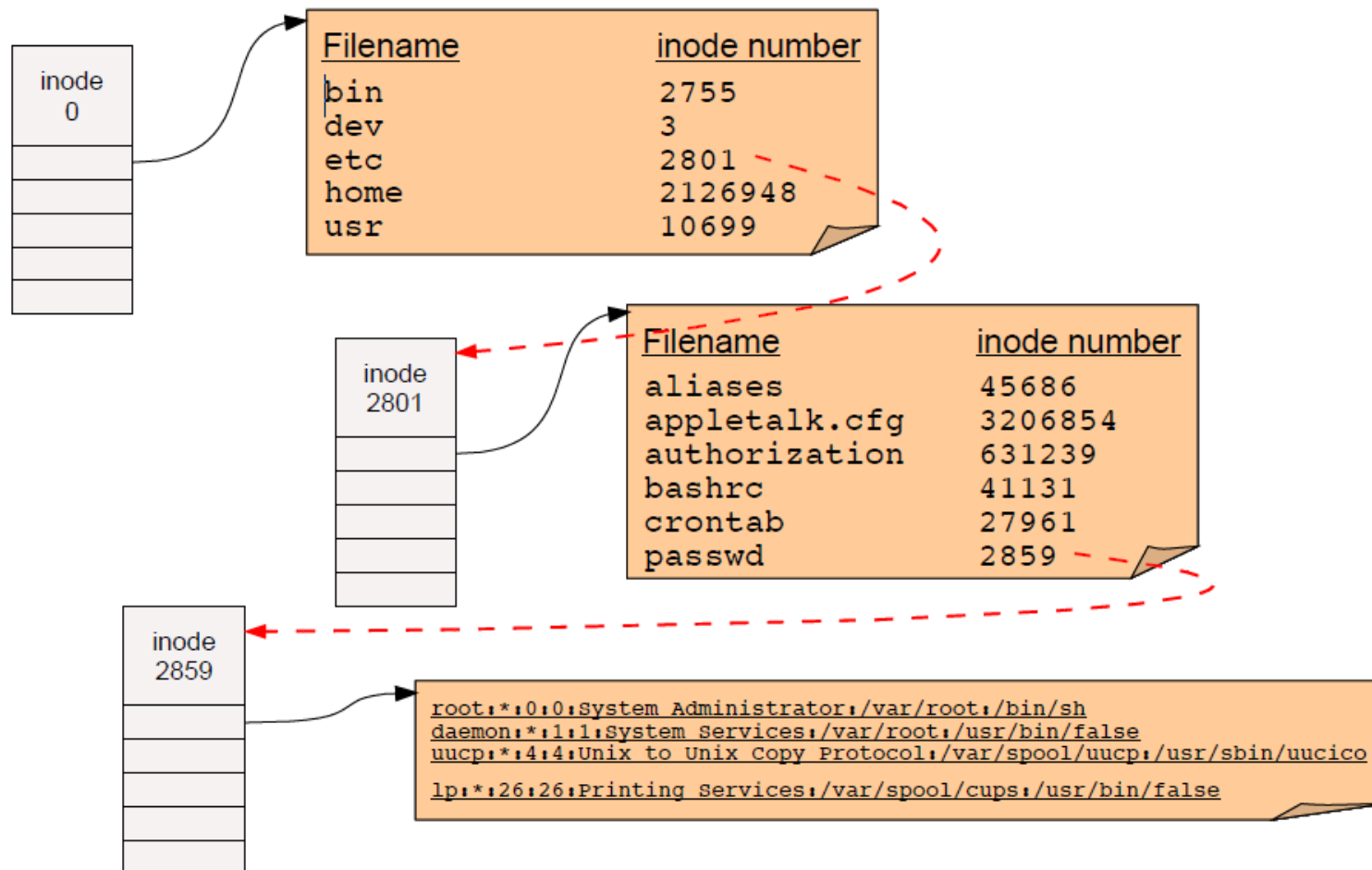
- A directory is a flat file of fixed-size entries
- Each entry consists of an inode number and a file name



- These are the contents of the directory “file data” itself – NOT the directory's inode!
- Filenames (in UNIX) are not stored in the inode at all!
- Two open questions:
 - How do we find the root directory (“ / ” on UNIX systems)?
 - How do we get from an inode number to the location of the inode on disk?

Pathname resolution

- To look up a pathname “/etc/passwd”, start at root directory and walk down chain of inodes...



Locating inodes on disk

- All right, so directories tell us the **inode number** of a file.
 - How the heck do we find the inode itself on disk?
- Basic idea: Top part of filesystem contains *all* of the inodes!



superblock

inodes

File and directory data blocks

- inode number is just the “index” of the inode
- Easy to compute the block address of a given inode:
 - $\text{block_addr}(\text{inode_num}) = \text{block_offset_of_first_inode} + (\text{inode_num} * \text{inode_size})$
- This implies that a filesystem has a fixed number of potential inodes
 - This number is generally set when the filesystem is created
- The superblock stores important metadata on filesystem layout, list of free blocks, etc.

Stupid directory tricks

- Directories map filenames to inode numbers. What does this imply?
- We can create multiple pointers to the same inode in different directories
 - Or even the same directory with different filenames
- In UNIX this is called a “hard link” and can be done using “ln”

```
bash$ ls -i /home/foo
287663 /home/foo          (This is the inode number of “foo”)
```

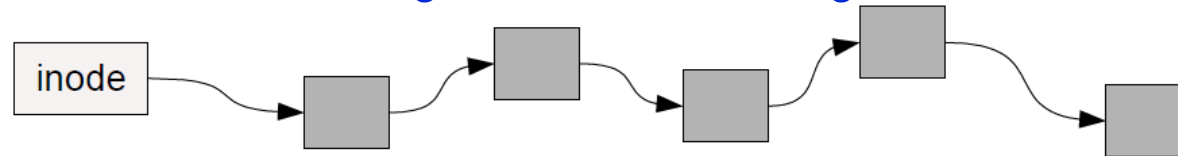
```
bash$ ln /home/foo /tmp/foo
```

```
bash$ ls -i /home/foo /tmp/foo
287663 /home/foo
287663 /tmp/foo
```

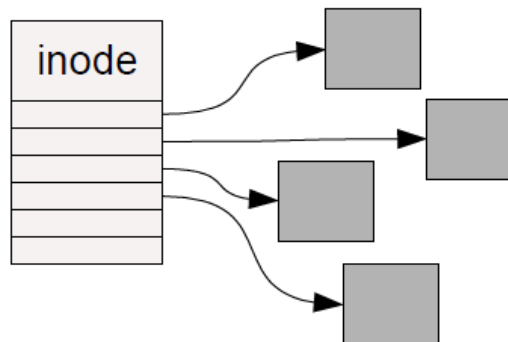
- “/home/foo” and “/tmp/foo” now refer to the same file on disk
 - Not a copy! You will always see identical data no matter which filename you use to read or write the file.
- Note: This is not the same as a “symbolic link”, which only links one filename to another.

How should we organize blocks on a disk?

- Very simple policy: A file consists of linked blocks
 - inode points to the first block of the file
 - Each block points to the next block in the file (just a linked list on disk)
 - What are the advantages and disadvantages??

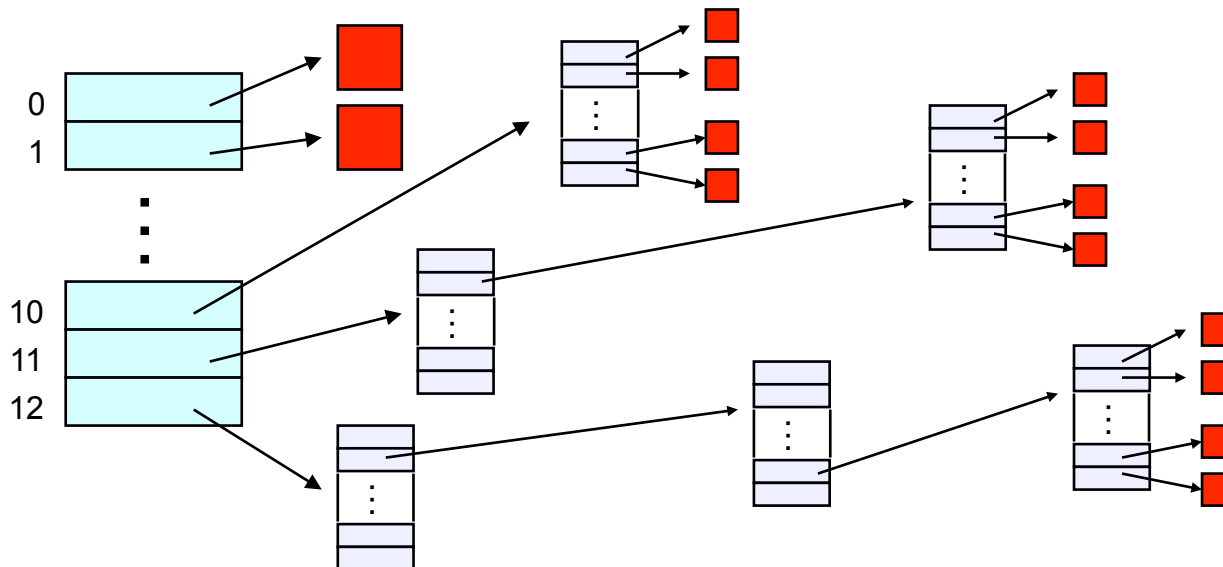


- Indexed files
 - inode contains a list of block numbers containing the file
 - Array is allocated when the file is created
 - What are the advantages and disadvantages??



The “block list” portion of the inode (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 ...

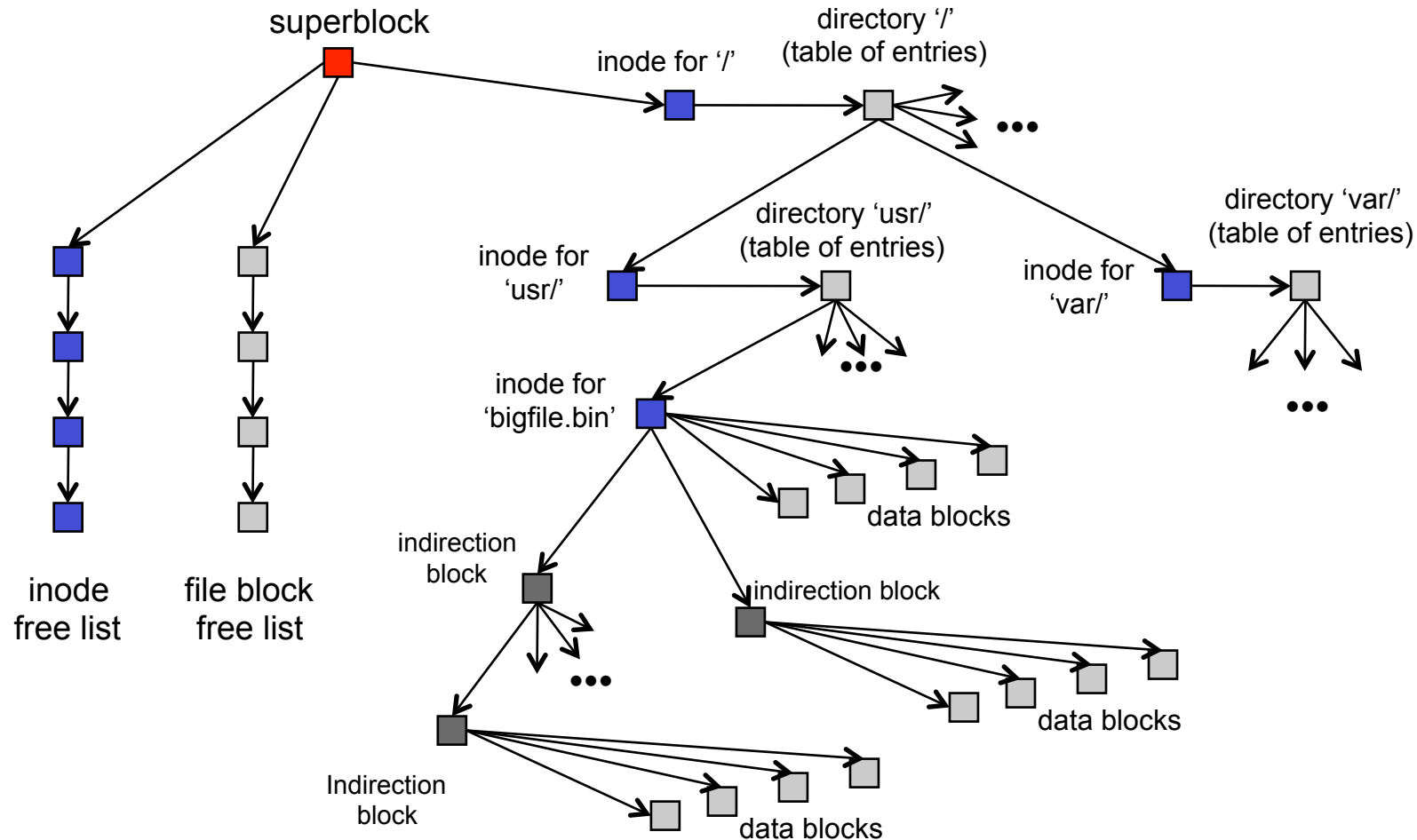
- Only occupies $13 \times 4\text{B}$ in the inode
- Can get to $10 \times 512\text{B} = \text{a } 5120\text{B}$ file directly
 - (10 direct pointers, blocks in the file contents area are 512B)
- Can get to $128 \times 512\text{B} = \text{an additional } 65\text{KB}$ with a single indirect reference
 - (the 11th pointer in the inode gets you to a 512B block in the file contents area that contains 128 4B pointers to blocks holding file data)
- Can get to $128 \times 128 \times 512\text{B} = \text{an additional } 8\text{MB}$ with a double indirect reference
 - (the 12th pointer in the inode 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 inode 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
- Berkeley Unix went to 1KB block sizes
 - What's the effect on the maximum file size?
 - $256 \times 256 \times 256 \times 1K = 17 \text{ GB} + \text{a smidge}$
 - What's the price?
- Subsequently went to 4KB blocks
 - $1K \times 1K \times 1K \times 4K = 4TB + \text{a smidge}$

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