

CSE 120

Principles of Operating Systems

Fall 2025

Lecture 13: File System Implementation

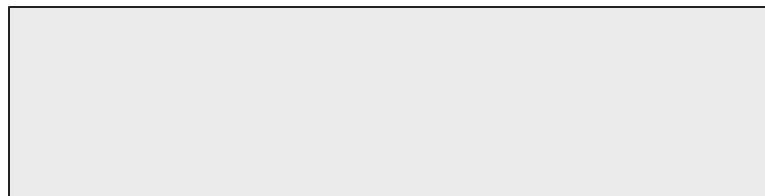
Geoffrey M. Voelker

File Systems

- First we'll discuss properties of physical disks
 - ◆ Structure
 - ◆ Performance
- Then how file systems support users and programs
 - ◆ Files + Directories
 - ◆ Sharing
 - ◆ File buffer cache
- Then how file systems are implemented
 - ◆ File system data structures and layouts
 - ◆ Name resolution
- End with protection

File System Layout

- We start with an empty disk



- Goal for the file system is to manage the disk space to implement the file and directory abstractions that are so convenient for programs and users

Key Questions

- How do we keep track of blocks used by a file?
- Where do we store metadata information?
- How do we (really) do path name translation?
- How do we implement common file operations?

- Our discussion will be Unix-oriented
 - ◆ Other file systems face same challenges, with analogous approaches and data structures for solving them

File System Layout

How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
 - ◆ Disk space is allocated in granularity of blocks
- A **superblock** determines location of root directory
 - ◆ Always at a well-known disk location
 - ◆ Often replicated across disk for reliability
- A **free map** determines which blocks are free, allocated
 - ◆ Usually a **bitmap**, one bit per block on the disk
 - ◆ Also stored on disk, **cached in memory for performance**
- Remaining disk blocks used to store files (and dirs)
 - ◆ There are many ways to do this

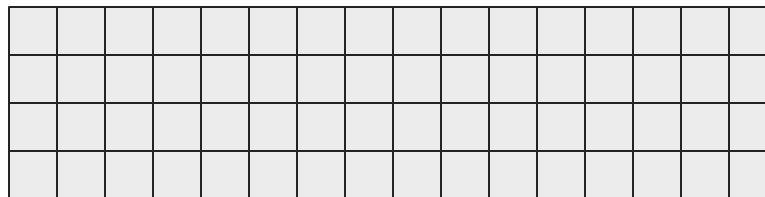
File System Layout

How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
 - ◆ Disk space is allocated in granularity of blocks
- A “superblock” determines location of root directory
 - ◆ Always at a well-known disk location
 - ◆ Often replicated across disk for reliability
- A free map determines which blocks are free, allocated
 - ◆ Usually a bitmap, one bit per block on the disk
 - ◆ Also stored on disk, cached in memory for performance
- Remaining disk blocks used to store files (and dirs)
 - ◆ There are many ways to do this

File System Layout

- Partition it into fixed-size file system blocks



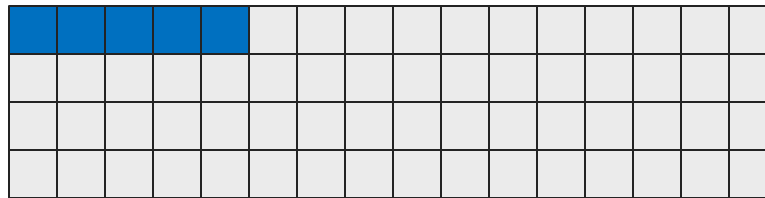
- Typically 4KB in size
 - ◆ Block size set when file system is formatted
- Independent of disk physical sector size
 - ◆ If sector is 512 bytes, file system will use 8 sectors/block

File System Layout

- Files span multiple disk blocks
 - ♦ $2\text{MB file uses } 2 \cdot 1024 \cdot 1024 / 4096 = 512 \text{ blocks (4KB block size)}$
- A small file still uses an entire block
 - ♦ A file of size 4001 bytes uses one block
 - ♦ What kind of fragmentation is this, internal or external?
- Challenge: How do we keep track of all blocks used by one file?

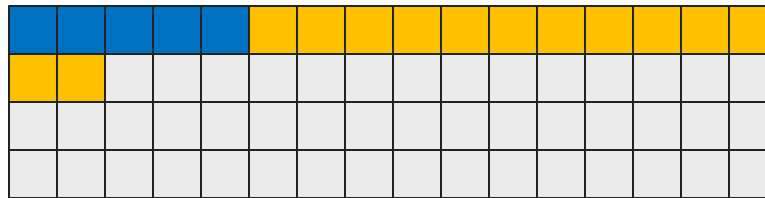
Contiguous Layout

- Can layout file blocks contiguously



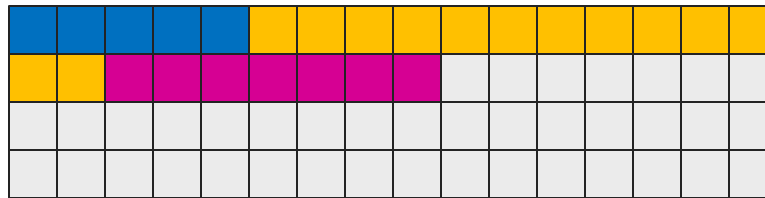
Contiguous Layout

- Can layout file blocks contiguously



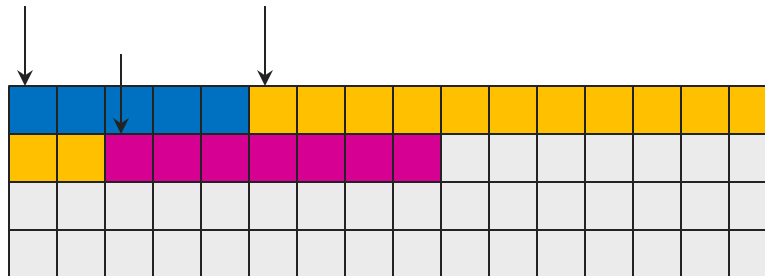
Contiguous Layout

- Can layout file blocks contiguously



Contiguous Layout

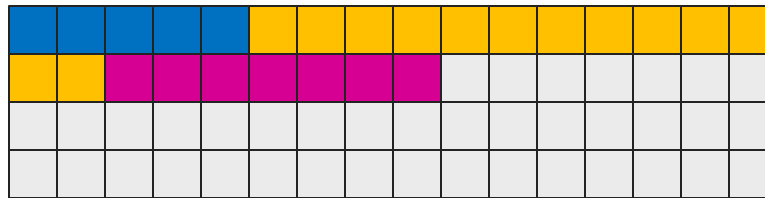
- Simple to keep track of where a file's blocks are



- Directory stores a pointer to the first block
 - ◆ All others are a simple offset from the first
 - ◆ Makes random access also straightforward
- Enables fast sequential access to disk for reads/writes
- But there are multiple disadvantages

Contiguous Layout

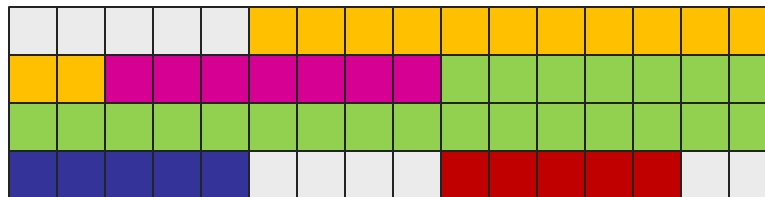
- Difficult to grow a file once it has been written



- If the blue or orange files need to grow, we're stuck

Contiguous Layout

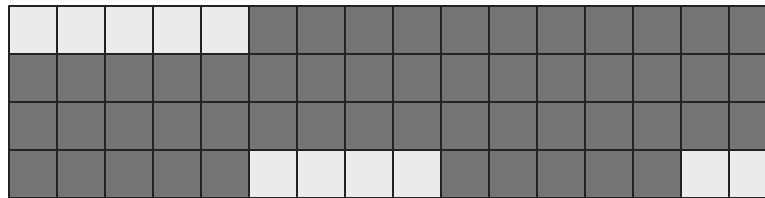
- As files are created and deleted, gaps will occur



- If we need to store a file using 8 blocks, we're stuck
 - ♦ What kind of fragmentation is this, internal or external?
 - ♦ What would be one method for rearranging?

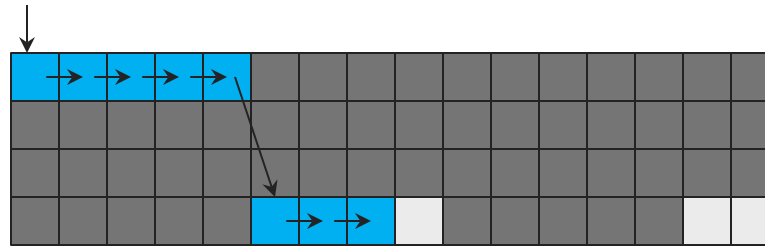
Linked Layout

- Need to store a file with 8 blocks into the “gaps”



Linked Layout

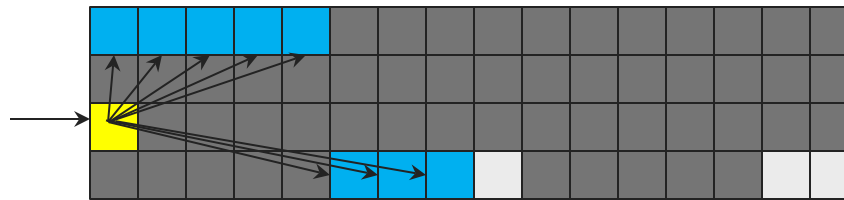
- Another option is to link each block to the next
 - ◆ Essentially a linked list on disk for each file



- Directory still just stores pointer to first block of file
- Fragmentation no longer a problem, can fill in gaps
- Random access now expensive
 - ◆ Need to traverse pointers to access a random block
 - ◆ Potentially many disk reads just to get to desired block

Indexed Layout

- Indexed layouts use a special block (index block) ■ to store pointers to the data blocks ■



- Directory points to the index block
- Still solves fragmentation problem (can fill in gaps)
- Also solves random access problem
 - ◆ After reading the index block, know the locations of all blocks
- For large files, need multiple index blocks

Disk Layout Summary

- Files span multiple disk blocks
- How do you find all of the blocks for a file?
 1. **Contiguous allocation**
 - » Like memory
 - » Fast, simplifies directory access
 - » Inflexible, causes fragmentation, needs compaction
 2. **Linked structure**
 - » Each block points to the next, directory points to the first
 - » Good for sequential access, bad for all others
 3. **Indexed structure (indirection, hierarchy)**
 - » An “index block” contains pointers to many other blocks
 - » Handles random better, still good for sequential
 - » May need multiple index blocks (linked together)

File Metadata

- Unix inodes are special index blocks that also store all metadata for a file
 - ◆ File size
 - » In bytes (actual file size) and blocks (data blocks allocated)
 - ◆ User & group of file owner
 - ◆ Protection bits
 - » user/group/other, read/write/execute
 - ◆ Reference count
 - » How many directory entries point to this inode
 - ◆ Timestamps
 - » Created, modified, last accessed, any change
- “ls -l” reads this info from the inode (**syscall: stat**)

ls -l /bin/

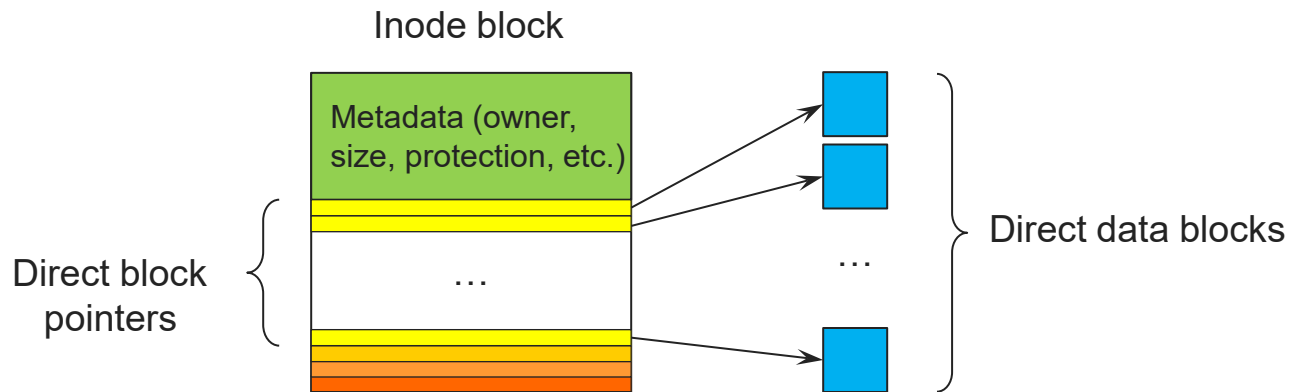
```
-rwxr-xr-x. 1 root root 428584 Jun 28 2017 gawk
-rwxr-xr-x. 1 root root 236504 Nov 2 2018 gcalccmd
-rwxr-xr-x. 2 root root 768608 Sep 29 2020 gcc
-rwxr-xr-x. 1 root root 27088 Sep 29 2020 gcc-ar
-rwxr-xr-x. 1 root root 27088 Sep 29 2020 gcc-nm
-rwxr-xr-x. 1 root root 27088 Sep 29 2020 gcc-ranlib
-rwxr-xr-x. 1 root root 173264 Nov 2 2018 gcm-calibrate
-rwxr-xr-x. 1 root root 65440 Nov 2 2018 gcm-import
-rwxr-xr-x. 1 root root 65416 Nov 2 2018 gcm-inspect
-rwxr-xr-x. 1 root root 82160 Nov 2 2018 gcm-picker
-rwxr-xr-x. 1 root root 107440 Nov 2 2018 gcm-viewer
-rwxr-xr-x. 1 root root 58216 Jun 9 2014 gconf-merge-tree
-rwxr-xr-x. 1 root root 62008 Jun 9 2014 gconftool-2
-rwxr-xr-x. 1 root root 2175 Sep 30 2020 gcore
-rwxr-xr-x. 1 root root 314832 Sep 29 2020 gcov
-rwxr-xr-x. 1 root root 11664 Oct 30 2018 gcr-viewer
-rwxr-xr-x. 1 root root 103640 Apr 3 2020 gctags
-rwxr-xr-x. 1 root root 6826488 Sep 30 2020 gdb
-rwxr-xr-x. 1 root root 1118 Sep 30 2020 gdb-add-index
-rwxr-xr-x. 1 root root 41136 Jun 9 2021 gdbus
-rwxr-xr-x. 1 root root 2050 Jun 9 2021 gdbus_codegen
-rwxr-xr-x. 1 root root 11504 Oct 30 2018 gdk-pixbuf-csource
```

Metadata from the inode
(using stat system call)

Name from the directory

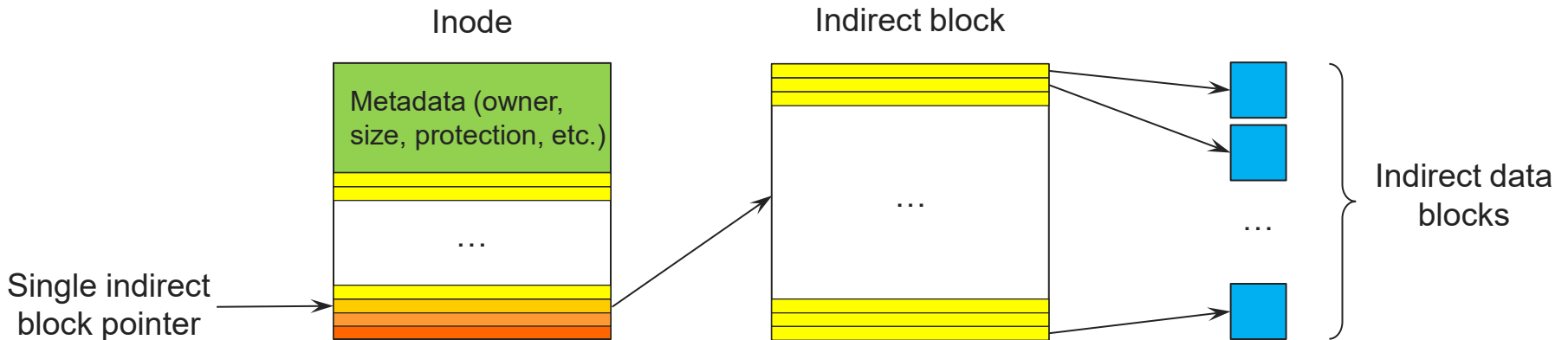
Unix Inodes

- Unix inodes use an “unbalanced” indexed structure
 - ◆ Each inode contains ~15 block pointers
 - ◆ First 12 are direct blocks (convenient for small files)
 - ◆ Then single, double, and triple indirect (for large files)



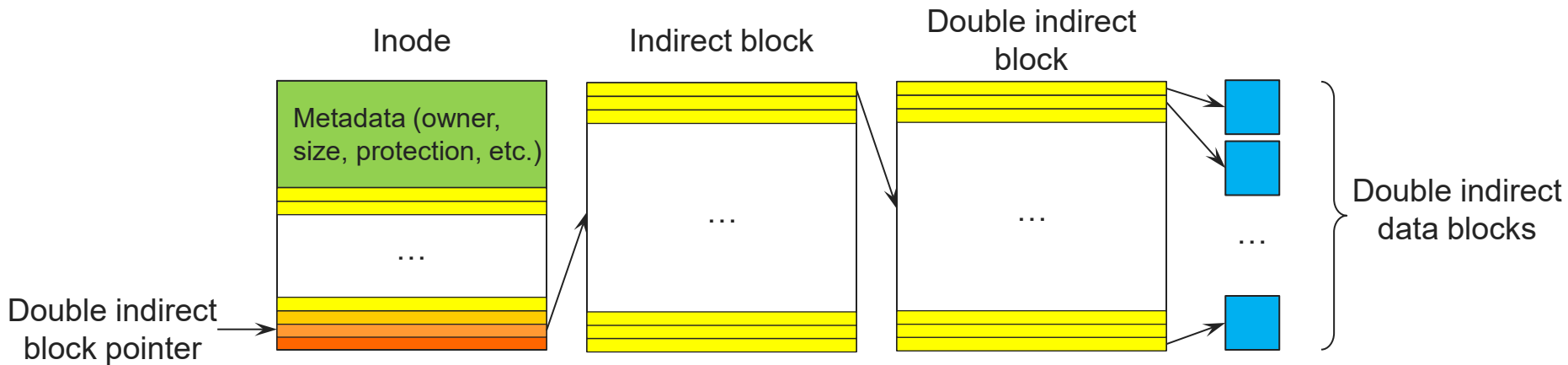
Unix Inodes

- Unix inodes implement an indexed structure for files
 - ◆ Each inode contains ~15 block pointers
 - ◆ First 12 are direct blocks (convenient for small files)
 - ◆ Then single, double, and triple indirect (for large files)



Unix Inodes

- Unix inodes implement an indexed structure for files
 - ◆ Each inode contains ~15 block pointers
 - ◆ First 12 are direct blocks (convenient for small files)
 - ◆ Then single, double, and triple indirect (for large files)



- ◆ ... and so on with triple indirect.
- ◆ Indirect blocks are full blocks (4K), inodes are small (256 bytes)
 - » Accessing one inode block accesses multiple inodes

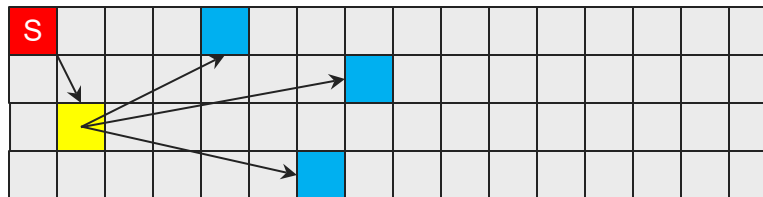
File System Layout

How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
 - ◆ Disk space is allocated in granularity of blocks
- A **superblock** determines location of root directory
 - ◆ Always at a well-known disk location
 - ◆ Often replicated across disk for reliability
- A free map determines which blocks are free, allocated
 - ◆ Usually a bitmap, one bit per block on the disk
 - ◆ Also stored on disk, cached in memory for performance
- Remaining disk blocks used to store files (and dirs)
 - ◆ There are many ways to do this

Superblock

- “/” is the directory that is the root of the file system
- How do we find the inode for “/”?



- The superblock stores a pointer to the inode of “/”
 - ◆ The inode for “/” has pointers to all of the blocks storing the directory entries for “/”
- It is the basis for translating all path names
- It is at a fixed, pre-defined location on disk
 - ◆ Replicated deterministically across the FS for redundancy

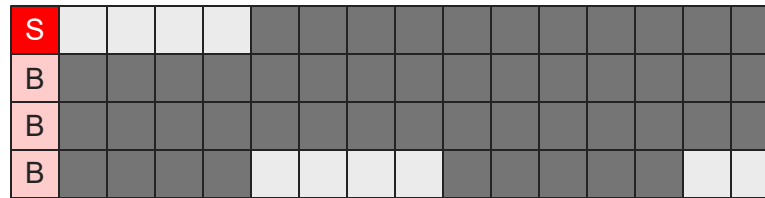
File System Layout

How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
 - ◆ Disk space is allocated in granularity of blocks
- A “superblock” determines location of root directory
 - ◆ Always at a well-known disk location
 - ◆ Often replicated across disk for reliability
- A **free map** determines which blocks are free, allocated
 - ◆ Usually a bitmap, one bit per block on the disk
 - ◆ Also stored on disk, **cached in memory for performance**
- Remaining disk blocks used to store files (and dirs)
 - ◆ There are many ways to do this

Block Allocation

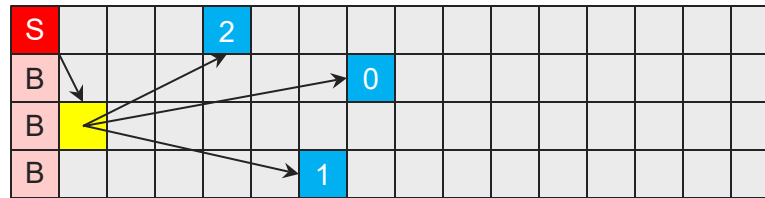
- The file system needs to keep track of which blocks have been allocated and which are free



- Free map blocks **B** store a bitmap, one bit per block
 - ◆ Bit is set → block is allocated
 - ◆ One bitmap for data blocks
 - ◆ Another for inode blocks

Types of Blocks on Disk

- Four basic kinds of blocks on disk
 - ◆ Only data blocks store file data and directory data



- Superblock S
- Bitmap blocks B
- Inode blocks
- Data blocks 0 1 2

Unix Inodes != Directories

- Unix inodes are **not** directories
 - ◆ Inodes describe where on the disk the blocks of a file are
- Directories are files, so inodes also describe where the blocks for directories are placed on the disk
 - ◆ Every directory and file on disk has its own inode

Path Name Translation (v2)

- Directory entries map file names to inodes
 - ◆ To open “/one”, use superblock to find inode for “/” on disk
 - ◆ Open “/”, look for entry for “one”
 - ◆ This entry gives the disk block number for the inode for “one”
 - ◆ Read the inode for “one” into memory
 - ◆ The inode says where first data block is on disk
 - ◆ Read that block into memory to access the data in the file

Symbolic (Soft) Links

- It is convenient to be able to create **aliases** in the FS
 - ◆ Have multiple names refer to the same file
- Soft links are the most familiar form in Unix
 - \$ `ln -s file alias` (`ln -s /a/b/c /tmp/softlink`)
 - ◆ **Syscall: `symlink`**
- Soft links create aliases via path name translation
 - ◆ Path name translation starts again when hitting a soft link
 - ◆ `/tmp/softlink` → `/a/b/c`
- Implemented by **storing the alias as a string in a file** and flagging the inode as a soft link
 - ◆ FS reads the path alias from the file and restarts translation

Hard Links

- Hard links are another form of aliasing
 - ◆ *\$ ln file alias (ln /a/b/c /tmp/hardlink)*
 - ◆ **Syscall: link**
- Hard links create aliases via inode pointers in dirs
 - ◆ Recall that a directory entry maps a name to an inode
 - ◆ Creating a hard link adds another directory entry mapping the new name to **the same inode** as the old name
 - ◆ It adds a new pointer, or link, to the inode
 - ◆ Reference count in the inode is also incremented
- The “.” and “..” names are hard links to directories
 - ◆ /a/b/c and /a/b/c/. point to the same inode

Create

- Creating a file “new” is relatively straightforward
- **Allocate an inode**
 - ◆ Initialize the metadata (owner, protection, timestamps, ...)
 - ◆ Update inode bitmap
- **Allocate a directory entry** in the directory for the file
 - ◆ Entry maps “new” to the inode allocated for “new”
- When process starts writing to file, **allocate data blocks**
 - ◆ Update inode to point to data blocks allocated
 - ◆ Update data block bitmap
 - ◆ Continue to allocate data blocks on demand
 - » Preallocating blocks in “extents” helps keep blocks contiguous

Rename

- One way to rename a file is to simply create a new one with the new name, copy the contents, and delete the old file
 - ♦ Method used in original version of Unix (test/mv.c in Nachos)
- More efficient + reliable to implement in FS
 - \$ mv old new
 - ♦ Syscall: **rename**
- Rename creates a new directory entry with the **new name that points (links) to the same inode** as the old
 - ♦ Then it deletes the entry directory for the old name
 - ♦ **Only directories are modified**, file and inode stay the same

Delete

- Deleting a file has a few steps
 - ◆ Remove the directory entry for the name being deleted
 - » Hence the syscall name **unlink**
 - ◆ Decrement the reference count in the inode
 - ◆ If the file still has links to it, nothing else happens
- If there are no remaining links
 - ◆ Free up the data blocks (update the data block bitmap)
 - ◆ Free up the inode (update the inode bitmap)
 - ◆ Block data is not erased
- If the file is still open in any process, the **directory entry is removed** but the **file blocks are not**
 - ◆ Until the last process with the file open finally closes it

Partitions

- What if we want multiple file systems on one disk?



Partitions

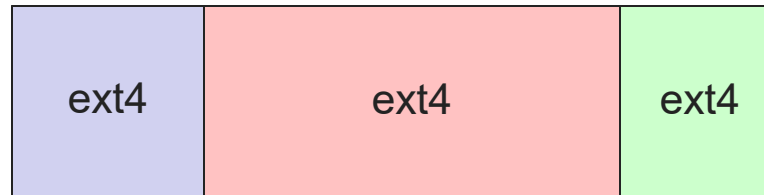
- What if we want multiple file systems on one disk?



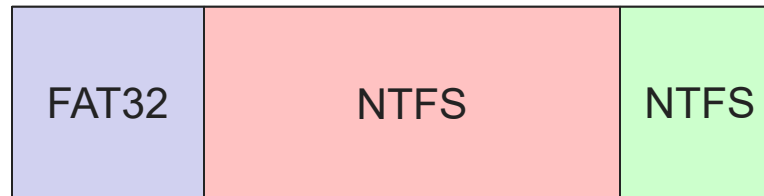
- Split up physical disk into multiple **partitions**
- Each partition has an entire file system inside of it
 - ♦ Superblock, bitmaps, inodes, data blocks, etc.

Partitions

- The partitions could have the same kind of file system



- Different kinds supported by the same OS



- Different kinds supported by different OSes (dual boot)



Partitions

- What if we want multiple file systems on one disk?



- Split up physical disk into multiple **partitions**
- Each partition has an entire file system inside of it
 - ♦ Superblock, bitmaps, etc.
- **How do we link them together into one name space?**

Mounting File Systems

- Mounting is the mechanism used to piece together multiple file systems into a single global name space
- One file system is mounted as “/” (root)
- Other file systems attached at **mount points**
 - ◆ An empty directory in file system, e.g., /home
 - ◆ Mounting the “home” file system attaches the root for “home” to /home in the name space
 - ◆ Opening “/home/user/file” starts path name translation in the “root” file system, continues in the “home” file system when crossing the mount point
- Mostly invisible to users and processes
 - ◆ Some exceptions (e.g., cannot hard link across file systems)

Next time...

- Read Chapters 53, 55