CS5460: Operating Systems

Lecture 18: File System Implementation
(Ch.10)

THE UNIVERSITY OF UTAH

Important From Last Time

- Disks
  - Appear as a big pile of fixed-size blocks
  - R/W operations are expensive (still true, but less so, for SSD)
- Filesystem goals
  - Persistence
  - Speed – sequential and random access
  - Size
  - Sharing vs. protection – implement OS security policy
  - Ease of use – abstractions are convenient to use
- Modern filesystems:
  - Hierarchical directory namespace
  - Files are extensible collections of bytes

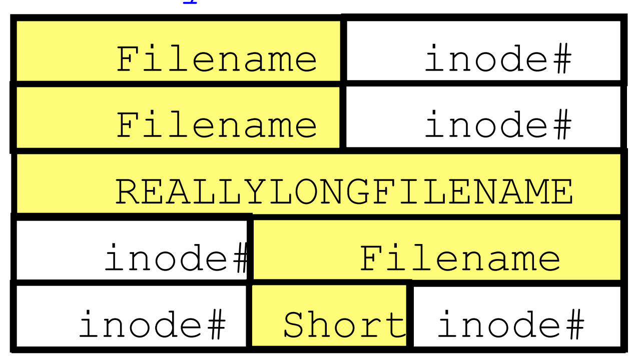
Key On-Disk Data Structures

- **File descriptor (aka "inode")**
  - Link count
  - Security attributes: UID, GID, …
  - Size
  - Access/modified times
  - "Pointers" to blocks
  - …
- **Directory file: array of…**
  - File name (fixed/variable size)
  - Inode number
  - Length of directory entry
- ***Free block bitmap***
- ***Free inode bitmap***
- ***Superblock***

File descriptor (inode):

```
ulong links;
uid_t uid;
gid_t gid;
ulong size;
time_t access_time;
time_t modified_time;
addr_t blocklist…;
```

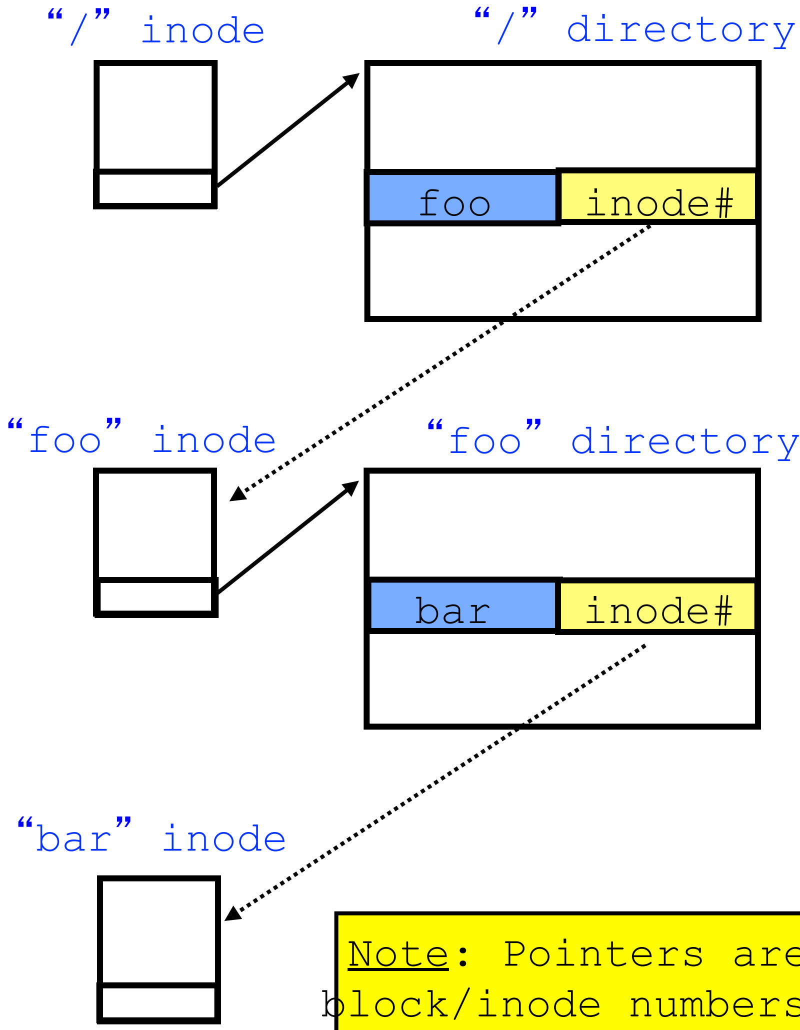
Directory file:



Finding a File's Inode on Disk

Locate inode for `/foo/bar`:

1. Find inode for "/"
   – Always in known location
2. Read "/" directory into memory
3. Find "`foo`" entry
   » If no match, fail lookup
4. Load "foo" inode from disk
5. Check permissions
   » If no permission, fail lookup
6. Load "foo" directory blocks
7. Find "bar" entry
   » If no match, fail lookup
8. Load "bar" inode from disk
9. Check permissions
   » If no permission, fail lookup



Note: Pointers are block/inode numbers, not addresses!

Finding a File's Blocks on Disk

- Conceptually, inode contains table:
  - One entry per block in file
  - Entry contains physical block address (e.g., platter 3, cylinder 1, sector 26)
  - To locate data at offset X, read block (X / block_size)

Block Address 0
Block Address 1
…
Block Address N

- Issues → How do we physically implement this table?
  - Most files are small
  - Most of the disk is contained in (relatively few) large files
  - Need to efficiently support both sequential and random access
  - Want simple inode lookup and management mechanisms

Allocating Blocks to Files

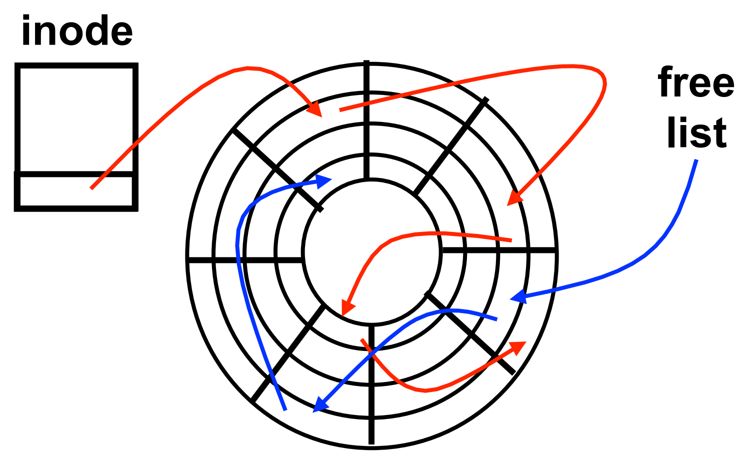
- Contiguous allocation
  - Files allocated (only) in contiguous blocks on disk
  - Analogous to base-and-bounds memory management
- Linked file allocation
  - Maintain a linked list of blocks used to contain file
  - At end of each block, add a (hidden) pointer to the next block
- Indexed file allocation
  - Maintain array of block numbers in inode
- Multi-level indexed file allocation
  - Maintain array of block numbers in inode
  - Maintain pointers to blocks full of more block numbers in inode (indirect blocks, double-indirect blocks, …)

Contiguous

- Files allocated in contiguous blocks on disk
- Maintain ordered list of free blocks
  - At create time, find large enough contiguous region to hold file
- Inode contains **START** and **SIZE**
- Advantages:
  - Very simple to implement
  - Easy *offset→block* computation for sequential or random access
  - Few seeks
- Disadvantages:
  - Fragmentation → analogous to base and bounds
  - How do we handle file growth/shrinkage?
- Question: When might this work well?

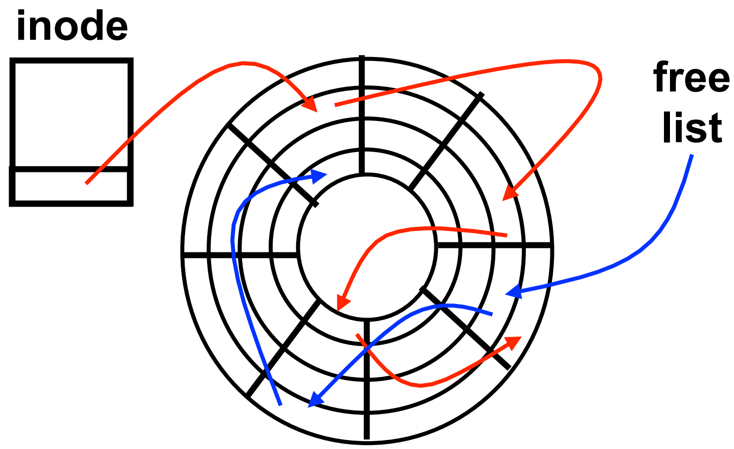
Linked File Allocation

- Linked list of free blocks
  - Allocate any free block
- At end of each block, reserve space for block#
- Inode contains START
- What are good/bad points of this scheme?



Linked File Allocation

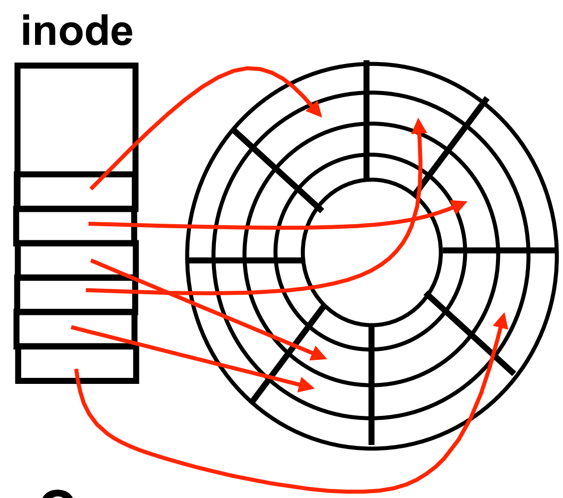
- Linked list of free blocks
  - Allocate any free block
- At end of each block, reserve space for block#
- Inode contains START



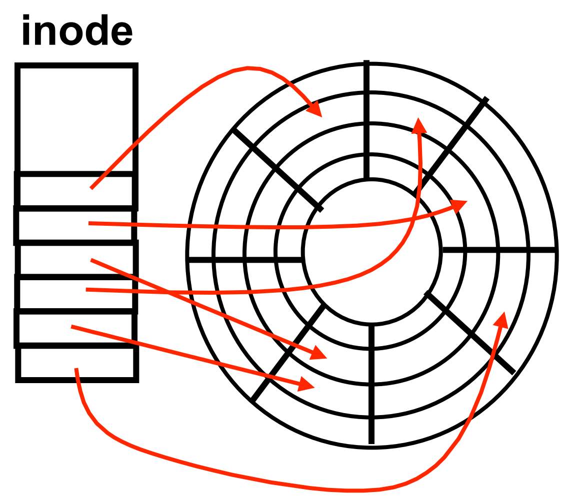
- Good points
  - Can extend/shrink files easily → no fragmentation
  - Handles sequential accesses somewhat efficiently
  - Efficient inode encoding (small, constant)
- Bad points
  - Random access of large files is really inefficient!
  - Lots of seeks → non-contiguous blocks

Indexed File Allocation

- Inode contains array of block addresses
  - Allocate table at file create time
  - Fill entries as blocks allocated
- Separate free block bitmap
- What are good and bad points?

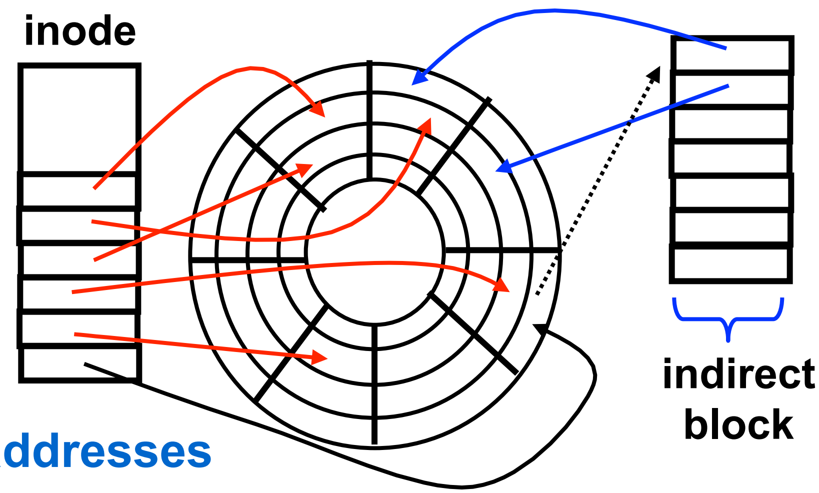


Indexed File Allocation



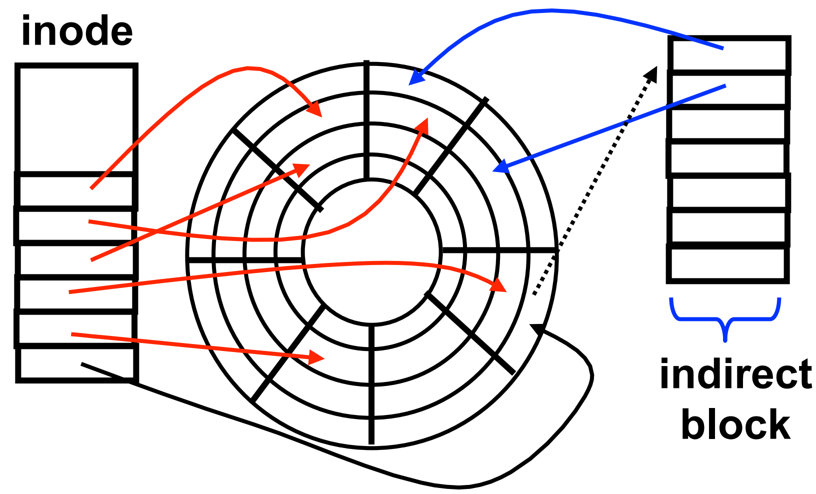
- **Inode contains array of block addresses**
  - Allocate table at file create time
  - Fill entries as blocks allocated
- **Separate free block bitmap**
- **Good points**
  - Can extend/shrink files… to a point
  - Simple *offset→block* computation for sequential or random access
- **Bad points**
  - Need to pre-declare maximum size of file
  - Variable sized inode structures
  - Lots of seeks → non-contiguous blocks

Multi-level Indexed File Allocation



- **Inode contains:**
  - Fixed-size array of *direct blocks*
  - Small array of *indirect blocks*
  - (Optional) double/triple indirect
- **Indirection:**
  - Indirect block: Block full of block addresses
  - Double indirect block: Block full of indirect block addresses
- **What are good and bad points of this scheme?**

Multi-level Indexed File Allocation



- **Inode contains:**
  - Fixed-size array of *direct blocks*
  - Small array of *indirect blocks*
  - (Optional) double/triple indirect
- **Good points**
  - Simple *offset→block* computation for sequential or random access
  - Allows incremental growth/shrinkage
  - Fixed size (small) inodes
  - Very fast access to (common) small files
- **Bad points**
  - Indirection adds overhead to random access to large files
  - Blocks can be spread all over disk → more seeks

Multi-level Indexed File Allocation

- Example: 4.3 BSD file system
  - Inode contains 12 direct block addresses
  - Inode contains 1 indirect block address
  - Inode contains 1 double-indirect block address
- If block addrs are 4-bytes and blocks are 1024-bytes, what is maximum file size?

Multi-level Indexed File Allocation

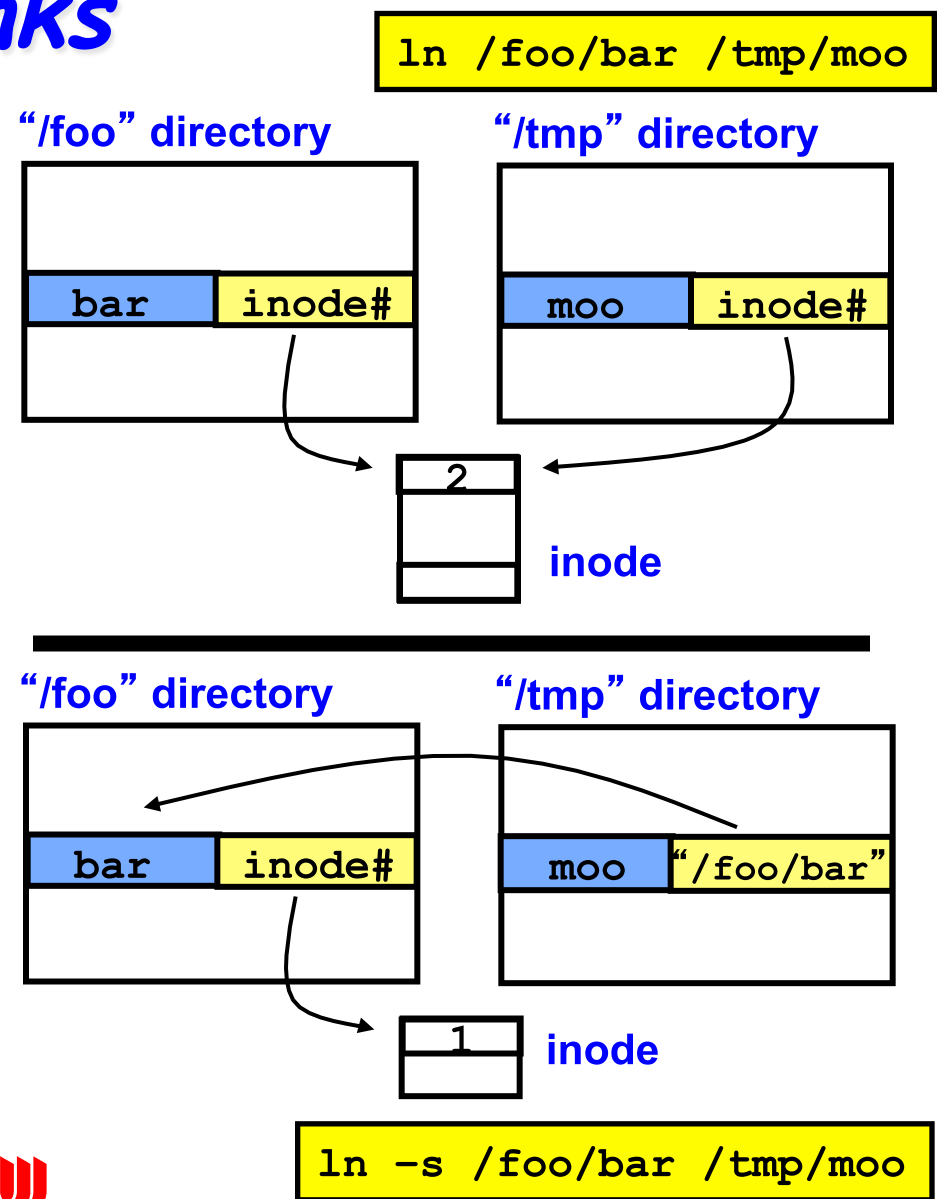
- **Example: 4.3 BSD file system**
  - Inode contains 12 direct block addresses
  - Inode contains 1 indirect block address
  - Inode contains 1 double-indirect block address
- **If block addrs are 4-bytes and blocks are 1024-bytes, what is maximum file size?**
  - Number of block addrs per block = 1024/4 = 256
  - Number of blocks mapped by direct blocks → 12
  - Number of blocks mapped by indirect block → 256
  - Number of blocks mapped by double-indirect block → 256^2 = 65536
  - Max file size: (12 + 256 + 65536) * 1024 = 66MB (67,383,296 bytes)

Extents

- Contiguous allocation is impractical, but real filesystems aim for as much contiguity as possible
- Extents provide good support for the common case where a file is somewhat, but not totally, contiguous
- An extent is a pair:
  - (starting block, length)
- Each file is represented by a list of extents
  - For a given list length, extents can refer to more data than can a block list
  - But still, we'll run out of extents at some point
  - How to support very large files in an extent-based FS?

Links

- Links let us have multiple names to same file
- Hard links:
  - Two entries point to same inode
  - Link count tracks connections
    - » Decrement link count on delete
    - » Only delete file when last connection is deleted
  - Problems: cannot cross filesystems, loops, unreachable directories
- Soft links:
  - Adds symbolic "pointer" to file
  - Special flag in directory entry
  - Only one "real" link to file
    - » File goes away when its deleted
  - Problems: Infinite loops



Important From Today

- **Key filesystem function: Rapidly find the blocks associated with each file**
  - **Support typical distribution of file sizes**
    - » **Most files are small**
    - » **Most bytes are in large files**
  - **Support efficient access**
    - » **Sequential**
    - » **Random**
  - **Avoid wasting much space**