Representing Files in File Systems

COMP 321

Dave Johnson

1

In Addition to the Inode for Each File?

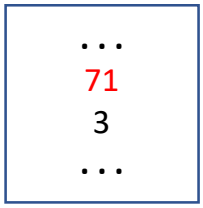
Need to keep track of which blocks store the data of a given file

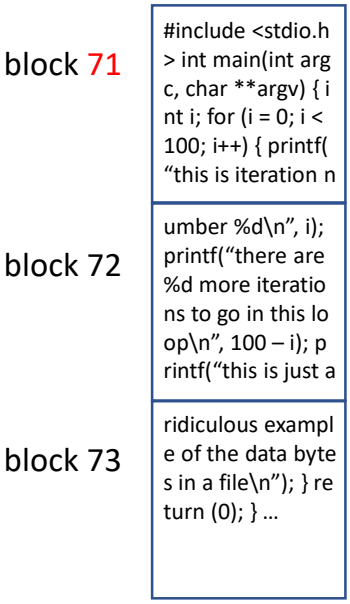
- Want to be able to efficiently read/write the file contents sequentially
- Want to be able to efficiently read/write the file contents in "random" order
- Want to be able to have a large maximum file size
- Want to be able to represent small files efficiently
- Want to be able to represent even very large files efficiently

2

Contiguous Allocation

- All data in the file is stored in a single contiguous collection of data blocks
- In the inode
  - block number of first block of file's contents
  - the number of contiguous blocks allocated for file's contents
- Usually, all blocks are preallocated when the file is created





Contiguous Allocation

Advantages

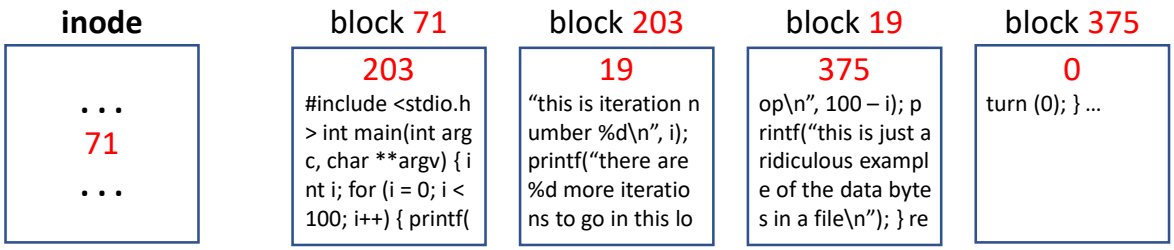
- Very easy to implement
- Efficient sequential access and random access

Problems

- Essentially the same problems as faced by malloc( ) for memory allocation, but disk is much slower (e.g., best fit?)
- Bad external fragmentation problem
- Bad internal fragmentation problem (if preallocated space)
- Often can't (or very inefficient to) grow the file

Linked Allocation

- In inode: the block number of the first block of the contents of the file
- In each data block of the file
  - block number of the next block of the contents of the file (e.g., 4 bytes)
  - the rest of that block is filled with part of the contents of the file



Linked Allocation

Advantages

- Files are easy to grow (except for finding last block to link new block onto)
- No external fragmentation, and internal fragmentation limited to < block size
- Fast sequential file access

Problems

- Inefficient and difficult random access
- Space taken out of each data block to store the next block number of the file (and no longer a power of 2 amount of data in each data block)
- More disk seeking even on sequential file access since not contiguous

Indexed Allocation

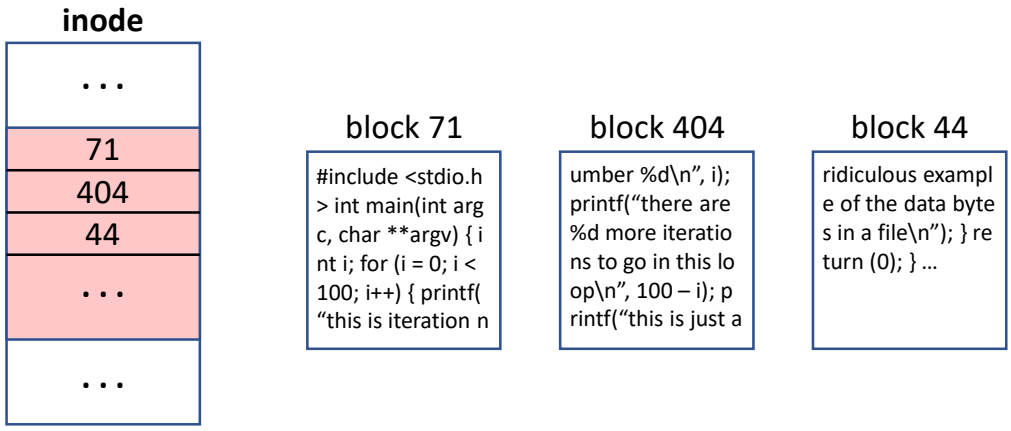
Many alternative variants on this basic idea

- Something abstractly similar to a page table
  - Given the file-relative block number of data to access within that file,
  - You can use that relative block number to go directly to the place in the index that gives the file system block number storing that data
  - The block index, of course, must be stored on the disk in the file system
  - (This is where the Unix file system "inode" gets its name)
- But big questions
  - Where on disk is the index for a given file stored?
  - How big (how many entries) is the index, and how big can the index be?

Indexed Allocation: Index in the Inode

The entire block index is in the inode

- An array listing all corresponding block numbers of the file's contents



Indexed Allocation: Index in the Inode

Advantages

- Can always directly, immediately get the block number for any part of a file

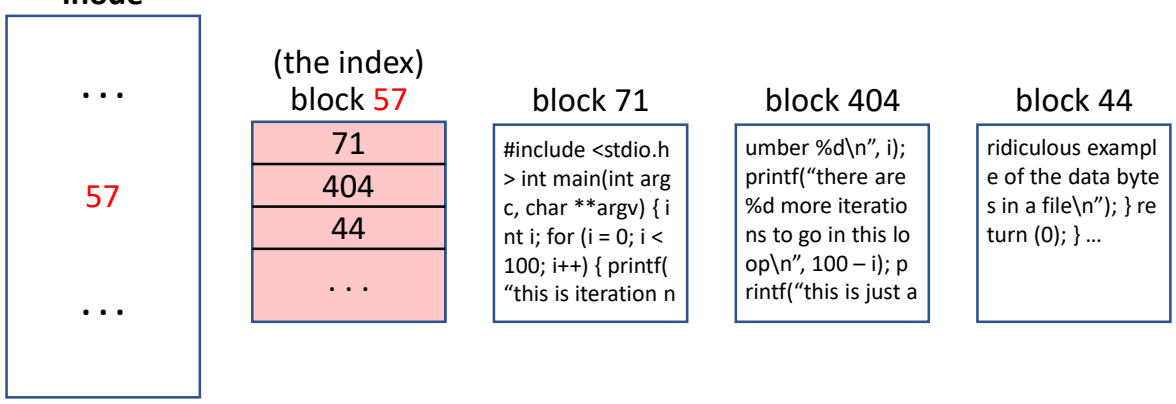
Problems

- Small maximum file size, or
- Large inode size, or
- Variable inode size
  - now hard to find an inode just given its inode number
  - and inodes can now be split across a block boundary
  - or increased internal fragmentation in allocation of space for inodes

Indexed Allocation: Index in a Separate Block

The file's index is in a single separate block outside the inode

- In inode: the block number of that index block



Indexed Allocation: Index in a Separate Block

Advantages

- Only uses space for the index for inodes that are in use, not for all inodes
- Easy to implement and efficient to use

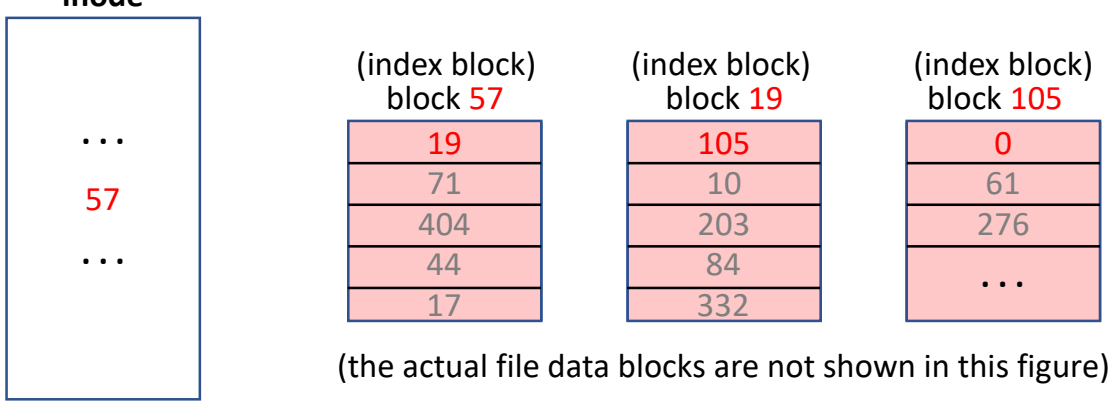
Problems

- Small maximum file size, with only a single block for the entire index
- Could make a file's index multiple blocks instead of a single block
  - Contiguous would be easy to implement and efficient to use, but very hard to manage allocation of the contiguous index blocks
  - Not contiguous requires some way to find the right block of the file's index when accessing some block of the file's data contents

Indexed Allocation: In a Variable List of Blocks

In each index block, give the block number of the next block of the index

- Like basic linked data allocation, but for the index, not for the data



(the actual file data blocks are not shown in this figure)

Indexed Allocation: In a Variable List of Blocks

Advantages

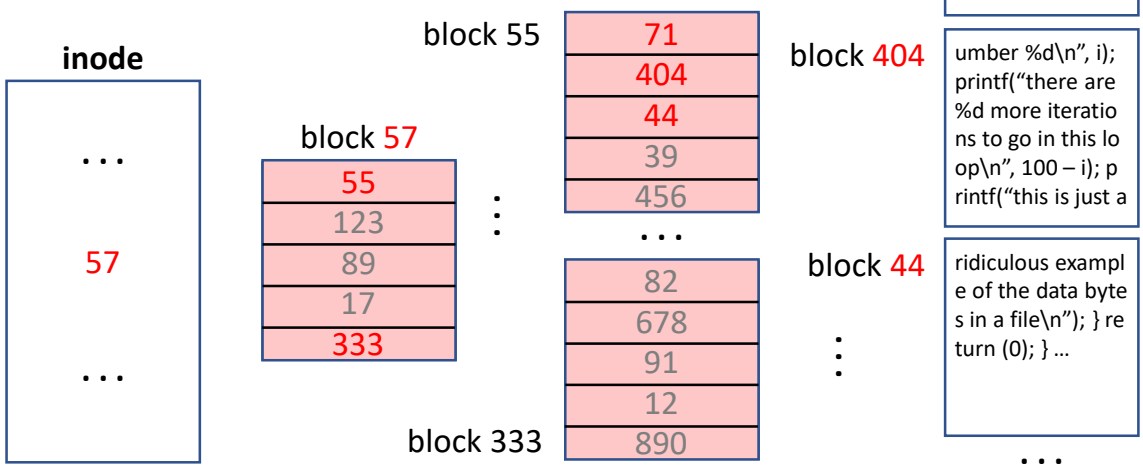
- Still only uses space for the index for inodes that are in use, not for all inodes
- Allows an arbitrarily large maximum file size, since the index can grow

Problems

- Can't go directly to the needed data block, since you have to find the correct block of the index first
- Similar to linked allocation for the data blocks, you have to read each index block just to get the block number of the next block of the index
- And those index blocks could be a long seek from one index block to the next

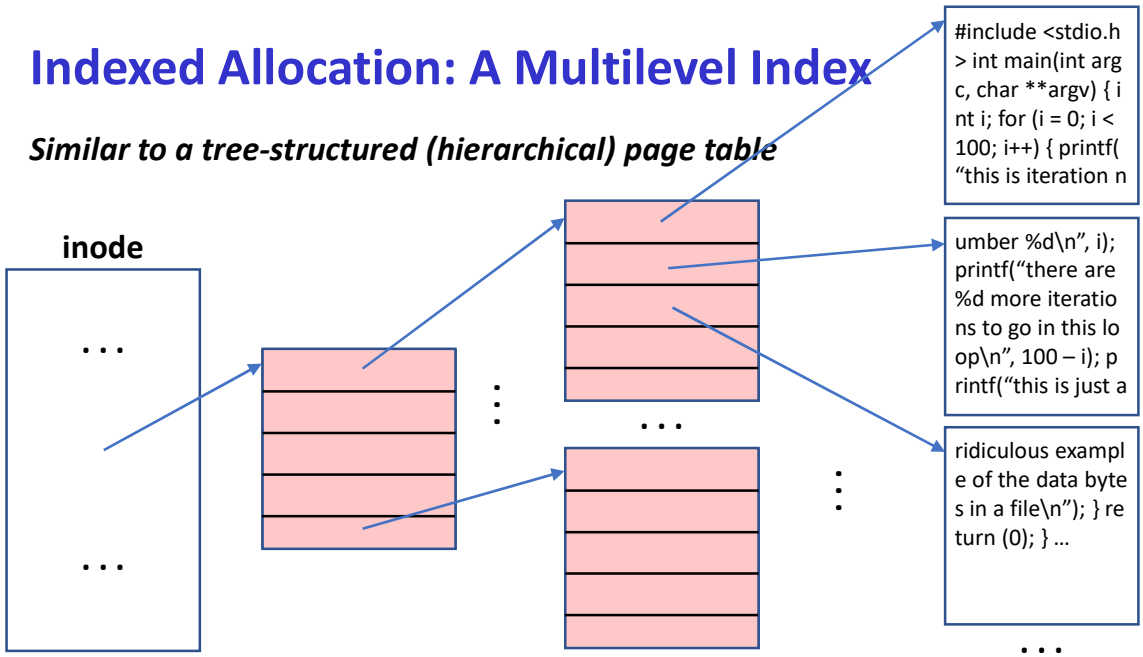
Indexed Allocation: A Multilevel Index

Similar to a tree-structured (hierarchical) page table:



Indexed Allocation: A Multilevel Index

Similar to a tree-structured (hierarchical) page table



Indexed Allocation: A Multilevel Index

Advantages

- Allows an arbitrarily large maximum file size, since the index can grow
- And now you can get to the block number (and thus the data) for any part of the file reasonably efficiently

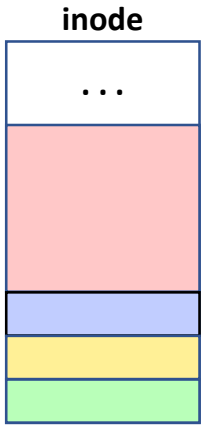
Problems

- How many levels of index hierarchy do we need or want?
  - Example: 4096 block size, 4-byte block numbers = 1024 numbers/block
  - 1-level = max file size 1024 blocks, 2-level = 1024^2, 3-level = 1024^3, …
  - Small number of levels: efficient for small files, but small max file size
  - Large number of levels: wasted index space for small files, ***and*** must go through all levels to access ***any*** part of the file

Indexed Allocation: The Unix Inode Scheme

Small index in inode (0-level), plus 1-level hierarchy, plus 2-level, plus 3-level

- Each level, and index blocks within that level, only allocated as needed

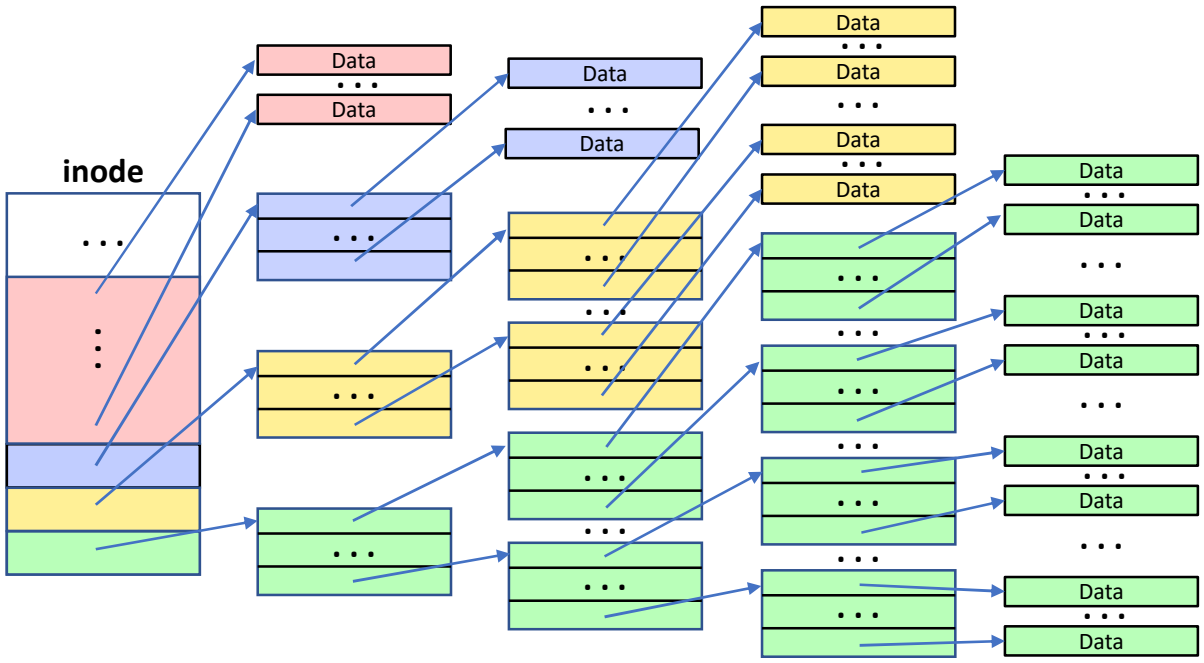


Example: 4096 block size, 4 bytes for each block number

10 ***direct*** block numbers, leading directly to 10 blocks of data

1 ***single indirect*** block number, leading to 1024 blocks of data
1 ***double indirect*** block number, leading to 1024^2 blocks of data
1 ***triple indirect*** block number, leading to 1024^3 blocks of data

17



18

Indexed Allocation: The Unix Inode Scheme

Advantages

- Space and time efficient for accessing any part of a small file
- Space and time efficient for accessing any part of a large file, relative to other ways of representing a large file
- Space and time efficient for accessing the ***beginning*** of a large file

Problems

- A little more complicated than a simple index, but still easily manageable

Separate Question: How Big is the File?

All unused block numbers in an inode and in index blocks "should" be 0

But this is in general only a convention

- And the first 0 block number does ***not*** mark the end of the file
- And what if there are exactly 5000 bytes (not 8192) in the file?
- Instead, ***a field in the inode*** gives the file size in ***bytes***
  - The number of block numbers in use is obtained from ***math*** on that size
  - All block numbers after that point should be ignored
  - And all index blocks after that point should not exist
  - (Returned as the st_size field in a "struct stat" from stat( ) and fstat( ))