File Systems: Fundamentals

Files

- What is a file?
  - A named collection of related information recorded on secondary storage (e.g., disks)
- File attributes
  - Name, type, location, size, protection, creator, creation time, last-modified-time, …
- File operations
  - Create, Open, Read, Write, Seek, Delete, …
- How does the OS allow users to use files?
  - "Open" a file before use
  - OS maintains an open file table per process, a file descriptor is an index into this file.
  - Allow sharing by maintaining a system-wide open file table

Fundamental Ontology of File Systems

- Metadata
  - The index node (inode) is the fundamental data structure
  - The superblock also has important file system metadata, like block size
- Data
  - The contents that users actually care about
- Files
  - Contain data and have metadata like creation time, length, etc.
- Directories
  - Map file names to inode numbers

Basic Data Structures

- Disk
  - An array of sectors, where a sector is a fixed size data array
- File
  - Sequence of blocks (fixed length data array)
- Directory
  - Creates the namespace of files
    - Hierarchical – traditional file names and GUI folders
    - Flat – like the all songs list on an ipod
- Design issues: Representing files, finding file data, finding free blocks

Terms

Field

- basic element of data
- contains a single value
- fixed or variable length

Database

- collection of related data
- relationships among elements of data are explicit
- designed for use by a number of different applications
- consists of one or more types of files

File

- collection of similar records
- treated as a single entity
- may be referenced by name
- access control restrictions usually apply at the file level

Record

- collection of related fields that can be treated as a unit by some application program
- fixed or variable length

Blocks and Sectors

- Recall: Disks write data in units of **sectors**
  - Historically 512 Bytes; Today mostly 4KiB
  - A sector write is all-or-nothing
- File systems allocate space to files in units of **blocks**
  - A block is 1+ **consecutive** sectors

Selecting a Block Size

- Convenient to have blocks match or be a multiple of page size (why?)
  - Cache space in memory can be managed with same page allocator as used for processes; mmap of a block to a virtual page is 1:1
- Large blocks can be more efficient for large read/writes (why?)
  - Fewer seeks per byte read/written (if all of the data useful)
- Large blocks can *amplify* small writes (why?)
  - One byte update may cause entire block to be rewritten

Functionality and Implementation

- File system functionality:
  - Allocate physical sectors for logical file blocks
    - Must balance locality with expandability.
    - Must manage free space.
  - Index file data, such as a hierarchical name space
- File system implementation:
  - File header (descriptor, inode): owner id, size, last modified time, and location of all data blocks.
    - OS should be able to find metadata block number N without a disk access (e.g., by using math or cached data structure).
  - Data blocks.
    - Directory data blocks (human readable names)
    - File data blocks (data).
  - Superblocks, group descriptors, other metadata…

File System Properties

- Most files are small.
  - Need efficient support for small files.
  - Block size can't be too big.
- Some files are very large.
  - Must allow large files (64-bit file offsets).
  - Large file access also should be reasonably efficient.

Three Problems for Today

- Indexing data blocks in a file:
  - What is the LBA of is block 17 of The_Dark_Knight.mp4?
- Allocating free disk sectors:
  - I add a block to fine-lru.c, where should it go on disk?
- Indexing file names:
  - I want to open /home/porter/foo.txt, does it exist, and where on disk is the metadata?

Problem 0: Indexing Files&Data

The information that we need:

For each file, a file header points to data blocks

Block 0 --> Disk sector 19

Block 1 --> Disk sector 4,528

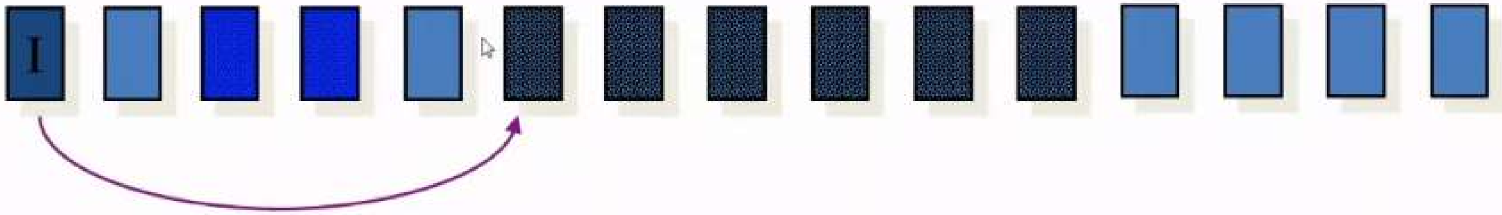
...

Key performance issues:

1. We need to support sequential and random access.
2. What is the right data structure in which to maintain file location information?
3. How do we lay out the files on the physical disk?

We will look at some data indexing strategies

Strategy 0: Contiguous Allocation

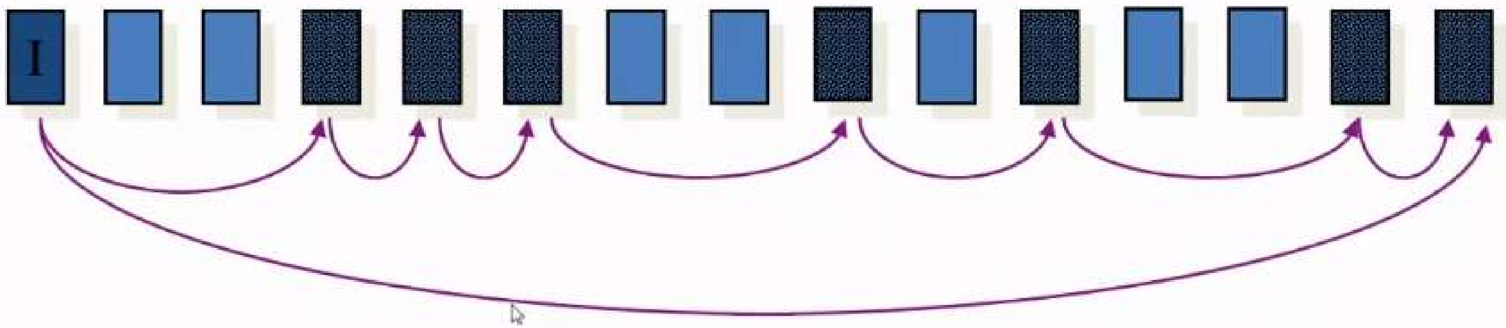


- File header specifies starting block & length
- Placement/Allocation policies
  - First-fit, best-fit, …

u Pluses
- ➢ Best file read performance
- ➢ Efficient sequential & random access

u Minuses
- ➢ Fragmentation!
- ➢ Problems with file growth
  - Pre-allocation?
  - On-demand allocation?

Strategy 1: Linked Allocation



- Files stored as a linked list of blocks
- File header contains a pointer to the first and last file blocks

- Pluses
  - Easy to create, grow & shrink files
  - No external fragmentation
    - Can "stitch" fragments together!

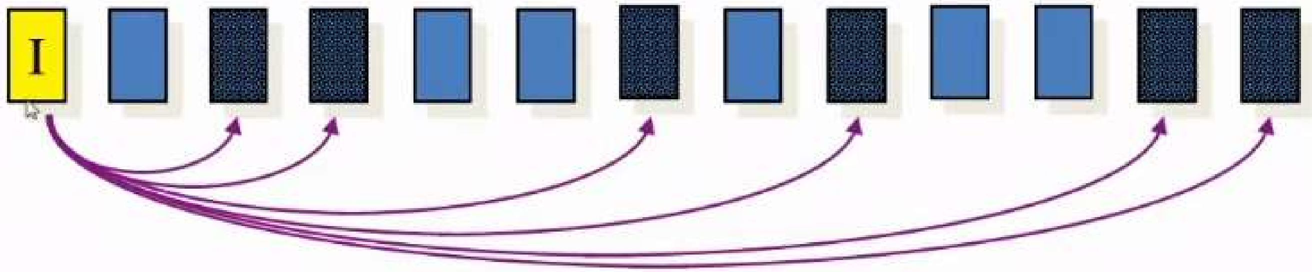
- Minuses
  - Impossible to do true random access
  - Reliability
    - Break one link in the chain and...

Strategy 2: File Allocation Table (FAT)

- Create a table with an entry for each block
  - Overlay the table with a linked list
  - Each entry serves as a link in the list
  - Each table entry in a file has a pointer to the next entry in that file (with a special "eof" marker)
  - A "0" in the table entry ➔ free block

- Comparison with linked allocation
  - If FAT is cached ➔ better sequential and random access performance
    - How much memory is needed to cache entire FAT?
      - 400GB disk, 4KB/block ➔ 100M entries in FAT ➔ 400MB
    - Solution approaches
      - Allocate larger clusters of storage space
      - Allocate different parts of the file near each other ➔ better locality for FAT

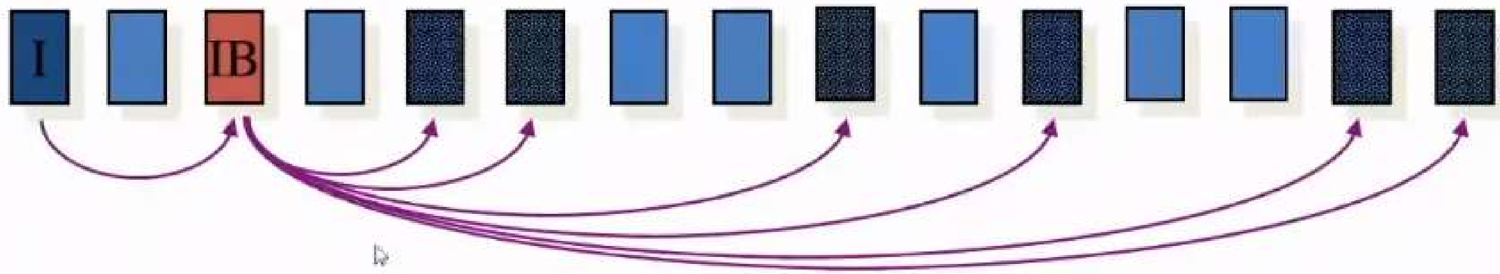
Strategy 3: Direct Allocation



- File header points to each data block

- Pluses
  - Easy to create, grow & shrink files
  - Little fragmentation
  - Supports direct access
- Minuses
  - Inode is big or variable size
  - How to handle large files?

Strategy 4: Indirect Allocation

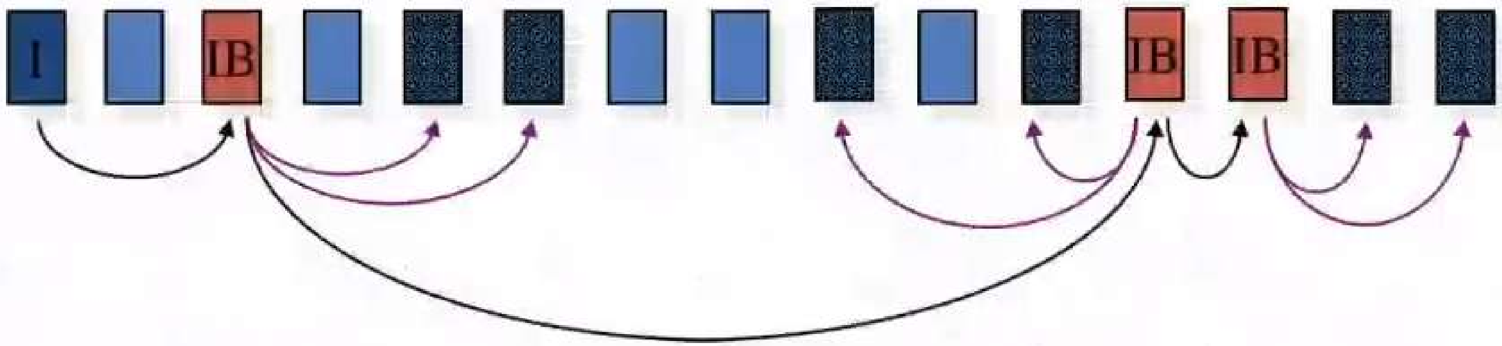


- Create a non-data block for each file called the *indirect block*
  - A list of pointers to file blocks
- File header contains a pointer to the indirect block

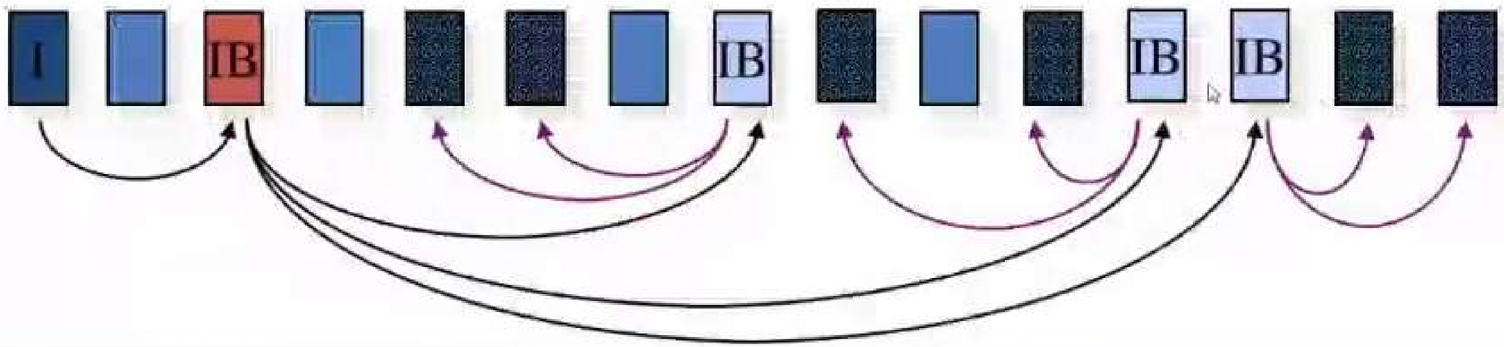
- Pluses
  - Easy to create, grow & shrink files
  - Little fragmentation
  - Supports direct access
- Minuses
  - Overhead of storing index when files are small
  - How to handle large files?

Indexed Allocation for Large Files

- Linked indirect blocks (IB+IB+…)



- Multilevel indirect blocks (IB*IB*…)



- Why bother with indirect blocks?
  - A. Allows greater file size.
  - B. Faster to create files.
  - C. Simpler to grow files.
  - D. Simpler to prepend and append to files.

Direct/Indirect Hybrid Strategy in Unix

- File header contains 13 pointers
  - 10 pointes to data blocks; 11th pointer → indirect block; 12th pointer → doubly-indirect block; and 13th pointer → triply-indirect block
- Implications
  - Upper limit on file size (~2 TB)
  - Blocks are allocated dynamically (allocate indirect blocks only for large files)
- Features
  - Pros
    - Simple
    - Files can easily expand (add indirect blocks proportional to file size)
    - Small files are cheap (fit in direct allocation)
  - Cons
    - Large files require a lot of seek to access indirect blocks

Visualization