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 Duality 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

Block vs. Sector

- The operating system may choose to use a larger block size than the sector size of the physical disk. Each block consists of consecutive sectors. Why?
  - A larger block size increases the transfer efficiency (why?)
  - It can be convenient to have block size match (a multiple of) the machine's page size (why?)
- Some systems allow transferring of many sectors between interrupts.
- Some systems interrupt after each sector operation (rare these days)
  - “consecutive” sectors may mean “every other physical sector” to allow time for CPU to start the next transfer before the head moves over the desired sector

File System Functionality and Implementation

- File system functionality:
  - Pick the blocks that constitute a file.
    - Must balance locality with expandability.
    - Must manage free space.
  - Provide file naming organization, 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, permissions)
    - File data blocks (data).
  - Superblocks, group descriptors, other metadata…

File System Properties

- Most files are small.
  - Need strong 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 should be reasonably efficient.
- Most systems fit the following profile:
  1. Most files are small
  2. Most disk space is taken up by large files.
  3. I/O operations target both small and large files.

  --> The per-file cost must be low, but large files must also have good performance.

If my file system only has lots of big video files what block size do I want?

1. Large
2. Small

How do we find and organize files on the disk?

The information that we need:

file header points to data blocks

fileID 0, Block 0 --> Disk block 19

fileID 0, Block 1 --> Disk block 4,528

…

Key performance issues:

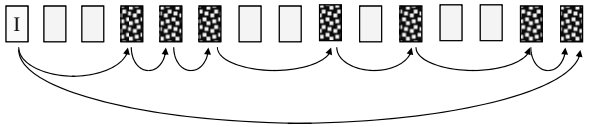
1. We need to support sequential and random access.
2. What is the right data structure in which to maintain file location information?
3. How do we lay out the files on the physical disk?

File Allocation Methods
Contiguous allocation

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

- Pluses
  - Best file read performance
  - Efficient sequential & random access
- Minuses
  - Fragmentation!
  - Problems with file growth
    - Pre-allocation?
    - On-demand allocation?

File Allocation Methods
Linked allocation



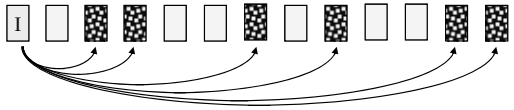
- Files stored as a linked list of blocks
- File header contains a pointer to the first and last file blocks

- Pluses
  - Easy to create, grow & shrink files
  - No external fragmentation
- Minuses
  - Impossible to do true random access
  - Reliability
    - Break one link in the chain and...

File Allocation Methods
Linked allocation – File Allocation Table (FAT) (Win9x, OS2)

- Maintain linked list in a separate table
  - A table entry for each block on disk
  - 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

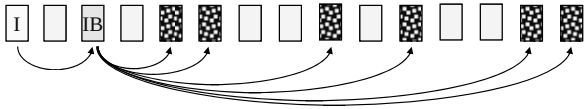
File Allocation Methods
Direct allocation



- 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?

File Allocation Methods
Indexed allocation

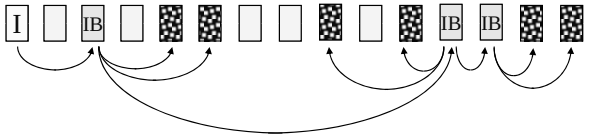


- Create a non-data block for each file called the *index block*
  - A list of pointers to file blocks
- File header contains the index block

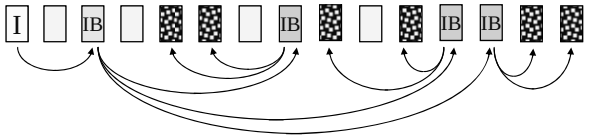
- Pluses
  - Easy to create, grow & shrink files
  - Little fragmentation
  - Supports direct access
- Minuses
  - Overhead of storing index when files are small
  - How to handle large files?

Indexed Allocation
Handling large files

- Linked index blocks (IB+IB+…)



- Multilevel index blocks (IB*IB*…)



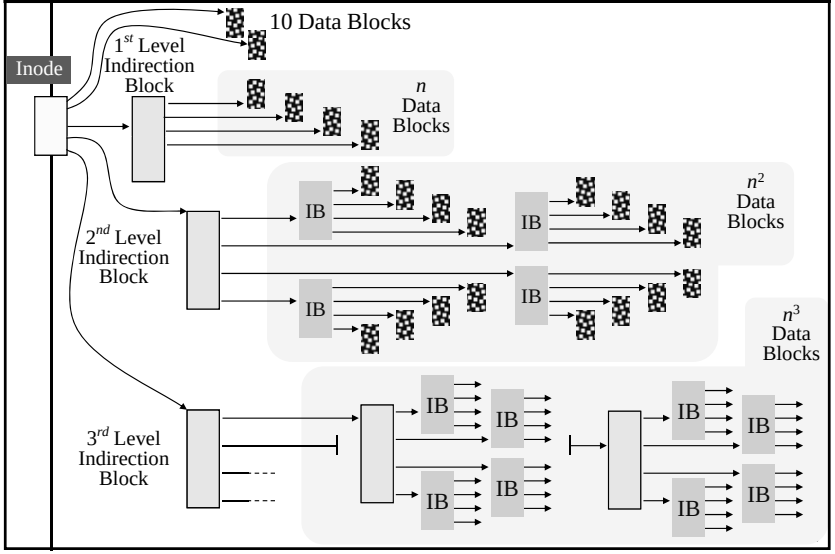
- Why bother with index blocks?
  - A. Allows greater file size.
  - B. Faster to create files.
  - C. Simpler to grow files.
  - D. Simpler to prepend and append to files.
  - E. Scott Summers is the X-men's Cyclops

Multi-level Indirection 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
    - Small files are cheap
  - Cons
    - Large files require a lot of seek to access indirect blocks

Indexed Allocation in UNIX

Multilevel, indirection, index blocks



- How big is an inode?
  - A. 1 byte
  - B. 16 bytes
  - C. 128 bytes
  - D. 1 KB
  - E. 16 KB

Allocate from a free list

- Need a data block
  - Consult list of free data blocks
- Need an inode
  - Consult a list of free inodes
- Why do inodes have their own free list?
  - A. Because they are fixed size
  - B. Because they exist at fixed locations
  - C. Because there are a fixed number of them

Free list representation

- Represent the list of free blocks as a *bit vector*:

  1111111111111110011101010111011111...
  - If bit *i* = 0 then block *i* is *free*, if *i* = 1 then it is *allocated*

Simple to use and vector is compact:
1TB disk with 4KB blocks is 2^28 bits or 32 MB

If free sectors are uniformly distributed across the disk then the expected number of bits that must be scanned before finding a "0" is

$$n/r$$

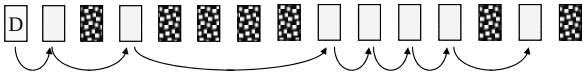
where

n = total number of blocks on the disk,
r = number of free blocks

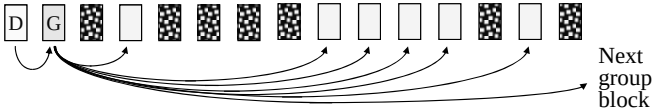
If a disk is 90% full, then the average number of bits to be scanned is 10, independent of the size of the disk

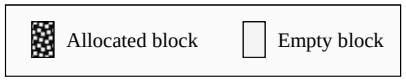
Other Free List Representations

- In-situ linked lists



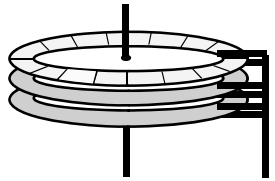
- Grouped lists





Naming and Directories

- Files are organized in directories
  - Directories are themselves files
  - Contain <name, pointer to file header> table
- Only OS can modify a directory
  - Ensure integrity of the mapping
  - Application programs can read directory (e.g., ls)
- Directory operations:
  - List contents of a directory
  - Search (find a file)
    - Linear search
    - Binary search
    - Hash table
  - Create a file
  - Delete a file



- Every directory has an inode
  - A. True
  - B. False
- Given only the inode number (inumber) the OS can find the inode on disk
  - A. True
  - B. False

Directory Hierarchy and Traversal

- Directories are often organized in a hierarchy
- Directory traversal:
  - How do you find blocks of a file? Let's start at the bottom
    - Find file header (inode) – it contains pointers to file blocks
    - To find file header (inode), we need its I-number
    - To find I-number, read the directory that contains the file
    - But wait, the directory itself is a file
    - Recursion !!
  - Example: Read file /A/B/C
    - C is a file
    - B/ is a directory that contains the I-number for file C
    - A/ is a directory that contains the I-number for file B
    - How do you find I-number for A?
      - "/" is a directory that contains the I-number for file A
      - What is the I-number for "/"? In Unix, it is 2

Directory Traversal (Cont'd.)

- How many disk accesses are needed to access file /A/B/C?
  1. Read I-node for "/" (root) from a fixed location
  2. Read the first data block for root
  3. Read the I-node for A
  4. Read the first data block of A
  5. Read the I-node for B
  6. Read the first data block of B
  7. Read I-node for C
  8. Read the first data block of C
- Optimization:
  - Maintain the notion of a current working directory (CWD)
  - Users can now specify relative file names
  - OS can cache the data blocks of CWD

Naming and Directories

- Once you have the file header, you can access all blocks within a file
  - How to find the file header? Inode number + layout.
- Where are file headers stored on disk?
  - In early Unix:
    - Special reserved array of sectors
    - Files are referred to with an index into the array (I-node number)
    - Limitations: (1) Header is not near data; (2) fixed size of array → fixed number of files on disk (determined at the time of formatting the disk)
  - Berkeley fast file system (FFS):
    - Distribute file header array across cylinders.
  - Ext2 (linux):
    - Put inodes in block group header.
- How do we find the I-node number for a file?
  - Solution: directories and name lookup

- A corrupt directory can make a file system useless
  - A. True
  - B. False