CS 537
Lecture 13
File Systems Internals

Michael Swift

Workloads

- Motivation: Workloads influence design of file system
- File characteristics (measurements of UNIX and NT)
  - Most files are small (about 8KB)
  - Most of the disk is allocated to large files
    - (90% of data is in 10% of files)
- Access patterns
  - Sequential: Data in file is read/written in order
    - Most common access pattern
  - Random (direct): Access block without referencing predecessors
    - Difficult to optimize
  - Access files in same directory together
    - Spatial locality
  - Access meta-data when access file
    - Need meta-data to find data

Goals

- OS allocates LBNs (logical block numbers) to meta-data, file data, and directory data
  - Workload items accessed together should be close in LBN space
- Implications
  - Large files should be allocated sequentially
  - Files in same directory should be allocated near each other
  - Data should be allocated near its meta-data
- Meta-Data: Where is it stored on disk?
  - Embedded within each directory entry
  - In data structure separate from directory entry
    - Directory entry points to meta-data

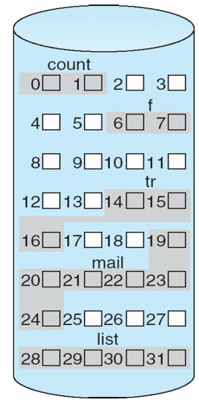
Allocation Strategies

- Progression of different approaches
  - Contiguous
  - Extent-based
  - Linked
  - File-allocation Tables
  - Indexed
  - Multi-level Indexed
- Questions
  - Amount of fragmentation (internal and external)?
  - Ability to grow file over time?
  - Seek cost for sequential accesses?
  - Speed to find data blocks for random accesses?
  - Wasted space for pointers to data blocks?

Contiguous Allocation

- Allocate each file to contiguous blocks on disk
  - Meta-data: Starting block and size of file
  - OS allocates by finding sufficient free space
    - Must predict future size of file; Should space be reserved?
  - Example: IBM OS/360
- Advantages
  - Little overhead for meta-data
  - Excellent performance for sequential accesses
  - Simple to calculate random addresses
- Drawbacks
  - Horrible external fragmentation (Requires periodic compaction)
  - May not be able to grow file without moving it

Contiguous Allocation of Disk Space

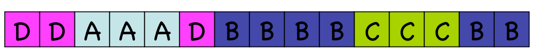


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-Based Allocation

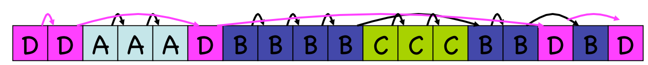
- Allocate multiple contiguous regions (extents) per file
  - Meta-data: Small array (2-6) designating each extent
    - Each entry: starting block and size



- Improves contiguous allocation
  - File can grow over time (until run out of extents)
  - Helps with external fragmentation
- Advantages
  - Limited overhead for meta-data
  - Very good performance for sequential accesses
  - Simple to calculate random addresses
- Disadvantages (Small number of extents):
  - External fragmentation can still be a problem
  - Not able to grow file when run out of extents

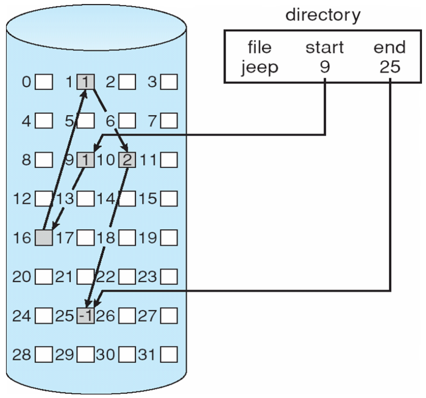
Linked Allocation

- Allocate linked-list of fixed-sized blocks
  - Meta-data: Location of first block of file
    - Each block also contains pointer to next block
  - Examples: TOPS-10, Alto



- Advantages
  - No external fragmentation
  - Files can be easily grown, with no limit
- Disadvantages
  - Cannot calculate random addresses w/o reading previous blocks
  - Sequential bandwidth may not be good
    - Try to allocate blocks of file contiguously for best performance
  - Sensitivity to corruption
- Trade-off: Block size (does not need to equal sector size)
  - Larger --> ??
  - Smaller --> ??

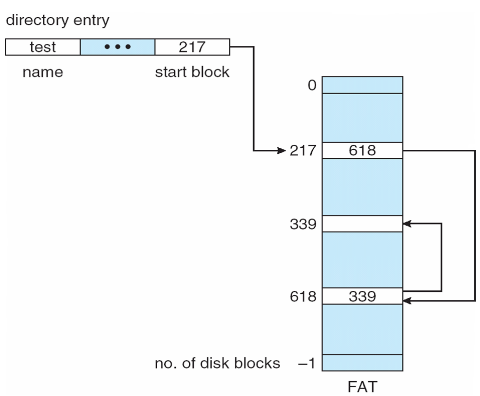
Linked Allocation



File-Allocation Table (FAT)

- Variation of Linked allocation
  - Keep linked-list information for all files in on-disk FAT table
  - Meta-data: Location of first block of file
    - And, FAT table itself
- Comparison to Linked Allocation
  - Same basic advantages and disadvantages
  - Disadvantage: Read from two disk locations for every data read
  - Optimization: Cache FAT in main memory
    - Advantage: Greatly improves random accesses

File-Allocation Table

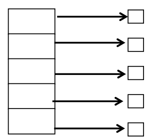


Indexed Allocation

- Allocate fixed-sized blocks for each file
  - Meta-data: Fixed-sized array of block pointers
    - Allocate space for ptrs at file creation time
- Advantages
  - No external fragmentation
  - Files can be easily grown, with no limit
  - Supports random access
- Disadvantages
  - Large overhead for meta-data:
    - Wastes space for unneeded pointers (most files are small!)

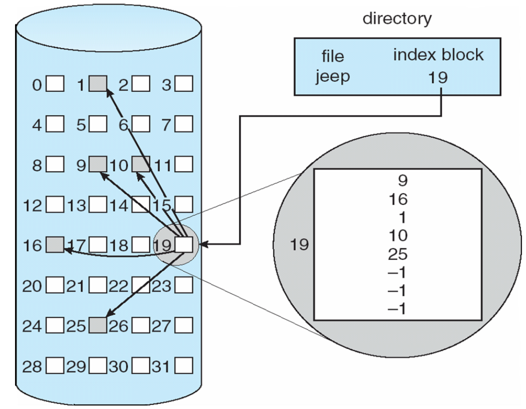
Indexed Allocation

- Brings all pointers together into the index block
- Logical view



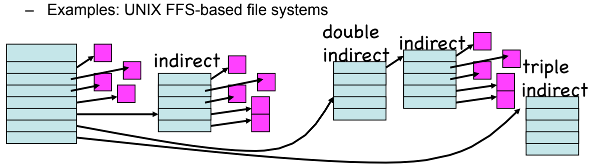
index table

Example of Indexed Allocation



Multi-Level Indexed Files

- Variation of Indexed Allocation
  - Dynamically allocate hierarchy of pointers to blocks as needed
  - Meta-data: Small number of pointers allocated statically
    - Additional pointers to blocks of pointers
  - Examples: UNIX FFS-based file systems

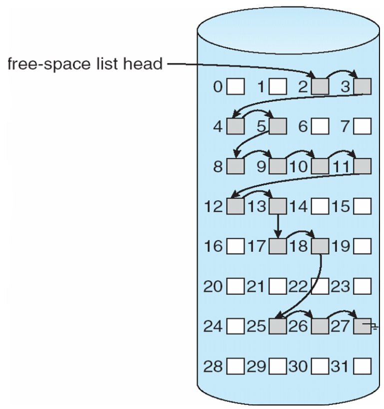


- Comparison to Indexed Allocation
  - Advantage: Does not waste space for unneeded pointers
    - Still fast access for small files
    - Can grow to what size??
  - Disadvantage: Need to read indirect blocks of pointers to calculate addresses (extra disk read)
    - Keep indirect blocks cached in main memory

Free space management

- How do you remember which blocks are free?
  - What operations are needed?
    - Free a block
    - Get a free block(s) -- in some particular location
- Free list: linked list of free blocks
  - Advantages: simple, constant-time operation
  - Disadvantage: rapidly loses locality
  - Used in Unix UFS and FAT
- Bitmap: bitmap of all blocks indicating which are free
  - Advantages: can find strings of consecutive free blocks
    - X86 provides instructions to find 1 bits
  - Disadvantages: space overhead
  - Used in Unix FFS

Linked Free Space List on Disk



Directory Implementation

- A directory is a file containing
  - name
  - metadata about file (Windows)
    - size
    - owner
    - data locations
  - Pointer to file metadata (Unix)
- Organization
  - Linear list of file names with pointer to the data blocks
    - simple to program
    - time-consuming to execute
  - BTree – balanced tree sorted by name
    - Faster searching for large directories

Efficiency and Performance

- Efficiency dependent on disk allocation and directory algorithms
    - How many accesses does it take to open a file?
    - Once you read a block, what do you have to do to read the next block?
- Performance
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
    - free-behind: release block as soon as read (make space for others)
    - read ahead: read blocks before you need them (so you don't need to wait)

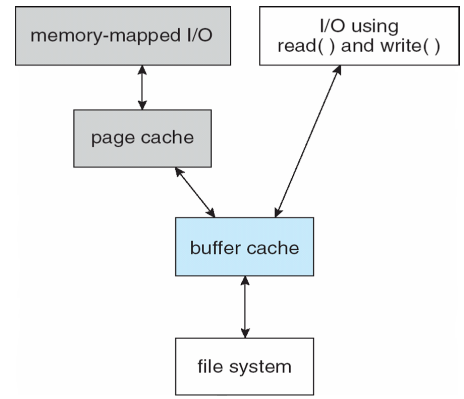
Caching

- File systems cache commonly used data in the **buffer cache**
  - Set of disk blocks cached in memory
  - Associated metadata say whether clean/dirty/where on disk it belongs.
  - Buffer cache is a layer below file system
    - File system asks buffer cache for data
    - If not available, buffer cache will ask disk for data
- File systems may cache metadata separately
  - Linux "dentry"s store directory entries for fast name parsing
  - Linux "inodes" store file metadata (block location) for fast file access

Page Cache

- A page cache caches pages rather than disk blocks using virtual memory techniques
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

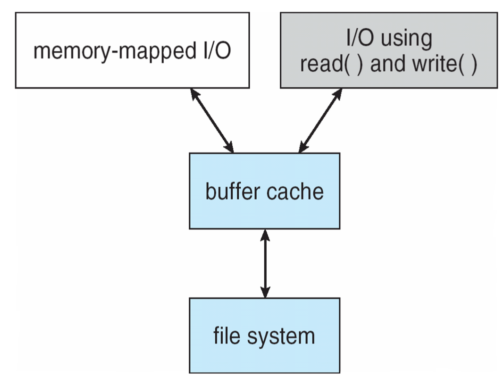
I/O Without a Unified Buffer Cache



Unified Buffer Cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O

I/O Using a Unified Buffer Cache



The flat (i-node) file system

- Each file is known by a number, which is the number of the i-node
  - seriously – 1, 2, 3, etc.!
  - why is it called "flat"?
- Files are created empty, and grow when extended through writes

The tree (directory, hierarchical) file system

- A directory is a flat file of fixed-size entries
- Each entry consists of an i-node number and a file name

i-node number	File name
152	.
18	..
216	my_file
4	another_file
93	oh_my_god
144	a_directory

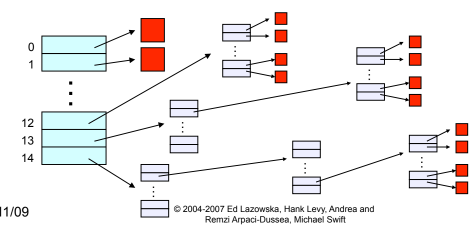
- It's as simple as that!

Using directories

- How do you find files?
  - Read the directory, search for the name you want (checking for wildcards)
- How do you list files (ls)
  - Read directory contents, print name field
- How do you list file attributes (ls -l)
  - Read directory contents, open inodes, print name + attributes
- How do you sort the output (ls -S, ls -t)
  - The FS doesn't do it -- ls does it!

The "block list" portion of the i-node

- Clearly it points to blocks in the file contents area
- Must be able to represent very small and very large files. How?
- Each inode contains 15 block pointers
  - first 12 are direct blocks (i.e., 4KB blocks of file data)
  - then, single, double, and triple indirect indexes



So …

- Only occupies 15 x 4B in the i-node
- Can get to 12 x 4KB = a 48KB file directly
  - (12 direct pointers, blocks in the file contents area are 4KB)
- Can get to 1024 x 4KB = an additional 4MB with a single indirect reference
  - (the 13th pointer in the i-node gets you to a 4KB block in the file contents area that contains 1K 4B pointers to blocks holding file data)
- Can get to 1024 x 1024 x 4KB = an additional 4GB with a double indirect reference
  - (the 14th pointer in the i-node gets you to a 4KB block in the file contents area that contains 1K 4B pointers to 4KB blocks in the file contents area that contian 1K 4B pointers to blocks holding file data)
- Maximum file size is 4TB

File system consistency

- Both i-nodes and file blocks are cached in memory
- The "sync" command forces memory-resident disk information to be written to disk
  - system does a sync every few seconds
- A crash or power failure between sync's can leave an inconsistent disk
- You could reduce the frequency of problems by reducing caching, but performance would suffer big-time

i-check: consistency of the flat file system

- Is each block on exactly one list?
  - create a bit vector with as many entries as there are blocks
  - follow the free list and each i-node block list
  - when a block is encountered, examine its bit
    - If the bit was 0, set it to 1
    - if the bit was already 1
      - if the block is both in a file and on the free list, remove it from the free list and cross your fingers
      - if the block is in two files, call support!
  - if there are any 0's left at the end, put those blocks on the free list

d-check: consistency of the directory file system

- Do the directories form a tree?
- Does the link count of each file equal the number of directories links to it?
  - I will spare you the details
    - uses a zero-initialized vector of counters, one per i-node
    - walk the tree, then visit every i-node

Protection systems

- FS must implement some kind of protection system
  - to control who can access a file (user)
  - to control how they can access it (e.g., read, write, or exec)
- More generally:
  - generalize files to objects (the "what")
  - generalize users to principals (the "who", user or program)
  - generalize read/write to actions (the "how", or operations)
- A protection system dictates whether a given action performed by a given principal on a given object should be allowed
  - e.g., you can read or write your files, but others cannot
  - e.g., your can read `/etc/motd` but you cannot write to it

Model for representing protection

- Two different ways of thinking about it:
  - access control lists (ACLs)
    - for each object, keep list of principals and principals' allowed actions
    - Like a guest list (check identity of caller on each access)
  - capabilities
    - for each principal, keep list of objects and principal's allowed actions
    - Like a key (something you present to open a door)
- Both can be represented with the following matrix:

objects

principals	/etc/passwd	/home/swift	/home/guest
root	rw	rw	rw
swift	r	rw	r
guest			r

ACL (column) · capability (row)

ACLs vs. Capabilities

- Capabilities are easy to transfer
  - they are like keys: can hand them off
  - they make sharing easy
- ACLs are easier to manage
  - object-centric, easy to grant and revoke
    - to revoke capability, need to keep track of principals that have it
    - hard to do, given that principals can hand off capabilities
- ACLs grow large when object is heavily shared
  - can simplify by using "groups"
    - put users in groups, put groups in ACLs
    - you are could be in the "cs537-students" group
  - additional benefit
    - change group membership, affects ALL objects that have this group in its ACL

Protection in the Unix FS

- Objects: individual files
- Principals: owner/group/world
- Actions: read/write/execute

- This is pretty simple and rigid, but it has proven to be about what we can handle!