CSCI [4|6]730 Operating Systems

File System: Implementation

How are file systems implemented?

- How do we represent
  - Directories (link file names to file “structure”)
  - The list of blocks containing the data
  - Other information such as access control list or permissions, owner, time of access, etc?
- How can we be smart about the layout?

File System Design Motivations

- Workloads influence design of file system
- File characteristics (measurements of UNIX and NT):
  - Most files are *small* (about 8KB)
    - Block size can’t be too big (why not?)
    - Is this still true? Why?
  - BUT - Most of the disk is allocated to large files
    - (90% of data is in 10% of number of files)
    - Large file access should be reasonable efficient.
- Support various file access patterns…

File System Design Motivation (cont)

- Access patterns:
  - Sequential: Data in file is read/written in order
    - Most common access pattern
  - Random (direct): Access block without referencing the predecessor block
    - Difficult to optimize
  - Access files in same directory together
    - Spatial locality
  - Access meta-data (i-node, FCB) when access file
    - Need meta-data to find data

File Operation Implementation

- Seek: Repositioning within a file:
  - Directory searched for appropriate entry & current file position pointer is updated (also called a file *seek*)
- Deleting a file:
  - Search directory entry for named file, release associated file space and erase directory entry
- Truncating a file:
  - Keep attributes the same, but reset file size to 0, and reclaim file space.

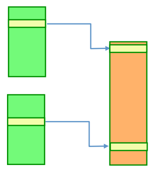
File Operation Implementation

- Create a file:
  - Find space in the file system, and add a directory entry.
- Writing in a file:
  - System call specifying name & information to be written.
    - Given name, system searches directory structure to find file. System keeps *write pointer* to the location where next write occurs, updating as writes are performed. Update meta-data.
- Reading a file:
  - System call specifying name of file & where in memory to stick contents. Name is used to find file, and a *read pointer* is kept to point to next read position. (can combine write & read to *current file position pointer*). Update meta-data.

***Thought* Questions**: How should files be accessed on reads and writes? How can we avoid reading/searching *directory* **on** every read/write access?

- Need to caches open file pointers
  - HINT: we have file descriptors in UNIX, it is a *reason* for this.
- How do we do this procedurally?

Opening Files



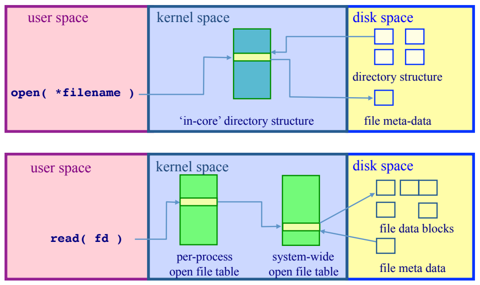
- Observation: Expensive to access files with full pathnames
  - On every read/write operation:
    - Traverse directory structure
    - Check access permissions
- Idea!: Separate `open()` before first access
  - User specifies mode: read and/or write
  - Search directories once for filename and check permissions
  - Copy relevant meta-data to *system wide open file table* in memory
  - Return index in open file table to process (file descriptor)
  - Process uses file descriptor to read/write to file
- Multi-process support: via a separate *per-process-open file table* where each process maintains
  - Current file position in file (offset for read/write)
  - Open mode

Multi-Process File Access Support

- Two level of internal tables:
  - Per-process open file table
    - Tracks all files open by a process (process-centric information):
      - Current position pointer (on read/write) where did it read/write last, and access Rights
      - Indexes into the system-wide table for other info.
  - System-wide *open* file table
    - Process Independent information
      - Location of file on disk
      - Access dates, file size
      - File open count (# processes accessing file)

Example: Accessing Files (Steps via `open()`)

1. Search directory structure (part may be cached in memory)
2. Get meta-data, copy (if needed) into system-wide open file table
3. Adjust count of #processes that have file open in the system wide table.
4. Entry made in per-process open file table, w/ pointer to system wide table
5. Return pointer to entry in per-process file table to application



Goals

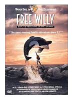
- OS allocates logical block numbers (LBN) to meta-data, file data, and directory data
  - Workload items accessed together should be close in LBN space
- Implications
  - Large files should be allocated sequentially
  - Files in same directory should be allocated near each other
  - Data should be allocated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

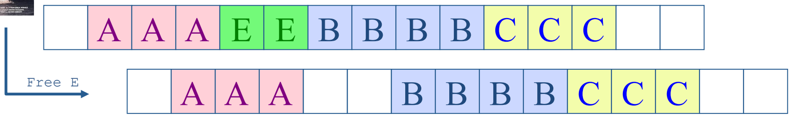
Allocation Strategies

- Progression of different approaches (reminiscent of memory structure 'progression' of approaches)
  - Contiguous
  - Extent-based
  - Linked
  - File-Allocation Tables
  - Indexed
  - Multi-level Indexed
- Questions/Issues:
  - 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

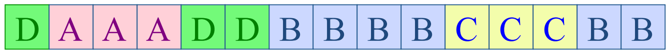
- Allocate each file to contiguous blocks on disk
  - Meta-data: (1) Starting block and (2) size of file (base & bound)
  - OS allocates by finding sufficient free space
    - Must predict future size of file; Should space be reserved?
  - Examples: IBM OS/360, CDROMS, DVDs.
- Advantages:
  - Little overhead for meta-data
  - Excellent performance for sequential accesses
  - Simple to calculate random addresses
- Disadvantages:
  - Horrible external *fragmentation* (Requires periodic compaction)
  - May not be able to grow file without moving it
    - Solution: Extends -- pointer to extent(s) in meta-data (i-node)… See next





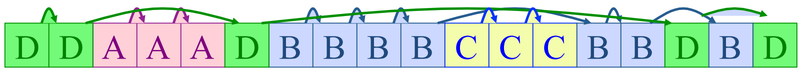
Extent-Based Allocation

- Allocate multiple contiguous regions (extents) per file (e.g., Veritas File System).
  - 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



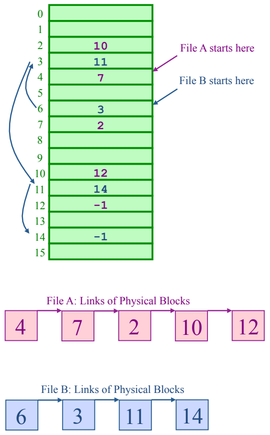
Linked Allocation

- Allocate linked-list of fixed-sized blocks
  - Meta-data: Location of first (fixed size) 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
  - Reliability - loose pointer (1) cluster blocks (2) user double linked list
- Trade-off: Block size (does not need to equal sector size)
  - Larger ⇒ ?? , Smaller ⇒ ?? [Thought Question]



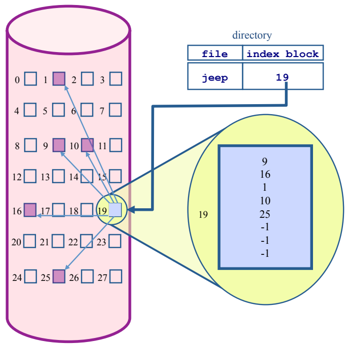
File-Allocation Table (FAT)

- Variation of Linked allocation (e.g., MS-DOS, OS/2)
  - Keep linked-list information for all files in on-disk FAT table
  - Meta-data: Location of first block of file
    - And then lookup rest in FAT table
  - FAT located at beginning of each partition
    - indexed by block number
    - entry contains block number of next entry
- Comparison to Linked Allocation
  - Advantage: Random access improved because disk head can read location in FAT
  - Disadvantage: Read from two disk locations for every data read (FAT + actual block)
  - Optimization: Cache FAT in main memory
    - Advantage: Greatly improves random accesses
    - Still very hard to access random file blocks):



Indexed Allocation

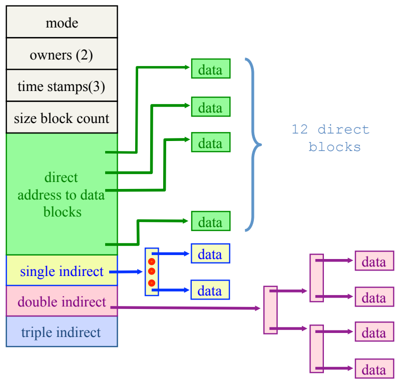
- Allocate *fixed-sized* blocks for each file
  - Meta-data: Fixed-sized array of block pointers
    - Allocate space for ptrs at file creation time
  - Directory Entry: Address of index block
- Advantages:
  - no external fragmentation (fixed sized blocks)
  - supports random access
- Disadvantages:
  - waste of space (pointer), space wise worse than linked list
    - A file of one block need the ENTIRE additional block for the index block
    - Need to know file size priory
- Implementation Issues:
  - How big should an index block be?
    - not too small: limits file size
    - too big: lots of wasted ointers
  - How do we accommodate very large files?
    - linked, multileveled, combined



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

i-node contains 15 pointers



Intuition: most files are small

Unix i-nodes

- 4.3 BSD file system
  - Inode
    - 12 direct block addresses
    - 1 indirect block of addresses
    - 1 double-indirect addresses
  - Any block can be found with at most 3 disk accesses
- Example: if block addresses are 4 bytes and blocks are 1024 bytes what is the maximum file size?
  - Number of block addresses per block = 1024/4 = 256
    - Number of blocks mapped by direct blocks → 12
    - Number of blocks mapped by in-direct blocks → 256 (256 addresses)
    - Number of blocks double in-direct blocks → 256^2 → 65,536
  - Max file size: (12 + 256 + 65,536) * 1024 = 66MB (67,383,296 bytes)
- Modern file system have 1 triple index blocks

Free-Space Management

- Motivation: Need to re-claim space from deleted files, keep a free space list, indexed by blocks.
- Two main approaches to implement the free 'list':
  - Bit Vector
  - Linked Lists

Bit Vector

- Represent the list of free blocks as a *bit vector,* 1 bit representing one block :

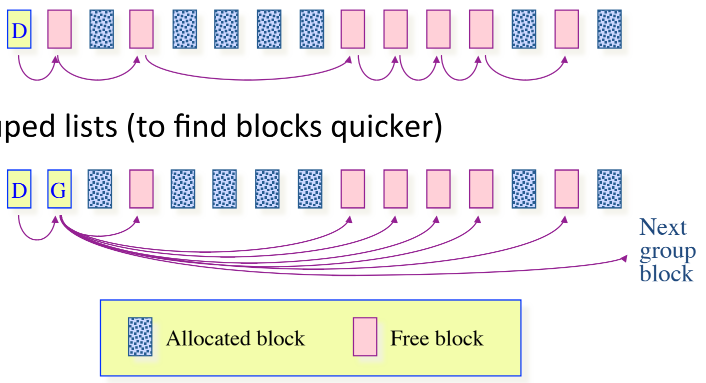
  `1111111111111111001110101011101111...`
  - If bit `i = 0` then block `i` is free, if `i = 1` then it is *allocated*
- Advantages: Simple to use.
- Disadvantages: The vector can be large, 17.5 million elements for a 9 GB disk (2.2 MB worth of bits)
- Justification: if free sectors are *uniformly* distributed 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

`Not likely, if they were I/O would be poor`

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

Linked List Representations

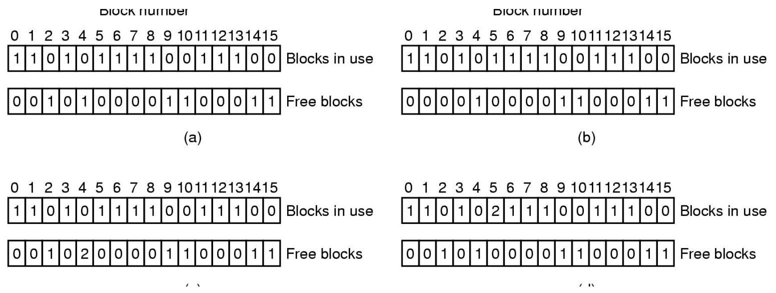
- In-situ linked lists (no wasted space)
- Grouped lists (to find blocks quicker)



File System Consistency

- Motivation: Recover from a system crash before modified files written back
  - Leads to inconsistency in FS
  - fsck (UNIX) & scandisk (Windows) check FS consistency
- Approach:
  - Check both (1) blocks (block consistency) and (2) files (consistency) separately.
- Algorithm 1: Block Consistency:
  - Build 2 tables, each containing counter for all blocks (init to 0)
    - 1st table checks how many times a block is in a file
    - 2nd table records how often block is present in the free list
      - >1 not possible if using a bitmap
  - Read all i-nodes, and modify table 1
  - Read free-list and modify table 2
  - Consistent state if block is either in table 1 or 2, but not both
- Algorithm 2: File Consistency:
  - Use a file counter instead of a block counter (appear in directories, compare with link count stored in inode)

Examples: Inconsistent States



Block number

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

(a)

Block number

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1

(b)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

- File system states
  - (a) (1-0) Consistent
  - (b) (0-0) missing block - no harm but wasted space
  - (c) (0-2) duplicate block in free list - ok, just add to free list
  - (d) (2-0) duplicate data block 5 - if either files are removed block will be on free list, leading to situations where block is in both free list and USE list, if both are removed, block in free list twice
    - ACTION: allocate new block to copy block 5 into it, insert copy in one of the files