File System Implementation Issues

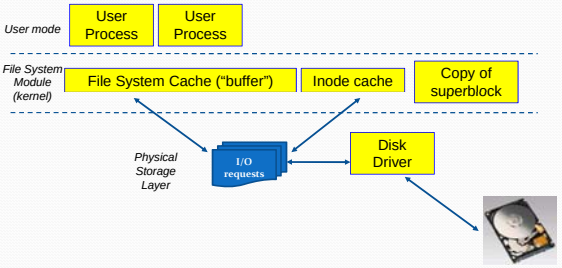
Ken Birman

File System Implementation

- Change of topic…
  - How exactly are file systems implemented?
  - Comes down to: how do we represent
    - Volumes
    - 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?
  - And, can we be smart about layout?

2

Overall structure



3

Basic sequence

- User application does a file operation
  - Perhaps, open, then seek, then read
- O/S tries to satisfy requests using cached records
  - If needed data isn't in the cache, posts an I/O request and makes the user process (actually: the kernel thread) wait
  - Upon completion, perform the request, copying data into or from the user's space as needed
  - Then make user process runable again
- With a cache hit, a file system operation might take 100us… if it misses, perhaps 10-50ms to a hard disk

4

Prefetch policy

- Most kernels *prefetch* file blocks
  - Fetch first two blocks of data when file is opened
  - If user reads blocks i, i+1 then prefetch i+2
- Goal?
  - Keep the disk a little ahead of the application
  - But don't waste time doing unnecessary I/O

5

Write-behind policy

- When user does a write, update the *cache*
  - Thus, cached version of file, inode and super block can be "more current" than the version actually on the disk
  - Periodically, like once every 60 seconds, "sync" the cache with the disk by writing out dirty entries
  - Also if file is closed or user explicitly calls *fsync()*
- *Crash?*
  - O/S wants file system to make sense… but doesn't care if some "unsynced" data was lost!
  - If an application does something sensitive, it calls fsync() before telling the human user the request is done

6

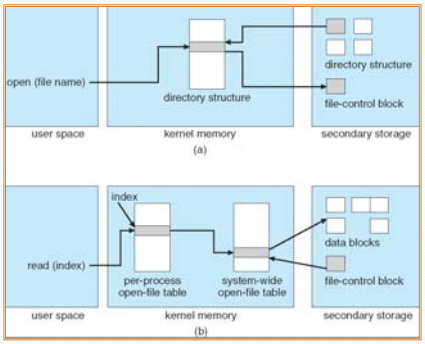
File Control Block

- FCB has all the information about the file
  - Linux systems call these i-node structures

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

7

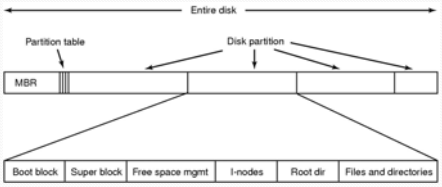
Files Open and Read



8

File System Layout

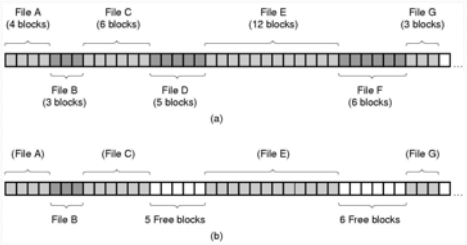
- File System is stored on disks
  - Disk is divided into 1 or more partitions
  - Sector 0 of disk called Master Boot Record
  - End of MBR has partition table (start & end address of partitions)
- First block of each partition has boot block



9

Implementing Files

- Contiguous Allocation: allocate files contiguously on disk



10

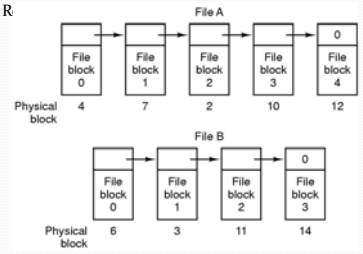
Contiguous Allocation

- Pros:
  - Simple: state required per file is start block and size
  - Performance: entire file can be read with one seek
- Cons:
  - Fragmentation: external is bigger problem
  - Usability: user needs to know size of file
- Used in CDROMs, DVDs

11

Linked List Allocation

- Each file is stored as linked list of blocks
  - First word of each block points to next block
  - R



12

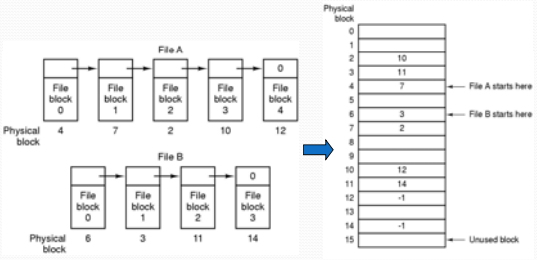
Linked List Allocation

- Pros:
  - No space lost to external fragmentation
  - Disk only needs to maintain first block of each file
- Cons:
  - Random access is costly
  - Data stored in blocks is no longer a power of 2

13

Using an in-memory table

- Implement a linked list allocation using a table
  - Called File Allocation Table (FAT)
  - Take pointer away from blocks, store in this table



14

FAT Discussion

- Pros:
  - Entire block is available for data
  - Random access is faster since entire FAT is in memory
- Cons:
  - Entire FAT should be in memory
    - For 20 GB disk, 1 KB block size, FAT has 20 million entries
    - If 4 bytes used per entry ⇒ 80 MB of main memory required for FS

15

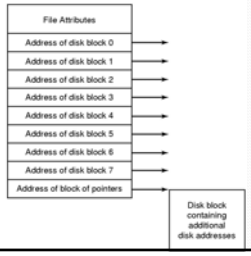
Linux: Three behaviors

- Small files: All the blocks are listed in the inode, so once the O/S has the inode cached, it also can find the blocks
- Medium sized files: the O/S can find the first few blocks. This gives it some breathing room to load the "indirection" block, which lists more file system blocks
- Large files: Here, all the file system blocks are indirection blocks. So to find any block, the O/S needs to first read the inode, then an indirection block, then the real block

16

I-nodes (Linux)

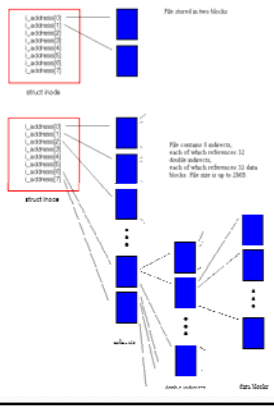
- Index-node (I-node) is a per-file data structure
  - Lists attributes and disk addresses of file's blocks
  - Pros: Space (max open files * size per I-node)
  - Cons: what if file expands beyond I-node address space?



17

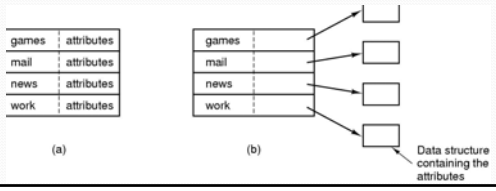
I-nodes (Linux)

- Small file with two blocks
- Large files: like a small file in which *all* the blocks are indirection blocks



Implementing Directories

- When a file is opened, OS uses path name to find dir
  - Directory has information about the file's disk blocks
    - Whole file (contiguous), first block (linked-list) or I-node
  - Directory also has attributes of each file
- Directory: map ASCII file name to file attributes & location
- 2 options: entries have all attributes, or point to file I-node



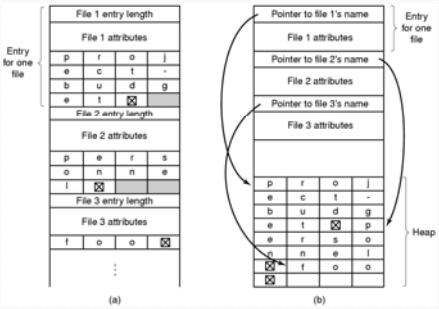
19

Implementing Directories

- What if files have large, variable-length names?
- Solution:
  - Limit file name length, say 255 chars, and use previous scheme
    - Pros: Simple Cons: wastes space
  - Directory entry comprises fixed and variable portion
    - Fixed part starts with entry size, followed by attributes
    - Variable part has the file name
    - Pros: saves space
    - Cons: holes on removal, page fault on file read, word boundaries
  - Directory entries are fixed in length, pointer to file name in heap
    - Pros: easy removal, no space wasted for word boundaries
    - Cons: manage heap, page faults on file names

20

Managing file names: Example



21

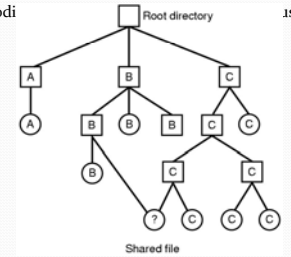
Directory Search

- Simple Linear search can be slow
- Alternatives:
  - Use a per-directory hash table
    - Could use hash of file name to store entry for file
    - Pros: faster lookup
    - Cons: More complex management
  - Caching: cache the most recent searches
    - Look in cache before searching FS

22

Shared Files

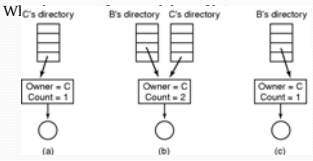
- If B wants to share a file owned by C
  - One Solution: copy disk addresses in B's directory entry
  - Problem: modi ıser's view



23

Sharing Files: Solutions

- 2 approaches:
  - Use i-nodes to store file information in directories
    - Cons: Wl

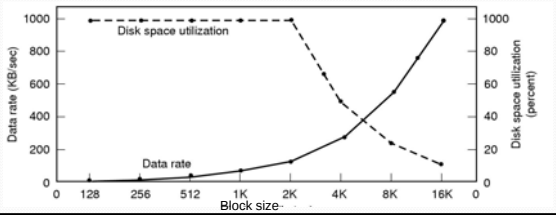


  - Symbolic links: B links to C's file by creating a file in its directory
    - The new Link file contains path name of file being linked
    - Cons: read overhead

24

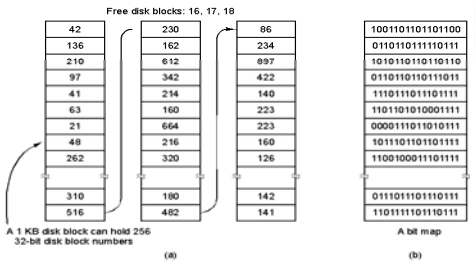
Disk Space Management

- Files stored as fixed-size blocks
- What is a good block size? (sector, track, cylinder?)
  - If 131,072 bytes/track, rotation time 8.33 ms, seek time 10 ms
  - To read k bytes block: 10+ 4.165 + (k/131072)*8.33 ms



Managing Free Disk Space

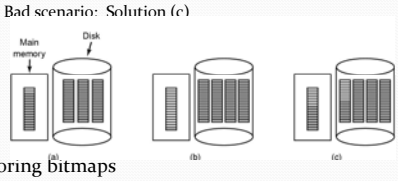
- 2 approaches to keep track of free disk blocks
  - Linked list and bitmap approach



26

Tracking free space

- Storing free blocks in a Linked List
  - Only one block need to be kept in memory
  - Bad scenario: Solution (c)

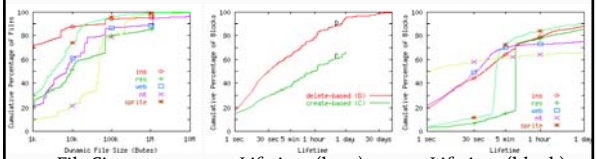


- Storing bitmaps
  - Lesser storage in most cases
  - Allocated disk blocks are closer to each other

27

Measured file lifetimes, sizes

- Source: "A Comparison of File System Workloads", Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson, 2000 USENIX Technical Conference



- File Size Lifetime (byte) Lifetime (block)
- (C): Block deleted-block created; (D): File deleted-File created

28

Observations?

- Many files are quite small
  - So Linux idea was a good one…
- Many files have fairly short lifetimes
  - So write-through cache policy can be a big win
  - With luck, file may be deleted before we ever write it to the disk at all!

29

Fancier issues

- Optimizing disk head movement
  - It turns out that the cost of moving the disk arm (seek) is very high
  - Optimal way to read a list of blocks?
    - Put them in order
    - Seek to the outside rim of the disk
    - Then pick up blocks in "inward" order….
  - Many disk drivers reorder requests for this reason. Algorithm is called "C-Scan" because disk arm moves in a circular motion

30

Perils of C-Scan

- Sorting the blocks before doing the I/O can cause some nasty surprises!
  - For example, suppose that John User
    - Creates a file "picture of Mom" (allocates inode, blocks..)
    - Crops the file: "Mom-cropped" (" " ")
    - Deletes "picture of Mom" (now some go back to free list)
    - Renames "Mom-cropped" as "picture of Mom" (updates dir.)
  - Sequence did MANY disk reads and writes!

31

Perils of C-Scan

- *Suppose the computer were to crash*
- If we did the I/O operations in order, then we simply chop off history in the past
- But if we were in the midst of a reordered list of operations we may see a mixture
  - Some things from the future
  - Some from the past

32

Bad things if a crash occurs

- Perhaps an inode we are using is still shown on the inode free list, or blocks allocated to one of the files are on the block free list
- For example free list could be updated and yet "Picture of Mom" wasn't actually deleted yet because that requires a write to the directory, too
- Could even end up with two files named Picture of Mom, or files that share blocks!
- Sounds like reordering disk I/O is very risky!

33

Perils of C-Scan

- To avoid this risk, modern O/S actually gives the disk driver a partially ordered list of I/O requests
  - Rule is: if the O/S says "A happens before B", the driver must respect that
  - But if O/S says "A and B can be done in any order" the driver is allowed to reorder them for C-Scan
- This ensures that if a crash occurs, we won't have reordered sensitive actions

34