CS 3204 Operating Systems

Lecture 25

Godmar Back

Announcements

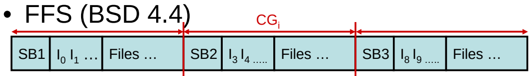
- Office Hours moved to one of McB 133, 116, or 124
- Please see revised grading policy posted on website
  - Will provide standing grade by May 2
  - Accept project 4 until May 7, 23:59pm
- Reading assignment: Ch 10, 11, 12

Storing Inodes

- Unix v7, BSD 4.3

Superblock	I_0 I_1 I_2 I_3 I_4	Rest of disk for files & directories

- FFS (BSD 4.4)



- Cylindergroups have superblock+bitmap+inode list+file space
- Try to allocate file & inode in same cylinder group to improve access locality

---

Positioning Inodes

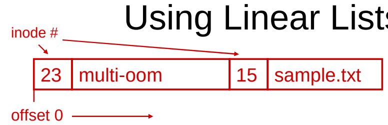
- Putting inodes in fixed place makes finding inodes easier
  - Can refer to them simply by inode number
  - After crash, there is no ambiguity as to what are inodes vs. what are regular files
- Disadvantage: limits the number of files per filesystem at creation time
  - Use “df –ih” on Linux to see how many inodes are used/free

---

Directories

- Need to find file descriptor (inode), given a name
- Approaches:
  - Single directory (old PCs), Two-level approaches with 1 directory per user
- Now exclusively hierarchical approaches:
  - File system forms a tree (or DAG)
- How to tell regular file from directory?
  - Set a bit in the inode
- Data Structures
  - Linear list of (inode, name) pairs
  - B-Trees that map name -> inode
  - Combinations thereof

Using Linear Lists



- Advantage: (relatively) simple to implement
- Disadvantages:
  - Scan makes lookup (& delete!) really slow for large directories
  - Could cause fragmentation (though not a problem in practice)

Using B-Trees

- Advantages:
  - Scalable to large number of files: in growth, in lookup time
- Disadvantage:
  - Complex
  - Overhead for small directories

Absolute Paths

- How to resolve a path name such as "/usr/bin/ls"?
  - Split into tokens using "/" separator
  - Find inode corresponding to root directory
    - (how? Use fixed inode # for root)
  - (*) Look up "usr" in root directory, find inode
  - If not last component in path, check that inode is a directory. Go to (*), looking for next comp
  - If last component in path, check inode is of desired type, return

Name Resolution

- Must have a way to scan an entire directory without other processes interfering -> need a "lock" function
  - But don't need to hold lock on /usr when scanning /usr/bin
- Directories can only be removed if they're empty
  - Requires synchronization also
- Most OS cache translations in "namei" cache – maps absolute pathnames to inode
  - Must keep namei cache consistent if files are deleted

Current Directory

- Relative pathnames are resolved relative to current directory
  - Provides default context
  - Every process has one in Unix/Pintos
- chdir(2) changes current directory
  - cd tmp; ls; pwd vs (cd tmp; ls); pwd
- lookup algorithm the same, except starts from current dir
  - process should keep current directory open
  - current directory inherited from parent

Hard & Soft Links

- Provides aliases (different names) for a file
- Hard links: (Unix: ln)
  - Two independent directory entries have the same inode number, refer to same file
  - Inode contains a reference count
  - Disadvantage: alias only possible with same filesystem
- Soft links: (Unix: ln –s)
  - Special type of file (noted in inode); content of file is absolute or relative pathname – stored inside inode instead of direct block list
- Windows: "junctions" & "shortcuts"

Filesystems

Consistency & Logging

Filesystems & Fault Tolerance

- Failure Model
  - Define acceptable failures (disk head hits dust particle, scratches disk – you will lose some data)
  - Define which failure outcomes are unacceptable
- Define *recovery procedure* to deal with unacceptable failures:
  - Recovery moves from an incorrect state A to correct state B
  - Must understand possible incorrect states A after crash!
  - A is like "snapshot of the past"
  - Anticipating all states A is difficult

Methods to Recover from Failure

- On failure, retry entire computation
  - Not a good model for persistent filesystems
- Use atomic changes
  - Problem: how to construct larger atomic changes from the small atomic units available (i.e., single sector writes)
- Use reconstruction
  - Ensure that changes are so ordered that if crash occurs after every step, a recovery program can either undo change or complete it
  - proactive to avoid unacceptable failures
  - reactive to fix up state after acceptable failures

Sensible Invariants

- In a Unix-style filesystem, want that:
  - File & directory names are unique within parent directory
  - Free list/map accounts for all free objects
    - all objects on free list are really free
  - All data blocks belong to exactly one file (only one pointer to them)
  - Inode's ref count reflects exact number of directory entries pointing to it
  - Don't show old data to applications
- Q.: How do we deal with possible violations of these invariants after a crash?

Crash Recovery (fsck)

- After crash, fsck runs and performs the equivalent of mark-and-sweep garbage collection
- Follow, from root directory, directory entries
  - Count how many entries point to inode, adjust ref count
- Recover unreferenced inodes:
  - Scan inode array and check that all inodes marked as used are referenced by dir entry
  - Move others to /lost+found
- Recomputes free list:
  - Follow direct blocks+single+double+triple indirect blocks, mark all blocks so reached as used – free list/map is the complement
- In following discussion, keep in mind what fsck could and could not fix!

Example 1: file create

- On create("foo"), have to
  1. Scan current working dir for entry "foo" (fail if found); else find empty slot in directory for new entry
  2. Allocate an inode #in
  3. Insert pointer to #in in directory: (#in, "foo")
  4. Write a) inode & b) directory back
- What happens if crash after 1, 2, 3, or 4a), 4b)?
- Does order of inode vs directory write back matter?
- Rule: never write persistent pointer to object that's not (yet) persistent

Example 2: file unlink

- To unlink("foo"), must
  1. Find entry "foo" in directory
  2. Remove entry "foo" in directory
  3. Find inode #in corresponding to it, decrement #ref count
  4. If #ref count == 0, free all blocks of file
  5. Write back inode & directory
- Q.: what's the correct order in which to write back inode & directory?
- Q.: what can happen if free blocks are reused before inode's written back?
- Rule: first persistently nullify pointer to any object before freeing it (object=freed blocks & inode)

Example 3: file rename

- To rename("foo", "bar"), must
  1. Find entry (#in, "foo") in directory
  2. Check that "bar" doesn't already exist
  3. Remove entry (#in, "foo")
  4. Add entry (#in, "bar")
- This does not work, because?

Example 3a: file rename

- To rename("foo", "bar"), conservatively
  1. Find entry (#i, "foo") in directory
  2. Check that "bar" doesn't already exist
  3. Increment ref count of #i
  4. Add entry (#i, "bar") to directory
  5. Remove entry (#i, "foo") from directory
  6. Decrement ref count of #i
- Worst case: have old & new names to refer to file
- Rule: never nullify pointer before setting a new pointer

Example 4: file growth

- Suppose file_write() is called.
  - First, find block at offset
- Case 1: metadata already exists for block (file is not grown)
  - Simply write data block
- Case 2: must allocate block, must update metadata (direct block pointer, or indirect block pointer)
  - Must write changed metadata (inode or index block) & data
- Both writeback orders can lead to acceptable failures:
  - File data first, metadata next – may lose some data on crash
  - Metadata first, file data next – may see previous user's deleted data after crash (very expensive to avoid – would require writing all data synchronously)

FFS's Consistency

- Berkeley FFS (Fast File System) formalized rules for filesystem consistency
- FFS acceptable failures:
  - May lose some data on crash
  - May see someone else's previously deleted data
    - Applications must zero data out if they wish to avoid this + fsync
  - May have to spend time to reconstruct free list
  - May find unattached inodes → lost+found
- Unacceptable failures:
  - After crash, get active access to someone else's data
    - Either by pointing at reused inode or reused blocks
- FFS uses 2 synchronous writes on each metadata operation that creates/destroy inodes or directory entries, e.g., creat(), unlink(), mkdir(), rmdir()
  - Updates proceed at disk speed rather than CPU/memory speed