

Slide 1

Administrivia

- Reminder: Homework 5 on Web, due Friday.

Slide 2

Filesystem Reliability — Backups

- Why do backups? sometimes data is more valuable than physical medium, and might need to
 - Recover from disaster (rare these days, but possible).
 - Recover from stupidity (less rare – hence “recycle bin” idea).
- Many issues involved — which files to back up, how to store backup media, etc., etc. — see textbook.

Filesystem Reliability — Consistency Checks

Slide 3

- Can easily happen that true state of filesystem is represented by a combination of what's on disk and what's in memory — a problem if shutdown is not orderly.
- Solution is a “fix-up” program (Unix `fsck`, Windows `scandisk`). Kinds of checking we can do:
 - Consistency check: For each block, how many files does it appear in (treating free list as a file)? If other than 1, problem — fix it as best we can.
 - File consistency check: For each file, count number of links to it and compare with number in its i-node. If not equal, change i-node.
 - Etc., etc. — see text.

Filesystem Performance

Slide 4

- Access to disk data is much slower than access to memory — seek time plus rotational delay plus transfer time.
- So, file systems include various optimizations ...

Slide 5

Improving Filesystem Performance — Caching

- Idea — keep some disk blocks in memory; keep track of which ones are there using hash table (base hash code on device and disk address).
- When cache is full and we must load a new block, which one to replace?
Could use algorithms based on page replacement algorithms, could even do LRU accurately — though that might be wrong (e.g., want to keep data blocks being filled).
- When should blocks be written out?
 - If block is needed for file system consistency, could write out right away. If block hasn't been written out in a while, also could write out, to avoid data loss in long-running program.
 - Two approaches: "Write-through cache" (Windows) — always write out modified blocks right away. Periodic "sync" to write out (Unix).

Slide 6

Improving Filesystem Performance — Block Read-Ahead

- Idea — if file is being read sequentially, can read some blocks "ahead". (Of course, doesn't help if file is being read non-sequentially. Decide based on recent access patterns.)

Improving Filesystem Performance — Reducing Disk Arm Motion

Slide 7

- Group blocks for each file together — easier if bitmap is used to keep track of free space. If not grouped together — “disk fragmentation” may affect performance.
- If i-nodes are being used, place them so they’re fast to get to (and so maybe we can read an i-node and associated file block together).

Disk Fragmentation

Slide 8

- Idea — if blocks that make up a file are (mostly) contiguous, faster to read them all. If not, “disk fragmentation”.
- How likely is disk fragmentation? Depends on filesystem, strategy for allocating space for files.
- “Defragmenter” utility can be run to correct it. Windows comes with one. Linux doesn’t. The claim is that Unix and Linux filesystems typically don’t become fragmented unless the disk is close to full.

Slide 9

Example Filesystem — Unix V7

- Filename restriction — each part of path name at most 14 characters.
- So, directory entry is just 14-byte name and i-node number.
- I-nodes are all stored in a contiguous array at the start of the file system (right after boot block and a “superblock” containing additional parameters).
- What’s in each i-node? attributes (permission bits, numeric owner and group ID, timestamps, links count) and list of blocks — last is pointer to more blocks.
- To find a file:
 - Start with root directory — its i-node is in a known place.
 - Scan directory for first part of path, get its i-node, read it, scan for next part of path, etc.
 - Relative path names are handled by including “.” and “..” in each directory, so no special code needed.

Slide 10

Journaling Filesystems — Overview

- Recall — o/s sometimes doesn’t perform “write to disk” operations right away (caching).
- One result is likely improved performance. Another is potential filesystem inconsistency — operations such as “move a block from the free list to a file” are no longer atomic.
- Idea of journaling filesystem — do something so we *can* regard updates to filesystem as atomic.
- To say it another way — record changes-in-progress in log, when complete mark them “done”.

Journaling Filesystems, Continued

Slide 11

- Can record “data”, “metadata” (directory info, free list, etc.), or both.
- “Undo logging” versus “redo logging”:
 - Undo logging: First copy old data to log, then write new data (possibly many blocks) to disk. If something goes wrong during update, “roll back” by copying old data from log.
 - Redo logging: First write new data to log (i.e., record changes we’re going to make), then write new data to disk. If something goes wrong during update, complete the update using data in log.
- A key benefit — after a system crash, we should only have to look at the log for incomplete updates, rather than doing a full filesystem consistency check.

Journaling Filesystems Versus Log-Structured Filesystems

Slide 12

- Log-structured filesystem — *everything* is written to log, and only to log. Seems like an interesting idea, but tough to implement with good performance.
- Journaling filesystem — log contains only recent and pending updates.

Minute Essay

- List as many reasons as you can think of why there seem to be so many different kinds of filesystems.

Slide 13