Slide 1

Administrivia

- Quiz 6 Friday.
- Next homework coming soon. Likely to combine questions for remaining chapters and be due next Monday.

Slide 2

Filesystems — Quotas

- Why have quotas? Disk space is cheap, right? yes, but more space used means more to back up, and on multi-user systems there are fairness issues, and the possibility that one careless user will affect others.
- Implementation involves keeping track, for each user, of space used versus space allowed. Must be updated every time a file is changed/created/deleted. Some systems allow “grace period”, but eventually all will disallow, for user over quota, creation of new files or expansion of existing files.

Slide 3

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.

Slide 4

Filesystem Reliability — Consistency Checks

- 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.

Slide 5

Journaling Filesystems — Overview

- As we'll discuss later (and as you may know!) — 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".

Slide 6

Journaling Filesystems, Continued

- 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.

Slide 7

Journaling Filesystems Versus Log-Structured Filesystems

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

Slide 8

Filesystem Performance

- 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 9

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 10

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.)

Slide 11

Improving Filesystem Performance — Reducing Disk Arm Motion

- 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).

Slide 12

Disk Fragmentation

- 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 13

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 14

UNIX Filesystems — Hard Links versus Symbolic Links

- As mentioned previously, many filesystems provide a mechanism for creating not-strictly-hierarchical relationships among files/folders. UNIX typically has two:
  - "Hard" links allow multiple directory entries to point to the same i-node.
  - "Soft" (symbolic) links are a special type of file containing a pathname (absolute or relative).
- (Why two? Good question. Compare and contrast . . .)

Slide 15

I/O Management

- Operating system as resource manager — share I/O devices among processes/users.
- Operating system as virtual machine — hide details of interaction with devices, present a nicer interface to application programs.

Slide 16

I/O Hardware, Revisited

- First, a review of I/O hardware — simplified and somewhat abstract view, mostly focusing on how low-level programs communicate with it.
- Many, many kinds of I/O devices — disks, tapes, mice, screens, etc., etc. Can be useful to try to classify as “block devices” versus “character devices”.
- Many/most devices are connected to CPU via a “device controller” that manages low-level details — so o/s talks to controller, not directly to device.
- Interaction between CPU and controllers is via registers in controller (write to tell controller to do something, read to inquire about status), plus (sometimes) data buffer.

  Example — parallel port (connected to printers, etc.) has control register (example bit — linefeed), status register (example bit — busy), data register (one byte of data). These map onto the wires connecting the device to the CPU.

Slide 17

Accessing Device Controller Registers

- Two basic approaches:
  - Define “I/O ports” and access via special instructions.
  - “Memory-mapped I/O” — map some (real) addresses to device-controller registers.

  Some systems use hybrid approach.
- Making either one work requires some hardware complexity, and there are tradeoffs; memory-mapped I/O currently more common.

Slide 18

Direct Memory Access (DMA)

- When reading more than one byte (e.g., from disk), device controller typically reads into internal buffer, checking for errors. How to then transfer to memory?
- One way — CPU makes transfer, byte by byte.
- Another way — DMA controller makes transfer, having been given a target memory location and a count.
- Which is better? consider speed of DMA versus speed of CPU, potential for overlapping data transfer and computation. DMA is extra hardware and could be slower than CPU, but would appear to offer potential to overlap transfer and computation.

Slide 19

Minute Essay

- None — quiz.