Slide 1

Administrivia

- Reminder: Homework 5 programming problems due Wednesday. As mentioned in e-mail, one student discovered that my starter code, which was meant to be 100% standard-conforming (and so would work with an conforming compiler) — wasn't quite.

  One student said she had a similar problem with the starter code for Homework 3 and just fixed the glitch and figured it had to do with using a different system. It did, but not in a way I meant! so I would have been glad to hear about it.

- Homework 6 on the Web. Due in a week.

Slide 2

Minute Essay From Last Lecture

- (Review question, my answer. What I was getting at is this: If the two filesystems don't support quite the same abstraction details — what file names look like, whether there's a notion of file ownership, etc. — then problems seem likely. Most people came reasonably close.)
- Whether spaces are allowed in file names is an issue, all right, but they *are* allowed in UNIX/Linux, just discouraged because they don't "play nice" with typical shells.

Slide 3

Minute Essay From Last Lecture, Continued

- Big-endian versus little-endian could be an issue with binary files, but as far as I know programs that read/write binary files and depend on byte order explicitly say "files may not be portable". Other differences in how files are stored are probably addressed (or not) similarly.

  (On reflection, I'm not sure what an O/S could even do to alleviate such problems: How could it know which files contain data for which this might be a problem?)

- Differences in how filesystems support the abstraction shouldn't matter, though (FAT versus i-nodes, e.g.).

Slide 4

Sidebar: Linux Memory Management — the "OOM Killer"

- (Someone tripped over this last year in doing the first programming problem for Homework 5.)
- Apparently on (some?) Linux systems `malloc` returns true as long as you haven't asked for more memory than you're allowed to have. But it doesn't actually try to find space for the allocated memory (either in real memory or on disk) until it's used — it "overcommits" memory resources.
- So what happens if a process tries to use space that was allocated but not previously used? system tries to find some — and if it can't, it calls the "OOM killer" to terminate one or more processes.
- (My first reaction is "what a bad design?" but it may make sense?)

Slide 5

Example Filesystem — MS-DOS FS

- Filename restriction — eight-character name plus three-character extension. (!) (Textbook doesn't say this, but there are/were ways of faking longer names, basically by mapping longer names into inscrutable short-enough ones.)
- Directory entries contain filename, attributes, timestamp, size, and block number of first block. How are other blocks found? FAT (File Allocation Table).
- Various versions depending on how many bits used to store block number (FAT-12, FAT-16, FAT-32, though the last is apparently really FAT-28). Each defines a set of permitted block sizes, all multiples of 512K.
- Simple, which is good, but imposes limits on file size and partition size. Keeping entire FAT in memory could be a problem if it's big (depends on number of bits used for block number).

Slide 6

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 three are pointers to "single indirect", "double indirect", and "triple indirect" blocks. (See figure 4-33 in textbook.)

Slide 7

Example Filesystem — UNIX V7, Continued

- 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(!).
- Not so simple, and still imposes a limit on total file size, but flexible? and probably requires less system memory, since only i-nodes for open files need to be in memory.

Slide 8

UNIX "Everything's a File"

- UNIX represents a lot of resources as "files" (so that programmers can work with them using familiar(?) mechanisms for accessing files).
- Already mentioned — `/dev` contains "special files" representing I/O devices, real and pretend ("pseudo-terminals").
- Somewhat similar is `/proc`, which presents information about system and all running processes as "files" (but they aren't really). `/sys` (Linux-specific?) is similar.

Slide 9

UNIX Filesystems — Hard Links versus Symbolic Links, Revisited

- 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 10

Minute Essay

- None really — unless questions about filesystems before we move on?