

Slide 1

Administrivia

- Reminder: Homework 5 due Monday.
- Homework 6 on Web. Due next Friday.

Slide 2

Unix Filesystems — Concepts

- (This is summarized from chapter 10, which you can skim if you want more details.)
- Single type of file — sequence of bytes. `lseek` allows random access.
- Single root directory. `mount` allows access to multiple physical devices.
- Links, hard or symbolic, to allow non-tree directory structure.
- Locks to control access to files/records.
- File descriptors for open files.
- “Pipes” — pseudofiles for connecting processes.

Slide 3

Unix Filesystems — Implementation Overview

- Superblock contains critical info — how many i-nodes, location of free list, how many blocks, etc.
- After that? In early implementations, all i-nodes followed by all data blocks. Later implementations (Berkeley FFS) use “cylinder groups”, each containing superblock copy, i-nodes, and data blocks — for better performance, reliability.
- Directory entries fixed-size in early implementations, varying-size in later ones.
- I-nodes contain file attributes, link count, list of (some) blocks, pointers to indirect blocks.
- In memory — table of i-nodes for open files, table of file descriptors (containing, e.g., info about position within file).
- NFS allows access to other systems' disks.

Slide 4

Linux Filesystems — Implementation Overview

- Originally, MINIX filesystem only — similar to early Unix.
- Later, VFS (virtual filesystem) added as intermediate layer to support many kinds.
- ext2/ext3 filesystems (ext3 is ext2 with addition of “journal” file and support for journaling): Similar to FFS, but with a single blocksize, and “block groups” rather than “cylinder groups”. Block group also includes bitmap for free space. Attempts to allocate all space for file within block group (may account for less fragmentation).
Superblock has bit that says whether filesystem is “clean” (no `fsck` needed at boot time).
- `/proc` filesystem represents much system info.

Slide 5

Windows Filesystems — Concepts

- (This is summarized from chapter 11, which you can skim if you want more details.)
- FAT filesystems as described in chapter 6. Designed for small disks and don't work very well for large ones. Also see description of how support for long filenames was added.
- NTFS filesystems — new with Windows NT. Some basic concepts:
 - Unicode filenames.
 - File can consist of multiple “streams” (not just one as in Unix) — generalization of Mac's data fork / resource fork idea.
 - Transparent compression and encryption.

Slide 6

Windows Filesystems — NTFS Implementation

- MFT (master file table) — analogous to i-nodes. One or more entries per file/directory, plus some for system. Boot block points to start.
- For small files, data is right in MFT record. Otherwise contains list of contiguous sequences of bytes.

Journaling Filesystems — Overview

Slide 7

- 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 8

- 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 9

- 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.

Security — Overview

Slide 10

- Goals:
 - Data confidentiality — prevent exposure of data.
 - Data integrity — prevent tampering.
 - System availability — prevent DOS.
- What can go wrong:
 - Deliberate intrusion — from casual snooping to “serious” intrusion.
 - Accidental data loss — “acts of God”, hardware or software error, human error.

User Authentication

Slide 11

- Based on “something the user knows” — e.g., passwords. Problems include where to store them, whether they can be guessed, whether they can be intercepted.
- Based on “something the user has” — e.g., key or smart card. Problems include loss/theft, forgery.
- Based on “something the user is” — biometrics. Problems include inaccuracy/spoofing.

Attacks From Within

Slide 12

- Trojan horses (and how this relates to `$PATH`).
- Login spoofing.
- Logic bombs and trap doors.
- Buffer overflows (and how this relates to, e.g, `gets`).
- And many more ...

Slide 13

Designing a Secure System

- “Security through obscurity” isn’t very.
- Better to give too little access than too much — give programs/people as little as will work.
- Security can’t be an add-on.
- “Keep it simple, stupid.”

Slide 14

Attacks From Outside

- Can categorize as viruses (programs that reproduce themselves when run) and worms (self-replicating) — similar ideas, though.
- Many, many ways such code can get invoked — when legit programs are run, at boot time, when file is opened by some applications (“macro viruses”), etc.
- Also many ways it can spread — once upon a time floppies were vector of choice, now networks or e-mail. Common factors:
 - Executable content from untrustworthy source.
 - Human factors.“Monoculture” makes it easier!
- Virus scanners can check all executables for known viruses (exact or fuzzy matches), but hard/impossible to do this perfectly.
- Better to try to avoid viruses — some nice advice on p. 633.

Slide 15

Safe Execution of "Mobile" Code

- Is there a way to safely execute code from possibly untrustworthy source? Maybe — approaches include sandboxing, interpretation, code signing.
- Example — Java's designed-in security:
 - At source level, very type-safe — no way to use `void*` pointers to access random memory.
 - When classes are loaded, "verifier" checks for potential security problems (not generated by normal compilers, but could be done by hand).
 - At runtime, security manager controls what library routines are called — e.g., applets by default can't do file operations, many kinds of network access.

Slide 16

Trusted Systems

- Is it possible to write a secure O/S? Yes (says Tanenbaum).
- Why isn't that done?
 - People want to run existing code.
 - People prefer (or are presumed to prefer) more features to more security.

Minute Essay

- Over the course of the semester I've told several "war stories" — tales of woe that taught me (or someone) something. Do you have a favorite war story to tell?

Slide 17