Slide 1

Administrivia

- (Via e-mail.)

Slide 2

Files and Filesystems — Overview

- Very abstract view — requirements for long-term information storage are:
  - Store large amounts of information.
  - Have information survive past end of creating process.
  - Allow concurrent access by multiple processes.
- Usual solution — "files" on disk and other external media, organized into "file systems".
- In terms of the two views of an O/S:
  - "Virtual machine" view — filesystem is important abstraction.
  - "Resource manager" view — filesystem manages disk (and other I/O device) resources.

Slide 3

File Abstraction

- Many, many aspects of "file abstraction" — name, type, ownership, etc., etc. Most involve choices/tradeoffs.
- In the following slides, a quick tour of some of the major ones, with some of the possible variations.

Slide 4

File Abstraction, Continued

- File names — always "text string", but some choices: maximum length? case-sensitive? ASCII or Unicode? "extension" required?
- File structure — how file appears to application program:
  - Unstructured sequence of bytes — maximum flexibility, but maybe more work for application.
  - Sequence of fixed-length records — widely used in older systems, not much any more.
  - Tree (or other) structure supporting access by key.

Slide 5

File Abstraction, Continued

- File types — include "regular files", also directories and (in some systems, such as UNIX) "special files". Regular files subdivide into:
  - ASCII files — sequences of ASCII characters, generally separated into lines by line-end character(s).
  - Binary files — everything else, including executables, various archives, MS Word format, etc., etc. Most have some structure, defined by the expectations of the program(s) that work with them — applications for some types, operating system for executables.
- File access — sequential versus random-access (both discussed in textbook).
- File attributes — "other stuff" associated with file (owner, protection info, time of creation / last use, etc.)

Slide 6

File Abstraction, Continued

- File operations (things one can do to a file) include create, delete, open, close, read, write, get attributes, set attributes.
- Many systems also support operations for "memory-mapped files" (read whole file into memory, process there, write back out — as mentioned in previous discussion of virtualizing memory).

Slide 7

Directory/Folder Abstraction

- Basic idea — way of grouping / keeping track of files. Can be
  - Single-level (simple but restrictive).
  - Two-level (almost as simple, better than single-level if multiple users, but also restrictive).
  - Hierarchical.
- Implies need for path names, which can be absolute or relative (to "working directory").
- "Hierarchical" implies a tree structure, but one could include support for something to allow a more-general directed graph (more later). Might be useful as a way to easily share files among users.
- Operations on directories include create, delete, open, close, read, add entry, remove entry, link, unlink.

Slide 8

Linux/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).
- `/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

Multiple File Systems

- Apparently several versions of filesystem abstraction, and more than one implementation of each. Do we have to choose one? no, different types can coexist . . .
- In Windows, having different filesystems on different logical drives managed via drive letters.
- In UNIX/Linux, conceptually a single hierarchy/tree with (tree) root `/`; filesystems added to tree via `mount`, removed via `umount`.

Slide 10

Filesystem Implementation — Overview

- After making decisions about what to implement — how?
- First some more hardware overview . . .

Slide 11

Sidebar: A Little About (Spinning) Disks

- Conceptually, "address space" for disk is a big array of blocks, all the same size.
- Physically often arranged in terms of "surfaces", each consisting of concentric tracks, each made up of sectors.
- Actual reading/writing done via "read/write heads", which can only move in and out (different tracks).
- So accessing data involves seek time (position r/w head), rotational delay (wait for desired sector to spin by), transfer time (actual access).

Slide 12

Filesystem Implementation — Overview Continued

- Basic organization of disk:
  - Master boot record (includes partition table).
  - Partitions, each containing boot block and lots more blocks. Abstract view of access to disk is in terms of reading/writing specified block.
- How to organize/use those "lots more blocks"? Must keep track of which blocks are used by which files, which blocks are free, directory info, file attributes, etc., etc.

  Typically start with superblock containing basic info about filesystem, then some blocks with info about free space and what files are there, then the actual files.

Slide 13

Implementing Files

- One problem is keeping track of which disk blocks belong to which files.
- No surprise — there are several approaches. (All assume some outside "directory"-type structure with some information about each file — a starting block, e.g.)

Slide 14

Implementing Files — Contiguous Allocation

- Key idea — what the name suggests, much like analogous idea for memory management.
- How well does it work? Simple, fast (for both sequential and random access), but can waste space because of external fragmentation.
- Widely used long ago, abandoned, but now maybe useful again.

Slide 15

Implementing Files — Linked-List Allocation

- Key idea — organize each file's blocks as a linked list, with pointer to next block stored within block.
- How well does it work? Fairly simple, sequential access should be fast, but random? But makes good use of space.

Slide 16

Implementing Files — Linked-List Allocation With Table In Memory

- Key idea — keep linked-list scheme, but use table in memory (File Allocation Table or FAT) for pointers rather than using part of disk blocks.
- How well does it work? Same advantages as what was just described, plus random access is fast. Works pretty well for small disks, not so well for large ones (consider table size! and it must be kept in memory).

Implementing Files — I-Nodes

- Key idea — associate with each file a data structure ("index node" or i-node) containing file attributes and disk block numbers, keep in memory for "open" files.

  (This is what the textbook focuses on.)

Slide 17

- How well does it work? Not quite so simple, but relatively fast, and makes good use of space.

Implementing Filesystems — File Attributes

- Another issue is where to keep file "attributes" (owner, timestamps, etc.).
- One way is to keep it in directory.
- Another way is to keep it elsewhere, e.g., in i-node.

Slide 18

Slide 19

Filesystem Implementation — Directories

- Many things to consider here — whether to keep attribute information in directory, whether to make entries fixed or variable size, etc.
- If directory abstraction is basically hierarchical but allows some way of creating a non-tree directed graph, must figure out how to do that. Windows has “shortcuts”; UNIX has “hard links” and “soft links”.

Slide 20

UNIX Filesystems — Hard Links versus Symbolic Links

- “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 21

Filesystem Implementation — Free Space

- One more thing to consider — how to keep track of which blocks are free.
- Two options are “bit map” (one bit per block, free / not free), “free list” (what it sounds like).

Slide 22

Filesystem Performance

- Access to disk data is much slower than access to memory, with seek time being slowest, then rotational delay, then transfer time. (Well, for disks that spin. Solid-state disks don’t, but they may have their own issues, e.g., limits on number of writes?)
- So, file systems include various optimizations . . .

Slide 23

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 24

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 25

Improving Filesystem Performance — Reducing Disk Arm Motion

- The less we have to move the read/write heads around (seek time), the better overall performance will be.
- Drivers for (spinning) disks typically have requests queued. So it makes sense to try to rearrange this queue a little to minimize seek time. Textbook discusses several algorithms.
- Also helps to group blocks for each file together as much as possible.
- And if i-nodes are being used, helps to place them so they're fast to get to (and so maybe we can read an i-node and associated file block together).

Slide 26

Filesystems — What Do Current Systems Use?

- Linux — default is now probably ext4. Other filesystems possible/supported, and support for accessing various Windows filesystems provided via Samba.
- Mac OS X ("macOS"?) — Apple File System, externally pretty UNIX-like, possibly internal differences.
- Windows — NTFS is default, support still provided for FAT-xx.

Slide 27

Minute Essay

- Questions? Does this give you some sense of how O/S's deal with disks and their contents?