Slide 1

Administrivia

- Reminder: Homework 5 (both parts) due Monday. I just made a small revision to written problem 2 to (I hope!) clarify.

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.
- We’ll look first at the user view, then at implementation.

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 3

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 4

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.
- 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. Example program using low-level wrappers for system calls on p. 274.
- 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 memory management).

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

Filesystem Implementation — Overview

- After making decisions about what to implement — how?
- Recall(?) 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.

  (Figure 4-9 in textbook.)
- 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.

  (Figure 4-9 in textbook.)

Slide 9

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 10

Implementing Files — Contiguous Allocation

- Key idea — what the name suggests, much like analogous idea for memory management.
- How well does it work? consider simplicity, speed (both sequential and random access), possibility of fragmentation (wasted space).
- Widely used long ago, abandoned, but now maybe useful again.

Slide 11

Implementing Files — Linked-List Allocation

- Key idea — organize each file's blocks as a linked list, with pointer to next block stored within block.

  (Figure 4-11 in textbook.)

- How well does it work? consider simplicity, speed (both sequential and random access), possibility of fragmentation (wasted space).

Slide 12

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.

  (Figure 4-12 in textbook.)

- How well does it work? consider simplicity, speed (both sequential and random access), possibility of fragmentation (wasted space).

Implementing Files — I-Nodes

Slide 13

- 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.

  (Figure 4-13 in textbook.)

- How well does it work? consider simplicity, speed (both sequential and random access), possibility of fragmentation (wasted space).

Implementing Filesystems — File Attributes

Slide 14

- 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 15

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” (in which different directory entries point to a common structure describing the file) and “soft (symbolic) links” (in which the link is a special type of file).

Slide 16

Virtual File Systems

- Apparently many possibilities for implementing filesystem abstraction, with the usual tradeoffs. Do we have to choose one, or can different types coexist? The latter . . .
- In Windows, having different filesystems on different logical drives is managed via drive letters.
- In UNIX, current approach is usually a “virtual file system” — basically, an extra layer of abstraction (remember the adage about how that can solve any programming problem).

Slide 17

Log-Structured Filesystems

- Log-structured filesystem — *everything* is written to log, and only to log. That sounds impractical, but . . .
- Key idea is that these many disk reads are satified from cache anyway, and lots of small writes to disk give poor performance, so it makes more sense to just write (to cache) a log, and periodically save that to disk.
- Not used much, though, because incompatible with other file systems. Instead . . .

Slide 18

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 19

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. (This can save a *lot* of time!)

Slide 20

Implementing Filesystems — Free Blocks

- Another issue is how to keep track of which blocks are free.
- More than one way . . .

  (Figure 4-22 in textbook.)

Managing Free Space — Free List

- One way to track which blocks are free: list of free blocks, kept on disk.
- How this works:
  - Keep one block of this list in memory.
  - Delete entries when files are created/expanded, add entries when files are deleted.
  - If block becomes empty/full, replace it.

Slide 21

Managing Free Space — Bitmap

- Another way to track which blocks are free: “bitmap” with one bit for each block on disk, also kept on disk.
- How this works:
  - Keep one block of map in memory.
  - Modify entries as for free list.
- Usually requires less space.

Slide 22

Slide 23

Minute Essay

- If you have a system that supports multiple different file systems (such as Linux with Samba to access Windows files), what problems might arise in copying files between different file systems?

  (We had an interesting problem many years ago with backing up `/users` to an OS X machine because the default for OS X filesystems is case-insensitive.)

Slide 24

Minute Essay Answer

- Case sensitivity is one source of potential problems. Other potential problems include restrictions on what characters can appear in filenames and what notion of file ownership and permissions is supported.
- In general, if the two filesystems don't support exactly the same abstraction, problems could arise. It might seem that it could also be a problem if they implement the idea of files in different ways, but a good copy program should be able to cope with that.