Slide 1

Administrivia

- Reminder: Homework 3 due today.
- Reminder: Quiz 4 Monday.
- Homework 4 to be on the Web soon. I will send mail.

Slide 2

Minute Essay From Last Lecture

- Could be trouble if filesystems have different ideas(?) about names, permissions, etc.(?).
- Filesystems might support different extensions? depends on what that means, maybe.
- Path separators (forward versus back slash) might be trouble? (I'm not sure.)
- "Everything's a file" versus not? but then what would that really mean . . .
- Different storage formats? (e.g., NTFS versus EXT3)? though I think if both are sequences of bytes, okay.
- Maximum file size.
- In general, copy program has to be somewhat smart?

Slide 3

Filesystem Implementation — Overview

- Last time we talked about many aspects of filesystem abstraction. 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.
- 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 4

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 5

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 6

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? consider simplicity, speed (both sequential and random access), possibility of fragmentation (wasted space).

Slide 7

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? consider simplicity, speed (both sequential and random access), possibility of fragmentation (wasted space).

Slide 8

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

Filesystem Implementation — Directories

Slide 9

- Many things to consider here — whether to keep attribute information in directory, whether to make entries fixed or variable size, etc.
- Also consider whether to allow some sort of sharing (making the hierarchy a directed graph rather than a tree). Different possibilities here; contrast UNIX “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).

Virtual File Systems

Slide 10

- 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 11

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 12

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 13

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 14

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 15

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 16

Filesystem Performance

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

Slide 17

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 18

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 19

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 20

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 21

Filesystem Implementation — Other Issues

- Quotas: Disk space is cheap and plentiful, but quotas can still be useful on multi-user systems (e.g., to reduce how much trouble one careless user can cause for others).
- Backups: Sometimes data is more valuable than physical medium. Backups useful in recovering from disaster (rare these days but possible) and from “stupidity” (alas, not so rare). Many details; read if interested.
- 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. (Partial) Solution is a “fix-up” program (UNIX `fsck`, Windows `scandisk`).

Slide 22

Minute Essay

- I mentioned that the contiguous-allocation scheme for files had gone out of favor but is now sometimes useful again. What are some circumstances in which it might be a good choice (particular media, particular files, etc.)?
- We’ll do one more class on filesystems. Anything you’d like to hear more about?

Slide 23

Minute Essay Answer

- It could be a good choice for a write-once medium, assuming every files is written all at once (rather than, say, writing some blocks of one file, then some blocks of another, and so forth).
- It might even be a good choice in situations in which faster access outweighs other considerations (such as difficulties in making files bigger, or the potential for fragmentation).