Slide 1

## Administrivia

- Homework 4 graded. I reduced the late penalty to 5% per working day since apparently this is a busy time of year.
- Reminder/change: Homework 5 written problems due today. Programming problems due Wednesday.
- Sample solution to Homework 3 programming problem posted (finally).

Slide 2

## Minute Essay From Last Lecture

- Most people got the basic idea: If the two filesystems don't support exactly the same abstraction, problems could arise in copying.
- A few people also thought it could be a problem if the two filesystems implement the idea of files in different ways (e.g., with i-nodes versus with a FAT), but a well-written copy program should cope with that.

Slide 3

## Homework 4 Essays

- A few people found the problems fairly straightforward (one said they seemed daunting at first but then turned out not to be).
- More, however, found the problems difficult. In truth this kind of puzzles me — to me it seems like the pictures are simple, and once you understand them filling in the details is straightforward. I guess not?

Slide 4

## Sidebar: Linux Memory Management — the “OOM Killer”

- (This can be a problem 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 — i.e., it “overcommits” memory resources.
- So what happens when 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 has its defenders. I’m still skeptical.)

**Slide 5**

## Files and Filesystems — Review/Recap

- Files and filesystems are a key abstraction provided by O/S's.
- Unlike with processes, details of abstraction can vary (e.g., case sensitivity, notions of ownership and permissions).
- Once the details of the abstraction are defined, O/S designer must figure out how to implement it, building on what the hardware supports (typically access to blocks by block number). Many variations possible.

**Slide 6**

## 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 7**

## 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 8**

## 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 9

## 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 10

## 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 11

## Filesystems — Quotas

- Why have quotas? Disk space is cheap, right? yes, but more space used means more to back up, and on multi-user systems there are fairness issues, and the possibility that one careless user will negatively affect others.
- Implementation involves keeping track, for each user, of space used versus space allowed. Must be updated every time a file is changed/created/deleted. Some systems allow “grace period”, but eventually all will disallow, for user over quota, creation of new files or expansion of existing files.

Slide 12

## Filesystem Reliability — Backups

- Why do backups? sometimes data is more valuable than physical medium, and might need to
  - Recover from disaster (rare these days, but possible).
  - Recover from stupidity (less rare – hence “recycle bin” idea).
- Many issues involved: which files to back up, how to store backup media, etc., etc. Discussion in textbook.

Slide 13

## Filesystem Reliability — 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.
- Solution is a "fix-up" program (UNIX `fsck`, Windows `scandisk`). Kinds of checking we can do:
  - Consistency check: For each block, how many files does it appear in (treating free list as a file)? If other than 1, problem — fix it as best we can.
  - File consistency check: For each file, count number of links to it and compare with number in its i-node. If not equal, change i-node.
  - Etc., etc. — see text.

Slide 14

## 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 15

## 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. (Figure 4-33 in textbook.)

Slide 16

## 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(!).

  (Figure 4-34 in textbook.)
- 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 17

## 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 18

## 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 . . . )

## Minute Essay

- If you had trouble with Homework 4, can you say at all what you found so difficult?
- Questions about filesystems before we move on? otherwise just sign in.

Slide 19