Slide 1

Administrivia

- (By 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”.

Slide 3

Files and Filesystems — Overview, Continued

- 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. (Briefly, but the key take-away may be how much more there is here than you might have thought about.)

Slide 4

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 5

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 6

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.)

File Abstraction, Continued

Slide 7

- 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).

Directory/Folder Abstraction

Slide 8

- 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 9

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 10

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 11

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 12

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 13

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).

Slide 14

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.

  (Figure 4-13 in textbook.)

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

Slide 15

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 16

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 17

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 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”.
- A key benefit — after a system crash, only have to look at log for incomplete updates, rather than doing a full filesystem consistency check. (This can save a *lot* of time!)

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.)

Slide 19

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 20

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

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 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

- 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 26

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 27

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 28

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 29

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 30

Example Filesystems

- Textbook describes several filesystems. Normally I talk in lecture about the first two (MS-DOS and UNIX V7).
- But we have limited time, so — review next few slides and skim textbook discussion, please.

Slide 31

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 32

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 33

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 34

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 35

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 . . .)

Slide 36

Filesystems — What Do Current Systems Use?

- Linux — default is now probably ext4, successor to ext2 and ext3 with journalling. Very much like UNIX V7 conceptually, though with support for much longer filenames. 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 37

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 38

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.