Today's plan

- File Systems
- File System implementation
- File System Correctness

File Attributes

- size (and maybe maximum size and/or record size, key offset in record)
- creation time, time of last (read, modify) access
- owner (and maybe group)
- permissions (read/write/execute/visible/system)
- has this file been archived? is it temporary? is it locked?

File Operations

- create, delete (unlink)
- open, close
- read, write, append, seek
- get attributes, set attributes
- rename

Directories (Folders)

- directory (folder) systems are used to structure files
- directories are usually hierarchical when strictly looked at (e.g. by a program), but have some escapes to allow loops (e.g. Unix parent directory ".." and symbolic links, Windows "My computer" and other links)
- often each user has his/her own directory, e.g. /usr/loginname or /home/loginname or /home/mount/loginname
- a directory is usually a special kind of file which may or may not support specific regular file operations, but usually supports operations such as create, delete, open, close, read (one record at a time), rename (a subfile), link (a subfile), unlink (a subfile)
- Unix has hard links (multiple hard links to a file, possibly with different names, are equivalent) and soft links
  - hard links must be to files on the same device
  - when the last hard link to a file is unlinked, the file is removed
    - file attributes include a reference count of links
- a file may have any number of soft links (essentially, the name of the hard link stored in the soft link file), which can work across devices
  - soft links become invalid if the underlying file is removed or renamed

Directory Structures

- very simple systems might only have a single-level (top-level) directory
- most systems allow almost arbitrary depth
- a file name is generally interpreted either relative to a per-process **current (working) directory**, or as an **absolute** path name (beginning with "/" in Unix-like systems)
- some operating systems allow a process to set the root of its file system (**chroot**), which may provide more security if the process is engaging in risky activities (i.e. listening on the network)
- directories are stored in (special) files, usually containing at least the names of the subfiles, pointers to the data, and either the attributes or pointers to the attributes
- which is more efficient: to store all the attributes in the directory, or to have a pointer to the attributes in the directory? Which is simpler?

File System Implementation: which blocks belong to which file?

- contiguous allocation is simple and makes sequential or random access fast, but also requires knowing the file size when allocating and may give fragmentation such that the space is available but not usable (until defragmentation)
- linked list allocation makes sequential access fast, but random access is very slow and blocks end up containing a number n of bytes that is not a power of two.
- linked list allocation in memory keeps a global table with one pointer per block (table is in memory while running, frequently backed up on disk): sequential and random access is fast, block sizes are a power of two, but the memory table may be large (4GB disk with blocks of size 16KB and 4-byte block pointers requires 1MB of RAM)
- **i-node** (index node) allocation keeps a per-file table of blocks, sequentially ordered, and stored on disk until the file is opened. The inode may also store the file attributes, and some of the entries may point to **indirect blocks** which contain more pointers. Fast for sequential and random access, does not use a lot of memory or disk, does not cause fragmentation
- the last two algorithms could be combined, with an inode containing just a pointer to the part of the in-memory table where the file begins

Unix Inodes

- keeping track of which blocks of data belong to which file:
- some versions of Unix used i-nodes with 13 block addresses per inode
- this makes the inode constant size and such that it fits in a block
- if more than 10 data blocks are needed, the 11th pointer is a single indirect block, i.e. points to a block containing addresses of data blocks
- if this is not sufficient, the 12th pointer is a double indirect block, containing the address of a block which contains addresses of blocks of addresses of data blocks
- the last address is a triple-indirect block
- if each indirect block holds the addresses of up to 64 other blocks of 512 bytes each, what is the maximum file size (in bytes) in this system?

Pseudo File Systems

- a device file is really a pseudo file -- it does not correspond to space on disk
- a file system can implement arbitrary access to internal data structures
  - simplest example is the RAM disk
  - a file system can also be used to support access to kernel-internal data structures, e.g. in linux the /proc and sysfs file systems, and in many unix-like systems `/dev/mem` and `/dev/kmem`
- for example, to access the USB on a specific (old) Linux system I had to mount /proc/bus/usb

Implementing Directories

- fixed-size directory entries are simple, but either
  - are too limited in the length of the file name that is supported, or
  - waste too much space
- MS-Dos uses fixed-size (32-byte) directory entries with 8 bytes for the file name and 3 for the extension, 10 unused bytes, and space for size, time, and date
- Unix stores the inode and the filename in variable-sized entries:
  - management of directory entry deletion is more complicated, unless
  - space is defragmented after deletion, which makes deletion from big directories very slow
- Unix directories are stored in files, i.e. inodes point to the blocks on disk holding the directory entries

Management of Disk Space

- similar to (virtual) memory management, e.g block size selection, keeping track of free blocks
- must select a block size
- smaller block sizes waste less space:
  - small files leave most of a block unused
  - even large files typically leave unused about 1/2 of the last block
- larger blocks are faster (for accessing large files or large directories), since a single seek results in reading or writing more data
- median file size reported as 1K (1984) or 12K-15K (1997), so a disk block should not be dramatically larger than this
- Linux tries to allocate blocks nearly sequentially (i.e. as close as possible) to try to improve time to access a large file without taking more space than needed
- the Berkeley Fast File System uses small blocks for small files and large blocks for large files, which is very efficient but more complicated
- free blocks are kept track of as either linked lists or bitmaps

File System Reliability

- any given block may be or may become bad, that is, unable to retain data
- hard disk controllers often
  - are configured with lists of bad blocks
  - may check for bad blocks (corrupted CRCs) during normal accesses
  - have spare tracks, or spare blocks in each track/cylinder used when the CPU tries to access one of the bad blocks
  - seek time for a spare track is usually much larger, for a spare block is usually quite close
- most commonly, controllers only know about bad blocks that are present when the disk leaves the factory
- file system could also build a special file containing all the bad blocks, and could grow this file as more bad blocks are found

Backups

- backups are essential for any data that is hard to replace:
  - to external medium such as tape, CD, or a remote file system (e.g. on another computer)
  - to another disk within the same system, e.g. by **mirroring**
  - with redundant storage such as RAID, where the same data is automatically written to multiple disks, or the data is written to i disks and *check bits* or *check words* are written to a number of other disks
- backups may be incremental, only saving data since the last backup -- this makes backups much fasters and more comprehensive (several snapshots are available going back through time), but restores are harder and slower (when did you create this file?)
- it is a good idea to protect users against user errors, e.g. copy files to a "wastebasket" directory rather than delete them (VMS automatically created backups of every modified file, e.g. with backup numbers ";1", ";2", etc, which could be cleared with a fairly safe purge command)

Bad Block Checking

- the controller usually has indications of errors during reading or writing
  - for example, the CRC may not match the contents of the block
  - the most common response is to try again up to a maximum number of times
- a **read-only** test for a bad block simply reads each block multiple times, and reports the block as bad if the CRC did not verify or if not all reads returned the same result
- a **read-write** test for a bad block saves the content of the block, then writes new patterns on it and tests to make sure they can be read again
- all this requires at least some cooperation from the controller to prevent reading/writing cached versions of the block (which are normally correct)
  - or requires reading/writing more blocks than the controller can cache

Consistency Checking

- usually, when the OS shuts down (or unmounts a file system) it writes a block to disk that says the file system was shut down correctly
- if this block does not have the correct value when mounting the file system, the OS (often automatically) starts a file system consistency checking program (fsck on Unix variants)
- block consistency: each block should be in at most one file exactly once, and otherwise should be listed as a free block
- if the block is not in any file, it is added to the free list if necessary
- if the block is multiple times in the same file, or in multiple files, it can be copied to a free block (as often as necessary), though this normally means the file(s) is/are corrupted
- file consistency: each inode should be in as many directories as its reference count specifies
- a deleted file may not have its reference count set to zero, but may not be in any directory entry, and should be deleted
- a file accessible from multiple directories may have a reference count too low, in which case the reference count is simply set correctly
- further checks can verify that inode numbers are valid, that permissions are reasonable, that directory structures are reasonable, etc

Log-Structured (journaling) File Systems

- with a large disk cache, files can be read-ahead, so latency for file reads can be small
- it is hard to optimize writes, since caching a write for too long increases the likelyhood of inconsistency
- so instead, group all the writes together and write them (as a log segment) when a sufficient number of them has accumulated
- the log segment may contain i-nodes, data blocks, and directory blocks
- finding i-nodes requires searching the entire log to build a map (this is done once), then using the map in memory to locate the inode
- a cleaner daemon (thread, process) searches the log (from the head) to find out which inodes are still in use, and writes these back to the head of the log
- as the cleaner proceeds along the log, less data will be written back for each segment read (assuming files are deleted or blocks overwritten), and the log gets smaller
- any log segments the cleaner has processed become free space and can be reused
- makes performance very good when there are many small writes, and makes consistency checking a lot faster, since only the last segment could be corrupted