File systems

Overview

- DOS FAT: Cute modification to linked list. Links reside in fixed size File Allocation Table (FAT). Still need to do pointer chasing but the entire FAT can fit in memory so chasing cheap.
  - Discussion (Entry size = 16 bits for FAT16, 28 bits for FAT32):
    - What's the maximum size of the FAT? (2^{16} for FAT16, 2^{28} for FAT32)
    - Each entry descibes a "block" that may be made up of multiple contiguous sectors.
    - Given a 2KByte block, what's the maximum size of FS? (2^{16}*2KB=128MB for FAT16, 2^{28}*2KB=512GB for FAT32)
    - Option: go to bigger blocks. FAT allows 2-32KB block sizes. Block size fixed for a disk partition at format time (called "Allocation Unit Size"). Pro? Bigger disk, better spatial locality exploitation. Con? Internal fragmentation, more time for data transfer than strictly needed, buffer cache pollution.
  - Space Overhead of FAT16: 4 bytes/2Kbyte block = approx .2%
  - Reliability: create duplicate copies of FAT to protect against errors
  - Bootstrapping: Where is root directory? Fixed location on disk / have a table in the bootsector (sector 0)
- Indexed Files: Each file has an array holding all it's block pointers. Fill in the block pointers as file grows/shrinks.
  - Pros: Fast random access, easy growth
  - Cons: Slow sequential access, growth beyond index size clumsy. Index size? (filesize/blocksize)*4 (assuming 4 bytes per block pointer).
- Multi-level Indexed Files (BSD/Unix/xv6): Direct blocks, Indirect blocks, Double-indirect blocks. Caters to common case of small files. Handles large files. Example: 4.3 BSD Unix
  - Inode contains 14 4-byte block pointers (initally 0, indicating no block)
  - First 12 point to data blocks
  - Next entry points to an indirect block containing 1024 4-byte block pointers
  - Last entry points to a doubly-indirect block.
  - Maximum file length is fixed, but large.
  - Indirect blocks are not allocated until needed.
  - Common case: small files require only one extra disk access for metadata lookup (inode lookup). Larger files require 2 or 3 metadata lookups.

A design issue is the semantics of a file system operation that requires multiple disk writes. In particular, what happens if the logical update requires writing multiple disks blocks and the power fails during the update? For example, to create a new file, requires allocating an inode (which requires updating the list of free inodes on disk), writing a directory entry to record the allocated i-node under the name of the new file (which may require allocating a new block and updating the directory inode). If the power fails during the operation, the list of free inodes and blocks may be inconsistent with the blocks and inodes in use. Again this is up to implementation of the file system to keep on disk data structures consistent:

- Don't worry about it much, but use a recovery program to bring file system back into a consistent state. Linux ext2 and FAT are such examples. xv6 is almost in this category, except that it has no recovery program.
- Journaling all file system state (kernel-managed structure as well as user-managed data). As we will discuss later, journaling provides atomicity over multiple disk operations. This tends to be quite expensive, in terms of performance. Linux's ext3 has a full data journaling mode that behaves in this fashion.
- Journaling only state that maintains kernel invariants. Never let the file system metadata get into an inconsistent state. In some sense, this resembles how the kernel deals with killing user processes: it cleans up kernel structures, so kernel invariants are upheld, but ignores application invariants. NTFS does this, as does Linux ext3 by default.

Another design issue is the semantics are of concurrent writes to the same data item. What is the order of two updates that happen at the same time? For example, two processes open the same file and write to it. Modern Unix operating systems allow the application to lock a file to get exclusive access. If file locking is not used and if the file descriptor is shared, then the bytes of the two writes will get into the file in some order (this happens often for log files). If the file descriptor is not shared, the end result is not defined. For example, one write may overwrite the other one (e.g., if they are writing to the same part of the file.) Lots of other examples with directories, names, etc.

An implementation issue is performance, because writing to magnetic disk is relatively expensive compared to computing. Three primary ways to improve performance are: careful file system layout and data structure design that induces few seeks (locality, btrees, logging, etc), an in-memory cache of frequently-accessed blocks (or even prefetching), and overlap I/O with computation so that file operations don't have to wait until their completion and so that that the disk driver has more data to write, which allows disk scheduling. (We will talk about performance in detail later.)

xv6 code examples

xv6 implements a minimal Unix file system interface. xv6 doesn't pay attention to file system layout. It overlaps computation and I/O, but doesn't do any disk scheduling. Its cache is write-through, which simplifies keeping on disk data structures consistent, but is bad for performance.

What is xv6's disk layout? Who determines how many inodes, blocks, etc. (mkfs.c). How does xv6 keep track of free blocks and inodes? See balloc()/bfree() and ialloc()/ifree(). Is this layout a good one for performance? What are other options?

- Block 0 is unused
- Block 1 is super block
- Inodes start at block 2
- Free-block bitmap at block `ninodes/IPB + 3`
- Free blocks after that

Free blocks are tracked using bitmap. Free inodes are tracked by linearly scanning the inode array to find one that is free. Other options (bitmap for inodes, better locality among inodes and blocks)

On disk, files are represented by an inode (struct dinode in fs.h), and blocks. Small files have up to 12 block addresses in their inode; large files use the last address in the inode as a disk address for a block with 128 disk addresses (512/4). The size of a file is thus limited to 12 * 512 + 128*512 bytes. What would you change to support larger files? (Ans: e.g., double indirect blocks.)

Directories are files with a bit of structure to them. The file contains of records of the type struct dirent. The entry contains the name for a file (or directory) and its corresponding inode number.

Directory contents: A directory is much like a file, but user can't directly write or read from it. It can only do so using system calls like `readdir()`. Also, the kernel reads the contents to do path resolution (e.g., converting "x/y" to inode number). The content of a "directory file" is an array of dirents (notice that the contents are structured this time).
dirent:

- inum
- 14-byte file name

dirent is free if inum is zero
How many files can appear in a directory? (max file size / sizeof(struct dirent))

xv6's on-disk inode (64 bytes):

- type (free, file, directory, device)
- nlink
- size
- addrs[12+1]
  direct and indirect blocks

Each i-node has an i-number. To turn i-number into inode's block, use 2 + inum/IPB. (IPB = inodes per block). Blocksize=512 inodesize=64, IPB=8

Can view FS as an on-disk data structure. Draw tree: dirs, inodes, blocks. There are two allocation pools: inodes and blocks.

Q: how does xv6 create a file?:

```
$ echo > a
log_write 4 ialloc (44, from create 54)
# allocating an inode for a

log_write 4 iupdate (44, from create 54)
# updating the inode of a

log_write 29 writei (47, from dirlink 48, from create 54)
# adding entry for a in the contents to the current working directory

log_write 2 iupdate
# updating the inode of the current working directory.
```