File systems

Required reading: iread, iwrite, and wdir, and code related to these calls in fs.c, bio.c, ide.c, file.c, and sysfile.c

Overview

The next 3 lectures are about file systems:

- Basic file system implementation
- Naming
- Performance

Users desire to store their data durable so that data survives when the user turns of his computer. The primary media for doing so are: magnetic disks, flash memory, and tapes. We focus on magnetic disks (e.g., through the IDE interface in xv6).

To allow users to remember where they stored a file, they can assign a symbolic name to a file, which appears in a directory.

The data in a file can be organized in a structured way or not. The structured variant is often called a database. UNIX uses the unstructured variant: files are streams of bytes. Any particular structure is likely to be useful to only a small class of applications, and other applications will have to work hard to fit their data into one of the pre-defined structures. Besides, if you want structure, you can easily write a user-mode library program that imposes that format on any file. The end-to-end argument in action. (Databases have special requirements and support an important class of applications, and thus have a specialized plan.)

The API for a minimal file system consists of: open, read, write, seek, close, and stat. Dup duplicates a file descriptor. For example:

```
fd = open("x", O_RDWR);
read (fd, buf, 100);
write (fd, buf, 512);
close (fd)
```

Maintaining the file offset behind the read/write interface is an interesting design decision . The alternative is that the state of a read operation should be maintained by the process doing the reading (i.e., that the pointer should be passed as an argument to read). This argument is compelling in view of the UNIX fork() semantics, which clones a process which shares the file descriptors of its parent. A read by the parent of a shared file descriptor (e.g., stdin, changes the read pointer seen by the child). On the other hand

the alternative would make it difficult to get "(data; ls) > x" right.

Unix API doesn't specify that the effects of write are immediately on the disk before a write returns. It is up to the implementation of the file system within certain bounds. Choices include (that aren't non-exclusive):

- At some point in the future, if the system stays up (e.g., after 30 seconds);
- Before the write returns;
- Before close returns;
- User specified (e.g., before fsync returns).

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.
- Journaling file system. Never let the file system get into an inconsistent state.

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.)

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 that induces few seeks, an in-memory cache of frequently-accessed blocks, 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 keep on disk datastructures consistent, but is bad for performance.

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 files 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. How many files can appear in a directory?

In memory files are represented by struct inode in fsvar.h. What is the role of the additional fields in struct inode?

What is xv6's disk layout? 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?

Let's assume that an application created an empty file x with contains 512 bytes, and that the application now calls read(fd, buf, 100), that is, it is requesting to read 100 bytes into buf. Furthermore, let's assume that the inode for x is is i. Let's pick up what happens by investigating readi(), line 4483.

- 4488-4492: can iread be called on other objects than files? (Yes. For example, read from the keyboard.) Everything is a file in Unix.
- 4495: what does bmap do?
    - 4384: what block is being read?
- 4483: what does bread do? does bread always cause a read to disk?
    - 4006: what does bget do? it implements a simple cache of recently-read disk blocks.
        - How big is the cache? (see param.h)
        - 3972: look if the requested block is in the cache by walking down a circular list.
        - 3977: we had a match.
        - 3979: some other process has "locked" the block, wait until it releases. the other processes releases the block using brelse(). Why lock a block?
            - Atomic read and update. For example, allocating an inode: read block containing inode, mark it allocated, and write it back. This operation must be atomic.
        - 3982: it is ours now.
        - 3987: it is not in the cache; we need to find a cache entry to hold the

block.
        - 3987: what is the cache replacement strategy? (see also brelse())
        - 3988: found an entry that we are going to use.
        - 3989: mark it ours but don't mark it valid (there is no valid data in the entry yet).
    - 4007: if the block was in the cache and the entry has the block's data, return.
    - 4010: if the block wasn't in the cache, read it from disk. are read's synchronous or asynchronous?
        - 3836: a bounded buffer of outstanding disk requests.
        - 3809: tell the disk to move arm and generate an interrupt.
        - 3851: go to sleep and run some other process to run. time sharing in action.
        - 3792: interrupt: arm is in the right position; wakeup requester.
        - 3856: read block from disk.
        - 3860: remove request from bounded buffer. wakeup processes that are waiting for a slot.
        - 3864: start next disk request, if any. xv6 can overlap I/O with computation.
    - 4011: mark the cache entry has holding the data.
- 4498: To where is the block copied? is dst a valid user address?

Now let's suppose that the process is writing 512 bytes at the end of the file a. How many disk writes will happen?

- 4567: allocate a new block
    - 4518: allocate a block: scan block map, and write entry
    - 4523: How many disk operations if the process would have been appending to a large file? (Answer: read indirect block, scan block map, write block map.)
- 4572: read the block that the process will be writing, in case the process writes only part of the block.
- 4574: write it. is it synchronous or asynchronous? (Ans: synchronous but with timesharing.)

Lots of code to implement reading and writing of files. How about directories?

- 4722: look for the directory, reading directory block and see if a directory entry is unused (inum == 0).
- 4729: use it and update it.
- 4735: write the modified block.

Reading and writing of directories is trivial.