Chapter 7

File system data structures

The disk driver and buffer cache (Chapter 6) provide safe, synchronized access to disk blocks. Individual blocks are still a very low-level interface, too raw for most programs. Xv6, following Unix, provides a hierarchical file system that allows programs to treat storage as a tree of named files, each containing a variable length sequence of bytes. The file system is implemented in four steps. The first step is the block allocator. It manages disk blocks, keeping track of which blocks are in use, just as the memory allocator in Chapter 2 tracks which memory pages are in use. The second step is unnamed files called inodes (pronounced i-node). Inodes are a collection of allocated blocks holding a variable length sequence of bytes. The third step is directories. A directory is a special kind of inode whose content is a sequence of directory entries, each of which lists a name and a pointer to another inode. The last step is hierarchical path names like `/usr/rtm/xv6/fs.c`, a convenient syntax for identifying particular files or directories.

File system layout

Xv6 lays out its file system as follows. Block 0 is unused, left available for use by the operating system boot sequence. Block 1 is called the superblock; it contains metadata about the file system. After block 1 comes a sequence of inodes blocks, each containing inode headers. After those come bitmap blocks tracking which data blocks are in use, and then the data blocks themselves.

The header `fs.h` (3150) contains constants and data structures describing the layout of the file system. The superblock contains three numbers: the file system size in blocks, the number of data blocks, and the number of inodes.

Code: Block allocator

The block allocator is made up of the two functions: `balloc` allocates a new disk block and `bfree` frees one. `Balloc` (3704) starts by calling `readsb` to read the superblock. (`Readsb` (3678) is almost trivial: it reads the block, copies the contents into `sb`, and releases the block.) Now that `balloc` knows the number of inodes in the file system, it can consult the in-use bitmaps to find a free data block. The loop (3712) considers every block, starting at block 0 up to `sb.size`, the number of blocks in the file system, checking for a block whose bitmap bit is zero, indicating it is free. If `balloc` finds such a block, it updates the bitmap and returns the block For efficiency, the loop is split into two pieces: the inner loop checks all the bits in a single bitmap block—there are `BPB`—and the outer loop considers all the blocks in increments of `BPB`. There may be multiple processes calling `balloc` simultaneously, and yet there is no explicit locking. Instead, `balloc` relies on the fact that the buffer cache (`bread` and `brelse`) only let one process use a buffer at a time. When reading and writing a bitmap block (3714-3722), `balloc` can be sure that it is the only process in the system using that block.

`Bfree` (3730) is the opposite of `balloc` and has an easier job: there is no search. It finds the right bitmap block, clears the right bit, and is done. Again the exclusive use implied by `bread` and `brelse` avoids the need for explicit locking.

When blocks are loaded in memory, they are referred to by pointers to `buf` structures; as we saw in the last chapter, a more permanent reference is the block's address on disk, its block number.

Inodes

In Unix technical jargon, the term inode refers to an unnamed file in the file system, but the precise meaning can be one of three, depending on context. First, there is the on-disk data structure, which contains metadata about the inode, like its size and the list of blocks storing its data. Second, there is the in-kernel data structure, which contains a copy of the on-disk structure but adds extra metadata needed within the kernel. Third, there is the concept of an inode as the whole unnamed file, including not just the header but also its content, the sequence of bytes in the data blocks. Using the one word to mean all three related ideas can be confusing at first but should become natural.

Inode metadata is stored in an inode structure, and all the inode structures for the file system are packed into a separate section of disk called the inode blocks. Every inode structure is the same size, so it is easy, given a number n, to find the nth inode structure on the disk. In fact, this number n, called the inode number or i-number, is how inodes are identified in the implementation.

The on-disk inode structure is a `struct dinode` (3172). The `type` field in the inode header doubles as an allocation bit: a type of zero means the inode is available for use. The kernel keeps the set of active inodes in memory; its `struct inode` is the in-memory copy of a `struct dinode` on disk. The access rules for in-memory inodes are similar to the rules for buffers in the buffer cache: there is an inode cache, `iget` fetches an inode from the cache, and `iput` releases an inode. Unlike in the buffer cache, `iget` returns an unlocked inode: it is the caller's responsibility to lock the inode with `ilock` before reading or writing metadata or content and then to unlock the inode with `iunlock` before calling `iput`. Leaving locking to the caller allows the file system calls (described in Chapter 8) to manage the atomicity of complex operations. Multiple processes can hold a reference to an inode `ip` returned by `iget` (`ip->ref` counts exactly how many), but only one process can lock the inode at a time.

The inode cache is not a true cache: its only purpose is to synchronize access by multiple processes to shared inodes. It does not actually cache inodes when they are not being used; instead it assumes that the buffer cache is doing a good job of avoiding unnecessary disk acceses and makes no effort to avoid calls to `bread`. The in-memory copy of the inode augments the disk fields with the device and inode number, the reference count mentioned earlier, and a set of flags.

Code: Inodes

To allocate a new inode (for example, when creating a file), xv6 calls `ialloc` (3802). `Ialloc` is similar to `balloc`: it loops over the inode structures on the disk, one block at a time, looking for one that is marked free. When it finds one, it claims it by writing the new `type` to the disk and then returns an entry from the inode cache with the tail call to `iget` (3818). Like in `balloc`, the correct operation of `ialloc` depends on the fact that only one process at a time can be holding a reference to `bp`: `ialloc` can be sure that some other process does not simultaneously see that the inode is available and try to claim it.

`Iget` (3853) looks through the inode cache for an active entry (`ip->ref > 0`) with the desired device and inode number. If it finds one, it returns a new reference to that inode. (3862-3866). As `iget` scans, it records the position of the first empty slot (3867-3868), which it uses if it needs to allocate a new cache entry. In both cases, `iget` returns one reference to the caller: it is the caller's responsibility to call `iput` to release the inode. It can be convenient for some callers to arrange to call `iput` multiple times. `Idup` (3888) increments the reference count so that an additional `iput` call is required before the inode can be dropped from the cache.

Callers must lock the inode using `ilock` before reading or writing its metadata or content. `Ilock` (3902) uses a now-familiar sleep loop to wait for `ip->flag`'s `I_BUSY` bit to be clear and then sets it (3911-3913). Once `ilock` has exclusive access to the inode, it can load the inode metadata from the disk (more likely, the buffer cache) if needed. `Iunlock` (3934) clears the `I_BUSY` bit and wakes any processes sleeping in `ilock`.

`Iput` (3952) releases a reference to an inode by decrementing the reference count (3968). If this is the last reference, so that the count would become zero, the inode is about to become unreachable: its disk data needs to be reclaimed. `Iput` relocks the inode; calls `itrunc` to truncate the file to zero bytes, freeing the data blocks; sets the type to 0 (unallocated); writes the change to disk; and finally unlocks the inode (3955-3967).

The locking protocol in `iput` deserves a closer look. The first part with examining is that when locking `ip`, `iput` simply assumed that it would be unlocked, instead of using a sleep loop. This must be the case, because the caller is required to unlock `ip` before calling `iput`, and the caller has the only reference to it (`ip->ref == 1`). The second part worth examining is that `iput` temporarily releases (3960) and reacquires (3964) the cache lock. This is necessary because `itrunc` and `iupdate` will sleep during disk i/o, but we must consider what might happen while the lock is not held. Specifically, once `iupdate` finishes, the on-disk structure is marked as available for use, and a concurrent call to `ialloc` might find it and reallocate it before `iput` can finish. `Ialloc` will return a reference to the block by calling `iget`, which will find `ip` in the cache, see that its `I_BUSY` flag is set, and sleep. Now the in-core inode is out of sync compared to the disk: `ialloc` reinitialized the disk version but relies on the caller to load it into memory during `ilock`. In order to make sure that this happens, `iput` must clear not only I_BUSY but also I_VALID before releasing the inode lock. It does this by zeroing flags (3965).

Code: Inode contents

The on-disk inode structure, struct dinode, contains a a size and a list of block numbers. The inode data is found in the blocks listed in the dinode's addrs array. The first NDIRECT blocks of data are listed in the first NDIRECT entries in the array; these blocks are called "direct blocks". The next NINDIRECT blocks of data are listed not in the inode but in a data block called the "indirect block". The last entry in the addrs array gives the address of the indirect block. Thus the first 6 kB (NDIRECT×BSIZE) bytes of a file can be loaded from blocks listed in the inode, while the next 64kB (NINDIRECT×BSIZE) bytes can only be loaded after consulting the indirect block. This is a good on-disk representation but a complex one for clients. Bmap manages the representation so that higher-level routines such as readi and writei, which we will see shortly. Bmap returns the disk block number of the bn'th data block for the inode ip. If ip does not have such a block yet, bmap allocates one.

Bmap (4010) begins by picking off the easy case: the first NDIRECT blocks are listed in the inode itself (4015-4019). The next NINDIRECT blocks are litsed in the indirect block at ip->addrs[NDIRECT]. Bmap reads the indirect block (4026) and then reads a block number from the right position within the block (4027). If the block number exceeds NDIRECT+NINDIRECT, bmap panics: callers are responsible for not asking about out-of-range block numbers.

Bmap allocates block as needed. Unallocated blocks are denoted by a block number of zero. As bmap encouters zeros, it replaces them with the numbers of fresh blocks, allocated on demand. (4016-4017, 4024-4025).

Bmap allocates blocks on demand as the inode grows; itrunc frees them, resetting the inode's size to zero. Itrunc (4054) starts by freeing the direct blocks (4060-4065) and then the ones listed in the indirect block (4070-4073), and finally the indirect block itself (4075-4076).

Bmap makes it easy to write functions to access the inode's data stream, like readi and writei. Readi (4102) reads data from the inode. It starts making sure that the offset and count are not reading beyond the end of the file. Reads that start beyond the end of the file return an error (4113-4114) while reads that start at or cross the end of the file return fewer bytes than requested (4115-4116). The main loop processes each block of the file, copying data from the buffer into dst (4118-4123). Writei (4152) is identical to readi, with three exceptions: writes that start at or cross the end of the file grow the file, up to the maximum file size (4165-4166); the loop copies data into the buffers instead of out (4171); and if the write has extended the file, writei must update its size (4176-4179).

Both readi and writei begin by checking for ip->type == T_DEV. This case handles special devices whose data does not live in the file system; we will return to this case in Chapter 8.

Stati (3674) copies inode metadata into the stat structure, which is exposed to user programs via the stat system call (see Chapter 8).

Code: Directories

Xv6 implements a directory as a special kind of file: it has type `T_DEV` and its data is a sequence of directory entries. Each entry is a `struct dirent` (3203), which contains a name and an inode number. The name is at most `DIRSIZ` (14) letters; if shorter, it is terminated by a NUL (0) byte. Directory entries with inode number zero are unallocated.

`Dirlookup` (4212) searches the directory for an entry with the given name. If it finds one, it returns the corresponding inode, unlocked, and sets `*poff` to the byte offset of the entry within the directory, in case the caller wishes to edit it. The outer for loop (4221) considers each block in the directory in turn; the inner loop (4223) considers each directory entry in the block, ignoring entries with inode number zero. If `dirlookup` finds an entry with the right name, it updates `*poff`, releases the block, and returns an unlocked inode obtained via `iget` (4228-4235). `Dirlookup` is the reason that `iget` returns unlocked inodes. The caller has locked `dp`, so if the lookup was for `.`, an alias for the current directory, attempting to lock the inode before returning would try to re-lock `dp` and deadlock. (There are more complicated deadlock scenarios involving multiple processes and `..`, an alias for the parent directory; `.` is not the only problem.) The caller can unlock `dp` and then lock `ip`, ensuring that it only holds one lock at a time.

If `dirlookup` is read, `dirlink` is write. `Dirlink` (4252) writes a new directory entry with the given name and inode number into the directory `dp`. If the name already exists, `dirlink` returns an error (4258-4262). The main loop reads directory entries looking for an unallocated entry. When it finds one, it stops the loop early (4268-4269), with `off` set to the offset of the available entry. Otherwise, the loop ends with `off` set to `dp->size`. Either way, `dirlink` then adds a new entry to the directory by writing at offset `off` (4272-4275).

`Dirlookup` and `dirlink` use different loops to scan the directory: `dirlookup` operates a block at a time, like `balloc` and `ialloc`, while `dirlink` operates one entry at a time by calling `readi`. The latter approach calls `bread` more often—once per entry instead of once per block—but is simpler and makes it easy to exit the loop with `off` set correctly. The more complex loop in `dirlookup` does not save any disk i/o—the buffer cache avoids redundant reads—but doe savoid repeated locking and unlocking of `bcache.lock` in `bread`. The extra work may have been deemed necessary in `dirlooup` but not `dirlink` because the former is so much more common than the latter. (TODO: Make this paragraph into an exercise?)

Path names

The code examined so far implements a hierarchical file system. The earliest Unix systems, such as the version described in Thompson and Ritchie's earliest paper, stops here. Those systems looked up names in the current directory only; to look in another directory, a process needed to first move into that directory. Before long, it became clear that it would be useufl to refer to directories further away: the name xv6/fs.c means first look up xv6, which must be a directory, and then look up fs.c in that directory. A path beginning with a slash is called rooted. The name /xv6/fs.c is like xv6/fs.c but starts the lookup at the root of the file system tree instead of the current directory. Now, decades later, hierarchical, optionally rooted path names are so commonplace that it is easy to forgoet that they had to be invented; Unix did that. (TODO: Is this really true?)

Code: Path names

The final section of fs.c interprets hierarchical path names. Skipelem (4315) helps parse them. It copies the first element of path to name and retrns a pointer to the remainder of path, skipping over leading slashes. Appendix A examines the implementation in detail.

Namei (4389) evaluates path as a hierarchical path name and returns the corresponding inode. Nameiparent is a variant: it stops before the last element, returning the inode of the parent directory and copying the final element into name. Both call the generalized function namex to do the real work.

Namex (4354) starts by deciding where the path evaluation begins. If the path begins with a slash, evaluation begins at the root; otherwise, the current directory (4358-4361). Then it uses skipelem to consider each element of the path in turn (4363). Each iteration of the loop must look up name in the current inode ip. The iteration begins by locking ip and checking that it is a directory. If not, the lookup fails (4364-4368). (Locking ip is necessary not because ip->type can change underfoot—it can't—but because until ilock runs, ip->type is not guaranteed to have been loaded from disk.) If the call is nameiparent and this is the last path element, the loop stops early, as per the definition of nameiparent; the final path element has already been copied into name, so namex need only return the unlocked ip (4369-4373). Finally, the loop looks for the path element using dirlookup and prepares for the next iteration by setting ip.=.next (4374-4379). When the loop runs out of path elements, it returns ip.

TODO: It is possible that namei belongs with all its uses, like open and close, and not here in data structure land.

Real world

Xv6's file system implementation assumes that disk operations are far more expensive than computation. It uses an efficient tree structure on disk but comparatively inefficient linear scans in the inode and buffer cache. The caches are small enough and disk accesses expensive enough to justify this tradeoff. Modern operating systems with larger caches and faster disks use more efficient in-memory data structures. The disk structure, however, with its inodes and direct blocks and indirect blocks, has been remarkably persistent. BSD's UFS/FFS and Linux's ext2/ext3 use essentially the same data structures. The most inefficient part of the file system layout is the directory, which requires a linear scan over all the disk blocks during each lookup. This is reasonable when directories are only a few disk blocks, especially if the entries in each disk block can be kept sorted, but when directories span many disk blocks. Microsoft Windows's NTFS, Mac OS X's HFS, and Solaris's ZFS, just to name a few, implement a directory as an on-disk balanced tree of blocks. This is more complicated than reusing the file implementation but guarantees logarithmic-time directory lookups.

Xv6 is intentionally naive about disk failures: if a disk operation fails, xv6 panics. Whether this is reasonable depends on the hardware: if an operating systems sits atop special hardware that uses redundancy to mask disk failures, perhaps the operating system sees failures so infrequently that panicking is okay. On the other hand, operating systems using plain disks should expect failures and handle them more gracefully, so that the loss of a block in one file doesn't affect the use of the rest of the files system.

Xv6, like most operating systems, requires that the file system fit on one disk device and not change in size. As large databases and multimedia files drive storage requirements ever higher, operating systems are developing ways to eliminate the "one disk per file system" bttleneck. The basic approach is to combine many disks into a single logical disk. Hardware solutions such as RAID are still the most popular, but the current trend is moving toward implementing as much of this logic in software as possible. These software implementations typically allowing rich functionality like growing or shrinking the logical device by adding or removing disks on the fly. Of course, a storage layer that can grow or shrink on the fly requires a file system that can do the same: the fixed-size array of inode blocks used by Unix file systems does not work well in such environments. Separating disk management from the file system may be the cleanest design, but the complex interface between the two has led some systems, like Sun's ZFS, to combine them.

Other features: snapshotting and backup.

Exercises

Exercise: why panic in balloc? Can we recover?

Exercise: why panic in ialloc? Can we recover?

Exercise: inode generation numbers.