Chapter 8

File system

The purpose of a file system is to organize and store data. File systems typically support sharing of data among users and applications, as well as *persistence* so that data is still available after a reboot.

The xv6 file system provides Unix-like files, directories, and pathnames (see Chapter 1), and stores its data on a virtio disk for persistence. The file system addresses several challenges:

- The file system needs on-disk data structures to represent the tree of named directories and files, to record the identities of the blocks that hold each file's content, and to record which areas of the disk are free.
- The file system must support *crash recovery*. That is, if a crash (e.g., power failure) occurs, the file system must still work correctly after a restart. The risk is that a crash might interrupt a sequence of updates and leave inconsistent on-disk data structures (e.g., a block that is both used in a file and marked free).
- Different processes may operate on the file system at the same time, so the file-system code must coordinate to maintain invariants.
- Accessing a disk is orders of magnitude slower than accessing memory, so the file system must maintain an in-memory cache of popular blocks.

The rest of this chapter explains how xv6 addresses these challenges.

8.1 Overview

The xv6 file system implementation is organized in seven layers, shown in Figure 8.1. The disk layer reads and writes blocks on an virtio hard drive. The buffer cache layer caches disk blocks and synchronizes access to them, making sure that only one kernel process at a time can modify the data stored in any particular block. The logging layer allows higher layers to wrap updates to several blocks in a *transaction*, and ensures that the blocks are updated atomically in the face of crashes (i.e., all of them are updated or none). The inode layer provides individual files, each

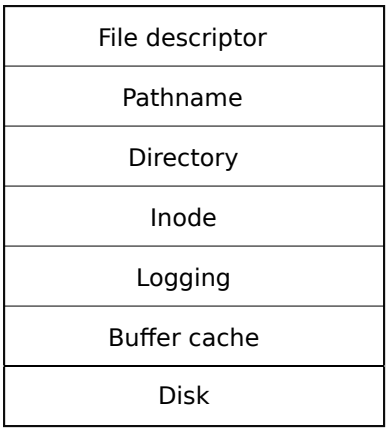

Figure 8.1: Layers of the xv6 file system.

represented as an *inode* with a unique i-number and some blocks holding the file's data. The directory layer implements each directory as a special kind of inode whose content is a sequence of directory entries, each of which contains a file's name and i-number. The pathname layer provides hierarchical path names like `/usr/rtm/xv6/fs.c`, and resolves them with recursive lookup. The file descriptor layer abstracts many Unix resources (e.g., pipes, devices, files, etc.) using the file system interface, simplifying the lives of application programmers.

Disk hardware traditionally presents the data on the disk as a numbered sequence of 512-byte *blocks* (also called *sectors*): sector 0 is the first 512 bytes, sector 1 is the next, and so on. The block size that an operating system uses for its file system maybe different than the sector size that a disk uses, but typically the block size is a multiple of the sector size. Xv6 holds copies of blocks that it has read into memory in objects of type `struct buf` (kernel/buf.h:1). The data stored in this structure is sometimes out of sync with the disk: it might have not yet been read in from disk (the disk is working on it but hasn't returned the sector's content yet), or it might have been updated by software but not yet written to the disk.

The file system must have a plan for where it stores inodes and content blocks on the disk. To do so, xv6 divides the disk into several sections, as Figure 8.2 shows. The file system does not use block 0 (it holds the boot sector). Block 1 is called the *superblock*; it contains metadata about the file system (the file system size in blocks, the number of data blocks, the number of inodes, and the number of blocks in the log). Blocks starting at 2 hold the log. After the log are the inodes, with multiple inodes per block. After those come bitmap blocks tracking which data blocks are in use. The remaining blocks are data blocks; each is either marked free in the bitmap block, or holds content for a file or directory. The superblock is filled in by a separate program, called `mkfs`, which builds an initial file system.

The rest of this chapter discusses each layer, starting with the buffer cache. Look out for situations where well-chosen abstractions at lower layers ease the design of higher ones.

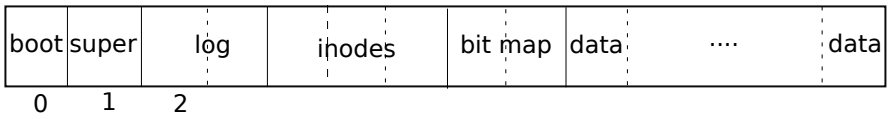

Figure 8.2: Structure of the xv6 file system.

8.2 Buffer cache layer

The buffer cache has two jobs: (1) synchronize access to disk blocks to ensure that only one copy of a block is in memory and that only one kernel thread at a time uses that copy; (2) cache popular blocks so that they don't need to be re-read from the slow disk. The code is in `bio.c`.

The main interface exported by the buffer cache consists of `bread` and `bwrite`; the former obtains a *buf* containing a copy of a block which can be read or modified in memory, and the latter writes a modified buffer to the appropriate block on the disk. A kernel thread must release a buffer by calling `brelse` when it is done with it. The buffer cache uses a per-buffer sleep-lock to ensure that only one thread at a time uses each buffer (and thus each disk block); `bread` returns a locked buffer, and `brelse` releases the lock.

Let's return to the buffer cache. The buffer cache has a fixed number of buffers to hold disk blocks, which means that if the file system asks for a block that is not already in the cache, the buffer cache must recycle a buffer currently holding some other block. The buffer cache recycles the least recently used buffer for the new block. The assumption is that the least recently used buffer is the one least likely to be used again soon.

8.3 Code: Buffer cache

The buffer cache is a doubly-linked list of buffers. The function `binit`, called by `main` (kernel/-main.c:27), initializes the list with the `NBUF` buffers in the static array `buf` (kernel/bio.c:43-52). All other access to the buffer cache refer to the linked list via `bcache.head`, not the `buf` array.

A buffer has two state fields associated with it. The field `valid` indicates that the buffer contains a copy of the block. The field `disk` indicates that the buffer content has been handed to the disk, which may change the buffer (e.g., write data from the disk into `data`).

`Bread` (kernel/bio.c:93) calls `bget` to get a buffer for the given sector (kernel/bio.c:97). If the buffer needs to be read from disk, `bread` calls `virtio_disk_rw` to do that before returning the buffer.

`Bget` (kernel/bio.c:59) scans the buffer list for a buffer with the given device and sector numbers (kernel/bio.c:65-73). If there is such a buffer, `bget` acquires the sleep-lock for the buffer. `Bget` then returns the locked buffer.

If there is no cached buffer for the given sector, `bget` must make one, possibly reusing a buffer that held a different sector. It scans the buffer list a second time, looking for a buffer that is not in use (`b->refcnt = 0`); any such buffer can be used. `Bget` edits the buffer metadata to record the new device and sector number and acquires its sleep-lock. Note that the assignment `b->valid = 0` ensures that `bread` will read the block data from disk rather than incorrectly using the buffer's

previous contents.

It is important that there is at most one cached buffer per disk sector, to ensure that readers see writes, and because the file system uses locks on buffers for synchronization. `Bget` ensures this invariant by holding the `bache.lock` continuously from the first loop's check of whether the block is cached through the second loop's declaration that the block is now cached (by setting `dev`, `blockno`, and `refcnt`). This causes the check for a block's presence and (if not present) the designation of a buffer to hold the block to be atomic.

It is safe for `bget` to acquire the buffer's sleep-lock outside of the `bcache.lock` critical section, since the non-zero `b->refcnt` prevents the buffer from being re-used for a different disk block. The sleep-lock protects reads and writes of the block's buffered content, while the `bcache.lock` protects information about which blocks are cached.

If all the buffers are busy, then too many processes are simultaneously executing file system calls; `bget` panics. A more graceful response might be to sleep until a buffer became free, though there would then be a possibility of deadlock.

Once `bread` has read the disk (if needed) and returned the buffer to its caller, the caller has exclusive use of the buffer and can read or write the data bytes. If the caller does modify the buffer, it must call `bwrite` to write the changed data to disk before releasing the buffer. `Bwrite` (kernel/bio.c:107) calls `virtio_disk_rw` to talk to the disk hardware.

When the caller is done with a buffer, it must call `brelse` to release it. (The name `brelse`, a shortening of b-release, is cryptic but worth learning: it originated in Unix and is used in BSD, Linux, and Solaris too.) `Brelse` (kernel/bio.c:117) releases the sleep-lock and moves the buffer to the front of the linked list (kernel/bio.c:128-133). Moving the buffer causes the list to be ordered by how recently the buffers were used (meaning released): the first buffer in the list is the most recently used, and the last is the least recently used. The two loops in `bget` take advantage of this: the scan for an existing buffer must process the entire list in the worst case, but checking the most recently used buffers first (starting at `bcache.head` and following `next` pointers) will reduce scan time when there is good locality of reference. The scan to pick a buffer to reuse picks the least recently used buffer by scanning backward (following `prev` pointers).

8.4 Logging layer

One of the most interesting problems in file system design is crash recovery. The problem arises because many file-system operations involve multiple writes to the disk, and a crash after a subset of the writes may leave the on-disk file system in an inconsistent state. For example, suppose a crash occurs during file truncation (setting the length of a file to zero and freeing its content blocks). Depending on the order of the disk writes, the crash may either leave an inode with a reference to a content block that is marked free, or it may leave an allocated but unreferenced content block.

The latter is relatively benign, but an inode that refers to a freed block is likely to cause serious problems after a reboot. After reboot, the kernel might allocate that block to another file, and now we have two different files pointing unintentionally to the same block. If xv6 supported multiple users, this situation could be a security problem, since the old file's owner would be able to read

and write blocks in the new file, owned by a different user.

Xv6 solves the problem of crashes during file-system operations with a simple form of logging. An xv6 system call does not directly write the on-disk file system data structures. Instead, it places a description of all the disk writes it wishes to make in a *log* on the disk. Once the system call has logged all of its writes, it writes a special *commit* record to the disk indicating that the log contains a complete operation. At that point the system call copies the writes to the on-disk file system data structures. After those writes have completed, the system call erases the log on disk.

If the system should crash and reboot, the file-system code recovers from the crash as follows, before running any processes. If the log is marked as containing a complete operation, then the recovery code copies the writes to where they belong in the on-disk file system. If the log is not marked as containing a complete operation, the recovery code ignores the log. The recovery code finishes by erasing the log.

Why does xv6's log solve the problem of crashes during file system operations? If the crash occurs before the operation commits, then the log on disk will not be marked as complete, the recovery code will ignore it, and the state of the disk will be as if the operation had not even started. If the crash occurs after the operation commits, then recovery will replay all of the operation's writes, perhaps repeating them if the operation had started to write them to the on-disk data structure. In either case, the log makes operations atomic with respect to crashes: after recovery, either all of the operation's writes appear on the disk, or none of them appear.

8.5 Log design

The log resides at a known fixed location, specified in the superblock. It consists of a header block followed by a sequence of updated block copies ("logged blocks"). The header block contains an array of sector numbers, one for each of the logged blocks, and the count of log blocks. The count in the header block on disk is either zero, indicating that there is no transaction in the log, or non-zero, indicating that the log contains a complete committed transaction with the indicated number of logged blocks. Xv6 writes the header block when a transaction commits, but not before, and sets the count to zero after copying the logged blocks to the file system. Thus a crash midway through a transaction will result in a count of zero in the log's header block; a crash after a commit will result in a non-zero count.

Each system call's code indicates the start and end of the sequence of writes that must be atomic with respect to crashes. To allow concurrent execution of file-system operations by different processes, the logging system can accumulate the writes of multiple system calls into one transaction. Thus a single commit may involve the writes of multiple complete system calls. To avoid splitting a system call across transactions, the logging system only commits when no file-system system calls are underway.

The idea of committing several transactions together is known as *group commit*. Group commit reduces the number of disk operations because it amortizes the fixed cost of a commit over multiple operations. Group commit also hands the disk system more concurrent writes at the same time, perhaps allowing the disk to write them all during a single disk rotation. Xv6's virtio driver doesn't support this kind of *batching*, but xv6's file system design allows for it.

Xv6 dedicates a fixed amount of space on the disk to hold the log. The total number of blocks written by the system calls in a transaction must fit in that space. This has two consequences. No single system call can be allowed to write more distinct blocks than there is space in the log. This is not a problem for most system calls, but two of them can potentially write many blocks: `write` and `unlink`. A large file write may write many data blocks and many bitmap blocks as well as an inode block; unlinking a large file might write many bitmap blocks and an inode. Xv6's write system call breaks up large writes into multiple smaller writes that fit in the log, and `unlink` doesn't cause problems because in practice the xv6 file system uses only one bitmap block. The other consequence of limited log space is that the logging system cannot allow a system call to start unless it is certain that the system call's writes will fit in the space remaining in the log.

8.6 Code: logging

A typical use of the log in a system call looks like this:

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

`begin_op` (kernel/log.c:127) waits until the logging system is not currently committing, and until there is enough unreserved log space to hold the writes from this call. `log.outstanding` counts the number of system calls that have reserved log space; the total reserved space is `log.outstanding` times `MAXOPBLOCKS`. Incrementing `log.outstanding` both reserves space and prevents a commit from occuring during this system call. The code conservatively assumes that each system call might write up to `MAXOPBLOCKS` distinct blocks.

`log_write` (kernel/log.c:215) acts as a proxy for `bwrite`. It records the block's sector number in memory, reserving it a slot in the log on disk, and pins the buffer in the block cache to prevent the block cache from evicting it. The block must stay in the cache until committed: until then, the cached copy is the only record of the modification; it cannot be written to its place on disk until after commit; and other reads in the same transaction must see the modifications. `log_write` notices when a block is written multiple times during a single transaction, and allocates that block the same slot in the log. This optimization is often called *absorption*. It is common that, for example, the disk block containing inodes of several files is written several times within a transaction. By absorbing several disk writes into one, the file system can save log space and can achieve better performance because only one copy of the disk block must be written to disk.

`end_op` (kernel/log.c:147) first decrements the count of outstanding system calls. If the count is now zero, it commits the current transaction by calling `commit()`. There are four stages in this process. `write_log()` (kernel/log.c:179) copies each block modified in the transaction from the buffer cache to its slot in the log on disk. `write_head()` (kernel/log.c:103) writes the header block to disk: this is the commit point, and a crash after the write will result in recovery replaying the

transaction's writes from the log. `install_trans` (kernel/log.c:69) reads each block from the log and writes it to the proper place in the file system. Finally `end_op` writes the log header with a count of zero; this has to happen before the next transaction starts writing logged blocks, so that a crash doesn't result in recovery using one transaction's header with the subsequent transaction's logged blocks.

`recover_from_log` (kernel/log.c:117) is called from `initlog` (kernel/log.c:55), which is called from `fsinit`(kernel/fs.c:42) during boot before the first user process runs (kernel/proc.c:520). It reads the log header, and mimics the actions of `end_op` if the header indicates that the log contains a committed transaction.

An example use of the log occurs in `filewrite` (kernel/file.c:135). The transaction looks like this:

```
begin_op();
ilock(f->ip);
r = writei(f->ip, ...);
iunlock(f->ip);
end_op();
```

This code is wrapped in a loop that breaks up large writes into individual transactions of just a few sectors at a time, to avoid overflowing the log. The call to `writei` writes many blocks as part of this transaction: the file's inode, one or more bitmap blocks, and some data blocks.

8.7 Code: Block allocator

File and directory content is stored in disk blocks, which must be allocated from a free pool. Xv6's block allocator maintains a free bitmap on disk, with one bit per block. A zero bit indicates that the corresponding block is free; a one bit indicates that it is in use. The program `mkfs` sets the bits corresponding to the boot sector, superblock, log blocks, inode blocks, and bitmap blocks.

The block allocator provides two functions: `balloc` allocates a new disk block, and `bfree` frees a block. `Balloc` The loop in `balloc` at (kernel/fs.c:71) considers every block, starting at block 0 up to `sb.size`, the number of blocks in the file system. It looks for a block whose bitmap bit is zero, indicating that 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 outer loop reads each block of bitmap bits. The inner loop checks all Bits-Per-Block (`BPB`) bits in a single bitmap block. The race that might occur if two processes try to allocate a block at the same time is prevented by the fact that the buffer cache only lets one process use any one bitmap block at a time.

`Bfree` (kernel/fs.c:90) finds the right bitmap block and clears the right bit. Again the exclusive use implied by `bread` and `brelse` avoids the need for explicit locking.

As with much of the code described in the remainder of this chapter, `balloc` and `bfree` must be called inside a transaction.

8.8 Inode layer

The term *inode* can have one of two related meanings. It might refer to the on-disk data structure containing a file's size and list of data block numbers. Or "inode" might refer to an in-memory inode, which contains a copy of the on-disk inode as well as extra information needed within the kernel.

The on-disk inodes are packed into a contiguous area of disk called the inode blocks. Every inode is the same size, so it is easy, given a number n, to find the nth inode 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 is defined by a `struct dinode` (kernel/fs.h:32). The `type` field distinguishes between files, directories, and special files (devices). A type of zero indicates that an on-disk inode is free. The `nlink` field counts the number of directory entries that refer to this inode, in order to recognize when the on-disk inode and its data blocks should be freed. The `size` field records the number of bytes of content in the file. The `addrs` array records the block numbers of the disk blocks holding the file's content.

The kernel keeps the set of active inodes in memory in a table called `itable`; `struct inode` (kernel/file.h:17) is the in-memory copy of a `struct dinode` on disk. The kernel stores an inode in memory only if there are C pointers referring to that inode. The `ref` field counts the number of C pointers referring to the in-memory inode, and the kernel discards the inode from memory if the reference count drops to zero. The `iget` and `iput` functions acquire and release pointers to an inode, modifying the reference count. Pointers to an inode can come from file descriptors, current working directories, and transient kernel code such as `exec`.

There are four lock or lock-like mechanisms in xv6's inode code. `itable.lock` protects the invariant that an inode is present in the inode table at most once, and the invariant that an in-memory inode's `ref` field counts the number of in-memory pointers to the inode. Each in-memory inode has a `lock` field containing a sleep-lock, which ensures exclusive access to the inode's fields (such as file length) as well as to the inode's file or directory content blocks. An inode's `ref`, if it is greater than zero, causes the system to maintain the inode in the table, and not re-use the table entry for a different inode. Finally, each inode contains a `nlink` field (on disk and copied in memory if in memory) that counts the number of directory entries that refer to a file; xv6 won't free an inode if its link count is greater than zero.

A `struct inode` pointer returned by `iget()` is guaranteed to be valid until the corresponding call to `iput()`; the inode won't be deleted, and the memory referred to by the pointer won't be re-used for a different inode. `iget()` provides non-exclusive access to an inode, so that there can be many pointers to the same inode. Many parts of the file-system code depend on this behavior of `iget()`, both to hold long-term references to inodes (as open files and current directories) and to prevent races while avoiding deadlock in code that manipulates multiple inodes (such as pathname lookup).

The `struct inode` that `iget` returns may not have any useful content. In order to ensure it holds a copy of the on-disk inode, code must call `ilock`. This locks the inode (so that no other process can `ilock` it) and reads the inode from the disk, if it has not already been read. `iunlock` releases the lock on the inode. Separating acquisition of inode pointers from locking helps avoid deadlock in some situations, for example during directory lookup. Multiple processes can hold a

C pointer to an inode returned by `iget`, but only one process can lock the inode at a time.

The inode table only stores inodes to which kernel code or data structures hold C pointers. Its main job is synchronizing access by multiple processes. The inode table also happens to cache frequently-used inodes, but caching is secondary; if an inode is used frequently, the buffer cache will probably keep it in memory. Code that modifies an in-memory inode writes it to disk with `iupdate`.

8.9 Code: Inodes

To allocate a new inode (for example, when creating a file), xv6 calls `ialloc` (kernel/fs.c:196). `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 table with the tail call to `iget` (kernel/fs.c:210). 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` (kernel/fs.c:243) looks through the inode table 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 (kernel/fs.c:252-256). As `iget` scans, it records the position of the first empty slot (kernel/fs.c:257-258), which it uses if it needs to allocate a table entry.

Code must lock the inode using `ilock` before reading or writing its metadata or content. `Ilock` (kernel/fs.c:289) uses a sleep-lock for this purpose. Once `ilock` has exclusive access to the inode, it reads the inode from disk (more likely, the buffer cache) if needed. The function `iunlock` (kernel/fs.c:317) releases the sleep-lock, which may cause any processes sleeping to be woken up.

`Iput` (kernel/fs.c:333) releases a C pointer to an inode by decrementing the reference count (kernel/fs.c:356). If this is the last reference, the inode's slot in the inode table is now free and can be re-used for a different inode.

If `iput` sees that there are no C pointer references to an inode and that the inode has no links to it (occurs in no directory), then the inode and its data blocks must be freed. `Iput` calls `itrunc` to truncate the file to zero bytes, freeing the data blocks; sets the inode type to 0 (unallocated); and writes the inode to disk (kernel/fs.c:338).

The locking protocol in `iput` in the case in which it frees the inode deserves a closer look. One danger is that a concurrent thread might be waiting in `ilock` to use this inode (e.g., to read a file or list a directory), and won't be prepared to find that the inode is not longer allocated. This can't happen because there is no way for a system call to get a pointer to an in-memory inode if it has no links to it and `ip->ref` is one. That one reference is the reference owned by the thread calling `iput`. It's true that `iput` checks that the reference count is one outside of its `itable.lock` critical section, but at that point the link count is known to be zero, so no thread will try to acquire a new reference. The other main danger is that a concurrent call to `ialloc` might choose the same inode that `iput` is freeing. This can happen only after the `iupdate` writes the disk so that the inode has type zero. This race is benign; the allocating thread will politely wait to acquire the inode's sleep-lock before reading or writing the inode, at which point `iput` is done with it.

`iput()` can write to the disk. This means that any system call that uses the file system may write the disk, because the system call may be the last one having a reference to the file. Even calls like `read()` that appear to be read-only, may end up calling `iput()`. This, in turn, means that even read-only system calls must be wrapped in transactions if they use the file system.

There is a challenging interaction between `iput()` and crashes. `iput()` doesn't truncate a file immediately when the link count for the file drops to zero, because some process might still hold a reference to the inode in memory: a process might still be reading and writing to the file, because it successfully opened it. But, if a crash happens before the last process closes the file descriptor for the file, then the file will be marked allocated on disk but no directory entry will point to it.

File systems handle this case in one of two ways. The simple solution is that on recovery, after reboot, the file system scans the whole file system for files that are marked allocated, but have no directory entry pointing to them. If any such file exists, then it can free those files.

The second solution doesn't require scanning the file system. In this solution, the file system records on disk (e.g., in the super block) the inode inumber of a file whose link count drops to zero but whose reference count isn't zero. If the file system removes the file when its reference counts reaches 0, then it updates the on-disk list by removing that inode from the list. On recovery, the file system frees any file in the list.

Xv6 implements neither solution, which means that inodes may be marked allocated on disk, even though they are not in use anymore. This means that over time xv6 runs the risk that it may run out of disk space.

8.10 Code: Inode content

The on-disk inode structure, `struct dinode`, contains a size and an array of block numbers (see Figure 8.3). 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 12 kB (`NDIRECT` x `BSIZE`) bytes of a file can be loaded from blocks listed in the inode, while the next `256` kB (`NINDIRECT` x `BSIZE`) bytes can only be loaded after consulting the indirect block. This is a good on-disk representation but a complex one for clients. The function `bmap` manages the representation so that higher-level routines, such as `readi` and `writei`, which we will see shortly, do not need to manage this complexity. `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.

The function `bmap` (kernel/fs.c:378) begins by picking off the easy case: the first `NDIRECT` blocks are listed in the inode itself (kernel/fs.c:383-387). The next `NINDIRECT` blocks are listed in the indirect block at `ip->addrs[NDIRECT]`. `Bmap` reads the indirect block (kernel/fs.c:394) and then reads a block number from the right position within the block (kernel/fs.c:395). If the block number exceeds `NDIRECT`+NINDIRECT, `bmap` panics; `writei` contains the check that prevents this from happening (kernel/fs.c:494).

`Bmap` allocates blocks as needed. An `ip->addrs[]` or indirect entry of zero indicates that no block is allocated. As `bmap` encounters zeros, it replaces them with the numbers of fresh blocks,

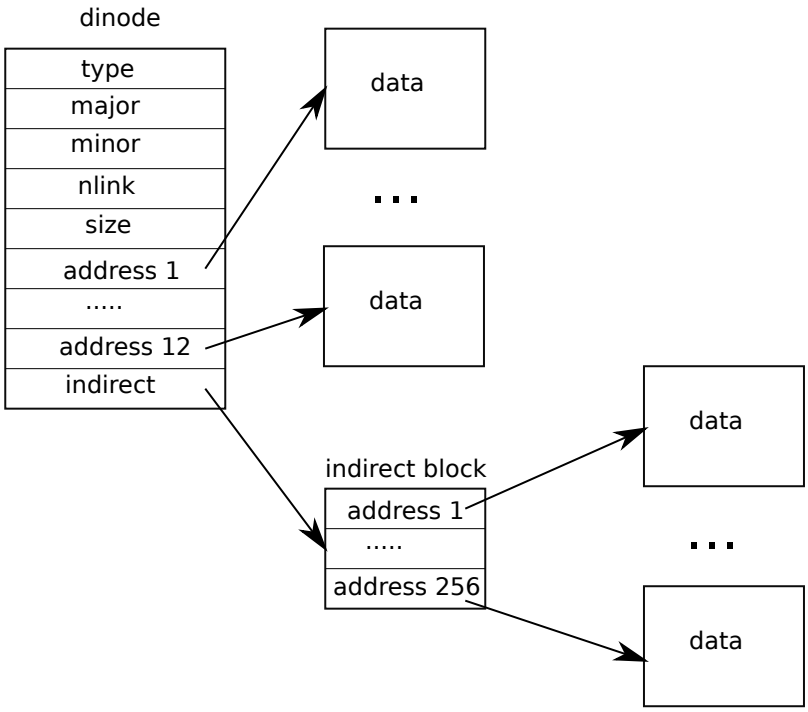

Figure 8.3: The representation of a file on disk.

allocated on demand (kernel/fs.c:384-385) (kernel/fs.c:392-393).

`itrunc` frees a file's blocks, resetting the inode's size to zero. `Itrunc` (kernel/fs.c:410) starts by freeing the direct blocks (kernel/fs.c:416-421), then the ones listed in the indirect block (kernel/fs.c:426-429), and finally the indirect block itself (kernel/fs.c:431-432).

`Bmap` makes it easy for `readi` and `writei` to get at an inode's data. `Readi` (kernel/fs.c:456) starts by making sure that the offset and count are not beyond the end of the file. Reads that start beyond the end of the file return an error (kernel/fs.c:461-462) while reads that start at or cross the end of the file return fewer bytes than requested (kernel/fs.c:463-464). The main loop processes each block of the file, copying data from the buffer into `dst` (kernel/fs.c:466-475). `writei` (kernel/fs.c:487) 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 (kernel/fs.c:494-495); the loop copies data into the buffers instead of out (kernel/fs.c:36); and if the write has extended the file, `writei` must update its size

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 the file descriptor layer.

The function `stati` (kernel/fs.c:442) copies inode metadata into the `stat` structure, which is exposed to user programs via the `stat` system call.

8.11 Code: directory layer

A directory is implemented internally much like a file. Its inode has type `T_DIR` and its data is a sequence of directory entries. Each entry is a `struct dirent` (kernel/fs.h:56), which contains a name and an inode number. The name is at most `DIRSIZ` (14) characters; if shorter, it is terminated by a NUL (0) byte. Directory entries with inode number zero are free.

The function `dirlookup` (kernel/fs.c:530) searches a directory for an entry with the given name. If it finds one, it returns a pointer to 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. If `dirlookup` finds an entry with the right name, it updates `*poff` and returns an unlocked inode obtained via `iget`. `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.

The function `dirlink` (kernel/fs.c:557) 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 (kernel/fs.c:563-567). The main loop reads directory entries looking for an unallocated entry. When it finds one, it stops the loop early (kernel/fs.c:541-542), 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` (kernel/fs.c:577-580).

8.12 Code: Path names

Path name lookup involves a succession of calls to `dirlookup`, one for each path component. `Namei` (kernel/fs.c:664) evaluates `path` and returns the corresponding `inode`. The function `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` (kernel/fs.c:629) starts by deciding where the path evaluation begins. If the path begins with a slash, evaluation begins at the root; otherwise, the current directory (kernel/fs.c:633-636). Then it uses `skipelem` to consider each element of the path in turn (kernel/fs.c:638). 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 (kernel/fs.c:639-643). (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` (kernel/fs.c:644-648). Finally, the loop looks for the path element using `dirlookup` and prepares for the next iteration by setting `ip = next` (kernel/fs.c:649-654). When the loop runs out of path elements, it returns `ip`.

The procedure `namex` may take a long time to complete: it could involve several disk operations to read inodes and directory blocks for the directories traversed in the pathname (if they are not in the buffer cache). Xv6 is carefully designed so that if an invocation of `namex` by one kernel

thread is blocked on a disk I/O, another kernel thread looking up a different pathname can proceed concurrently. `Namex` locks each directory in the path separately so that lookups in different directories can proceed in parallel.

This concurrency introduces some challenges. For example, while one kernel thread is looking up a pathname another kernel thread may be changing the directory tree by unlinking a directory. A potential risk is that a lookup may be searching a directory that has been deleted by another kernel thread and its blocks have been re-used for another directory or file.

Xv6 avoids such races. For example, when executing `dirlookup` in `namex`, the lookup thread holds the lock on the directory and `dirlookup` returns an inode that was obtained using `iget`. `Iget` increases the reference count of the inode. Only after receiving the inode from `dirlookup` does `namex` release the lock on the directory. Now another thread may unlink the inode from the directory but xv6 will not delete the inode yet, because the reference count of the inode is still larger than zero.

Another risk is deadlock. For example, `next` points to the same inode as `ip` when looking up ".". Locking `next` before releasing the lock on `ip` would result in a deadlock. To avoid this deadlock, `namex` unlocks the directory before obtaining a lock on `next`. Here again we see why the separation between `iget` and `ilock` is important.

8.13 File descriptor layer

A cool aspect of the Unix interface is that most resources in Unix are represented as files, including devices such as the console, pipes, and of course, real files. The file descriptor layer is the layer that achieves this uniformity.

Xv6 gives each process its own table of open files, or file descriptors, as we saw in Chapter 1. Each open file is represented by a `struct file` (kernel/file.h:1), which is a wrapper around either an inode or a pipe, plus an I/O offset. Each call to `open` creates a new open file (a new `struct file`): if multiple processes open the same file independently, the different instances will have different I/O offsets. On the other hand, a single open file (the same `struct file`) can appear multiple times in one process's file table and also in the file tables of multiple processes. This would happen if one process used `open` to open the file and then created aliases using `dup` or shared it with a child using `fork`. A reference count tracks the number of references to a particular open file. A file can be open for reading or writing or both. The `readable` and `writable` fields track this.

All the open files in the system are kept in a global file table, the `ftable`. The file table has functions to allocate a file (`filealloc`), create a duplicate reference (`filedup`), release a reference (`fileclose`), and read and write data (`fileread` and `filewrite`).

The first three follow the now-familiar form. `Filealloc` (kernel/file.c:30) scans the file table for an unreferenced file (`f->ref == 0`) and returns a new reference; `filedup` (kernel/file.c:48) increments the reference count; and `fileclose` (kernel/file.c:60) decrements it. When a file's reference count reaches zero, `fileclose` releases the underlying pipe or inode, according to the type.

The functions `filestat`, `fileread`, and `filewrite` implement the `stat`, `read`, and `write` operations on files. `Filestat` (kernel/file.c:88) is only allowed on inodes and calls `stati`. `Fileread`

and `filewrite` check that the operation is allowed by the open mode and then pass the call through to either the pipe or inode implementation. If the file represents an inode, `fileread` and `filewrite` use the I/O offset as the offset for the operation and then advance it (kernel/file.c:122-123) (kernel/file.c:153-154). Pipes have no concept of offset. Recall that the inode functions require the caller to handle locking (kernel/file.c:94-96) (kernel/file.c:121-124) (kernel/file.c:163-166). The inode locking has the convenient side effect that the read and write offsets are updated atomically, so that multiple writing to the same file simultaneously cannot overwrite each other's data, though their writes may end up interlaced.

8.14 Code: System calls

With the functions that the lower layers provide the implementation of most system calls is trivial (see (kernel/sysfile.c)). There are a few calls that deserve a closer look.

The functions `sys_link` and `sys_unlink` edit directories, creating or removing references to inodes. They are another good example of the power of using transactions. `Sys_link` (kernel/sysfile.c:120) begins by fetching its arguments, two strings `old` and `new` (kernel/sysfile.c:125). Assuming `old` exists and is not a directory (kernel/sysfile.c:129-132), `sys_link` increments its `ip->nlink` count. Then `sys_link` calls `nameiparent` to find the parent directory and final path element of `new` (kernel/sysfile.c:145) and creates a new directory entry pointing at `old` 's inode (kernel/sysfile.c:148). The new parent directory must exist and be on the same device as the existing inode: inode numbers only have a unique meaning on a single disk. If an error like this occurs, `sys_link` must go back and decrement `ip->nlink`.

Transactions simplify the implementation because it requires updating multiple disk blocks, but we don't have to worry about the order in which we do them. They either will all succeed or none. For example, without transactions, updating `ip->nlink` before creating a link, would put the file system temporarily in an unsafe state, and a crash in between could result in havoc. With transactions we don't have to worry about this.

`Sys_link` creates a new name for an existing inode. The function `create` (kernel/sysfile.c:242) creates a new name for a new inode. It is a generalization of the three file creation system calls: `open` with the `O_CREATE` flag makes a new ordinary file, `mkdir` makes a new directory, and `mkdev` makes a new device file. Like `sys_link`, `create` starts by calling `nameiparent` to get the inode of the parent directory. It then calls `dirlookup` to check whether the name already exists (kernel/sysfile.c:252). If the name does exist, `create`'s behavior depends on which system call it is being used for: `open` has different semantics from `mkdir` and `mkdev`. If `create` is being used on behalf of `open` (`type == T_FILE`) and the name that exists is itself a regular file, then `open` treats that as a success, so `create` does too (kernel/sysfile.c:256). Otherwise, it is an error (kernel/sysfile.c:257-258). If the name does not already exist, `create` now allocates a new inode with `ialloc` (kernel/sysfile.c:261). If the new inode is a directory, `create` initializes it with `.` and `..` entries. Finally, now that the data is initialized properly, `create` can link it into the parent directory (kernel/sysfile.c:274). `Create`, like `sys_link`, holds two inode locks simultaneously: `ip` and `dp`. There is no possibility of deadlock because the inode `ip` is freshly allocated: no other process in the system will hold `ip` 's lock and then try to lock `dp`.

Using `create`, it is easy to implement `sys_open`, `sys_mkdir`, and `sys_mknod`. `Sys_open` (kernel/sysfile.c:287) is the most complex, because creating a new file is only a small part of what it can do. If `open` is passed the `O_CREATE` flag, it calls `create` (kernel/sysfile.c:301). Otherwise, it calls `namei` (kernel/sysfile.c:307). `Create` returns a locked inode, but `namei` does not, so `sys_open` must lock the inode itself. This provides a convenient place to check that directories are only opened for reading, not writing. Assuming the inode was obtained one way or the other, `sys_open` allocates a file and a file descriptor (kernel/sysfile.c:325) and then fills in the file (kernel/sysfile.c:337-342). Note that no other process can access the partially initialized file since it is only in the current process's table.

Chapter 7 examined the implementation of pipes before we even had a file system. The function `sys_pipe` connects that implementation to the file system by providing a way to create a pipe pair. Its argument is a pointer to space for two integers, where it will record the two new file descriptors. Then it allocates the pipe and installs the file descriptors.

8.15 Real world

The buffer cache in a real-world operating system is significantly more complex than xv6's, but it serves the same two purposes: caching and synchronizing access to the disk. Xv6's buffer cache, like V6's, uses a simple least recently used (LRU) eviction policy; there are many more complex policies that can be implemented, each good for some workloads and not as good for others. A more efficient LRU cache would eliminate the linked list, instead using a hash table for lookups and a heap for LRU evictions. Modern buffer caches are typically integrated with the virtual memory system to support memory-mapped files.

Xv6's logging system is inefficient. A commit cannot occur concurrently with file-system system calls. The system logs entire blocks, even if only a few bytes in a block are changed. It performs synchronous log writes, a block at a time, each of which is likely to require an entire disk rotation time. Real logging systems address all of these problems.

Logging is not the only way to provide crash recovery. Early file systems used a scavenger during reboot (for example, the UNIX `fsck` program) to examine every file and directory and the block and inode free lists, looking for and resolving inconsistencies. Scavenging can take hours for large file systems, and there are situations where it is not possible to resolve inconsistencies in a way that causes the original system calls to be atomic. Recovery from a log is much faster and causes system calls to be atomic in the face of crashes.

Xv6 uses the same basic on-disk layout of inodes and directories as early UNIX; this scheme has been remarkably persistent over the years. 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, but is expensive for directories holding many files. Microsoft Windows's NTFS, macOS's HFS, and Solaris's ZFS, just to name a few, implement a directory as an on-disk balanced tree of blocks. This is complicated but guarantees logarithmic-time directory lookups.

Xv6 is 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 file system.

Xv6 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" bottleneck. 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 allow 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 xv6 would 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.

Xv6's file system lacks many other features of modern file systems; for example, it lacks support for snapshots and incremental backup.

Modern Unix systems allow many kinds of resources to be accessed with the same system calls as on-disk storage: named pipes, network connections, remotely-accessed network file systems, and monitoring and control interfaces such as `/proc`. Instead of xv6's `if` statements in `fileread` and `filewrite`, these systems typically give each open file a table of function pointers, one per operation, and call the function pointer to invoke that inode's implementation of the call. Network file systems and user-level file systems provide functions that turn those calls into network RPCs and wait for the response before returning.

8.16 Exercises

1. Why panic in `balloc` ? Can xv6 recover?
2. Why panic in `ialloc` ? Can xv6 recover?
3. Why doesn't `filealloc` panic when it runs out of files? Why is this more common and therefore worth handling?
4. Suppose the file corresponding to `ip` gets unlinked by another process between `sys_link` 's calls to `iunlock(ip)` and `dirlink`. Will the link be created correctly? Why or why not?
5. `create` makes four function calls (one to `ialloc` and three to `dirlink`) that it requires to succeed. If any doesn't, `create` calls `panic`. Why is this acceptable? Why can't any of those four calls fail?
6. `sys_chdir` calls `iunlock(ip)` before `iput(cp->cwd)`, which might try to lock `cp->cwd`, yet postponing `iunlock(ip)` until after the `iput` would not cause deadlocks. Why not?

7. Implement the `lseek` system call. Supporting `lseek` will also require that you modify `filewrite` to fill holes in the file with zero if `lseek` sets `off` beyond `f->ip->size`.
8. Add `O_TRUNC` and `O_APPEND` to `open`, so that `>` and `>>` operators work in the shell.
9. Modify the file system to support symbolic links.
10. Modify the file system to support named pipes.
11. Modify the file and VM system to support memory-mapped files.