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. 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:490).

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, 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:466-474). writei (kernel/fs.c:483) 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:490-491); 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 (kernel/fs.c:504-511).

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:527) 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:554) 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:560-564). The main loop reads directory entries looking for an unallocated entry. When it finds one, it stops the loop early (kernel/fs.c:538-539), 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:574-577).

8.12 Code: Path names

Path name lookup involves a succession of calls to `dirlookup`, one for each path component. `Namei` (kernel/fs.c:661) 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:626) 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:630-633). Then it uses `skipelem` to consider each element of the path in turn (kernel/fs.c:635). 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:636-640). (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:641-645). Finally, the loop looks for the path element using `dirlookup` and prepares for the next iteration by setting `ip = next` (kernel/fs.c:646-651). 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 caling `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, 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 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 de-

veloping 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.

Chapter 9

Concurrency revisited

Simultaneously obtaining good parallel performance, correctness despite concurrency, and understandable code is a big challenge in kernel design. Straightforward use of locks is the best path to correctness, but is not always possible. This chapter highlights examples in which xv6 is forced to use locks in an involved way, and examples where xv6 uses lock-like techniques but not locks.

9.1 Locking patterns

Cached items are often a challenge to lock. For example, the filesystem's block cache (kernel/bio.c:26) stores copies of up to `NBUF` disk blocks. It's vital that a given disk block have at most one copy in the cache; otherwise, different processes might make conflicting changes to different copies of what ought to be the same block. Each cached block is stored in a `struct buf` (kernel/buf.h:1). A `struct buf` has a lock field which helps ensure that only one process uses a given disk block at a time. However, that lock is not enough: what if a block is not present in the cache at all, and two processes want to use it at the same time? There is no `struct buf` (since the block isn't yet cached), and thus there is nothing to lock. Xv6 deals with this situation by associating an additional lock (`bcache.lock`) with the set of identities of cached blocks. Code that needs to check if a block is cached (e.g., `bget` (kernel/bio.c:59)), or change the set of cached blocks, must hold `bcache.lock`; after that code has found the block and `struct buf` it needs, it can release `bcache.lock` and lock just the specific block. This is a common pattern: one lock for the set of items, plus one lock per item.

Ordinarily the same function that acquires a lock will release it. But a more precise way to view things is that a lock is acquired at the start of a sequence that must appear atomic, and released when that sequence ends. If the sequence starts and ends in different functions, or different threads, or on different CPUs, then the lock acquire and release must do the same. The function of the lock is to force other uses to wait, not to pin a piece of data to a particular agent. One example is the `acquire` in `yield` (kernel/proc.c:515), which is released in the scheduler thread rather than in the acquiring process. Another example is the `acquiresleep` in `ilock` (kernel/fs.c:289); this code often sleeps while reading the disk; it may wake up on a different CPU, which means the lock may be acquired and released on different CPUs.

Freeing an object that is protected by a lock embedded in the object is a delicate business, since owning the lock is not enough to guarantee that freeing would be correct. The problem case arises when some other thread is waiting in `acquire` to use the object; freeing the object implicitly frees the embedded lock, which will cause the waiting thread to malfunction. One solution is to track how many references to the object exist, so that it is only freed when the last reference disappears. See `pipeclose` (kernel/pipe.c:59) for an example; `pi->readopen` and `pi->writeopen` track whether the pipe has file descriptors referring to it.

9.2 Lock-like patterns

In many places xv6 uses a reference count or a flag as a kind of soft lock to indicate that an object is allocated and should not be freed or re-used. A process's `p->state` acts in this way, as do the reference counts in `file`, `inode`, and `buf` structures. While in each case a lock protects the flag or reference count, it is the latter that prevents the object from being prematurely freed.

The file system uses `struct inode` reference counts as a kind of shared lock that can be held by multiple processes, in order to avoid deadlocks that would occur if the code used ordinary locks. For example, the loop in `namex` (kernel/fs.c:626) locks the directory named by each pathname component in turn. However, `namex` must release each lock at the end of the loop, since if it held multiple locks it could deadlock with itself if the pathname included a dot (e.g., `a/./b`). It might also deadlock with a concurrent lookup involving the directory and `..`. As Chapter 8 explains, the solution is for the loop to carry the directory inode over to the next iteration with its reference count incremented, but not locked.

Some data items are protected by different mechanisms at different times, and may at times be protected from concurrent access implicitly by the structure of the xv6 code rather than by explicit locks. For example, when a physical page is free, it is protected by `kmem.lock` (kernel/kalloc.c:24). If the page is then allocated as a pipe (kernel/pipe.c:23), it is protected by a different lock (the embedded `pi->lock`). If the page is re-allocated for a new process's user memory, it is not protected by a lock at all. Instead, the fact that the allocator won't give that page to any other process (until it is freed) protects it from concurrent access. The ownership of a new process's memory is complex: first the parent allocates and manipulates it in `fork`, then the child uses it, and (after the child exits) the parent again owns the memory and passes it to `kfree`. There are two lessons here: a data object may be protected from concurrency in different ways at different points in its lifetime, and the protection may take the form of implicit structure rather than explicit locks.

A final lock-like example is the need to disable interrupts around calls to `mycpu()` (kernel/proc.c:68). Disabling interrupts causes the calling code to be atomic with respect to timer interrupts that could force a context switch, and thus move the process to a different CPU.

9.3 No locks at all

There are a few places where xv6 shares mutable data with no locks at all. One is in the implementation of spinlocks, although one could view the RISC-V atomic instructions as relying on locks