Chapter 8

File system calls

The previous chapter layed described the file system data structures, and how they are used to implemented files and directories. This chapter completes the file system by explaning how the system calls for file operations are implemented. In this chapter, "file" means an open file.

Code: Files

Xv6 gives each process its own table of open files, as we saw in Chapter 0. Each open file is represented by a `struct file` (3650), 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`. Like the inode cache, the file table has a function 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` (4821) scans the file table for an unreferenced file (`f->ref == 0`) and returns a new reference; `filedup` (4839) increments the reference count; and `fileclose` (4852) decrements it. When a file's reference count reaches zero, `fileclose` releases the underlying pipe or inode, according to the type.

`Filestat`, `fileread`, and `filewrite` implement the `stat`, `read`, and `write` operations on files. `Filestat` (4876) 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 (4912-4913, 4932-4933). Pipes have no concept of offset. Remember from Chapter 7 that the inode functions require the caller to handle locking (4879-4881, 4911-4914, 4931-4934). 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.

Code: System calls

Chapter 3 introduced helper functions for implementing system calls: `argint`, `argstr`, and `argptr`. The file system adds another: `argfd` (4963) interprets the `n`th argument as a file descriptor. It calls `argint` to fetch the integer `fd` and then checks that `fd` is a valid file table index. Although `argfd` returns a reference to the file in `*pf`, it does not increment the reference count: the caller shares the reference from the file table. As we will see, this convention avoids reference count operations in most system calls.

The function `fdalloc` (4982) helps manage the current process's file table: it scans the table for an open slot, and if it finds one, inserts `f` and returns the index of the slot, which will serve as the file descriptor. It is up to the caller to manage the reference count.

Finally we are ready to implement system calls. The simplest is `sys_dup` (5001), which makes use of both of these helpers. It calls `argfd` to obtain the file corresponding to the system call argument and then calls `fdalloc` to assign it an additional file descriptor. If both are successful, it calls `filedup` to adjust the reference count: `fdalloc` has created a new reference. Similarly, `sys_close` (5039) obtains a file, removes it from the file table, and releases the reference.

`Sys_read` (5015) parses its arguments as a file descriptor, a pointer, and a size and then calls `fileread`. Note that no reference count operations are necessary: `sys_read` is piggybacking on the reference in the file table. The reference cannot disappear during the `sys_read` because each process has its own file table, and it is impossible for the process to call `sys_close` while it is in the middle of `sys_read`. `Sys_write` (5027) is identical to `sys_read` except that it calls `filewrite`. `Sys_fstat` (5051) is very similar to the previous two.

`Sys_link` and `sys_unlink` edit directories, creating or removing references to inodes. They are another good example of the power of exposing the file system locking to higher-level functions.

`Sys_link` (5063) begins by fetching its arguments, two strings `old` and `new` (5068). Assuming `old` exists and is not a directory (5070-5076), `sys_link` increments its `ip->nlink` count—the number of directories in which it appears—and flushes the new count to disk (5077-5078). Then `sys_link` calls `nameiparent` to find the parent directory and final path element of `new` (5081) and creates a new directory entry pointing at `old`'s inode (5084). 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`.

`Sys_link` would have simpler control flow and error handling if it delayed the increment of `ip->nlink` until it had successfully created the link, but doing this would put the file system temporarily in an unsafe state. The low-level file system code in Chapter 7 was careful not to write out pointers to disk blocks before writing the disk blocks themselves, lest the machine crash with a file system with pointers to old blocks. The same principle is being used here: to avoid dangling pointers, it is important that the link count always be at least as large as the true number of links. If the

system crashed after sys_link creating the second link but before it incremented ip->nlink, then the file system would have an inode with two links but a link count set to one. Removing one of the links would cause the inode to be reused even though there was still a reference to it.

Sys_unlink (5151) is the opposite of sys_link: it removes the named path from the file system. It calls nameiparent to find the parent directory, sysfile.c:/nameiparent.path/, checks that the final element, name, exists in the directory (5170), clears the directory entry (5185), and then updates the link count (5193). As was the case for sys_link, the order here is important: sys_unlink must update the link count only after the directory entry has been removed. There are a few more steps if the entry being removed is a directory: it must be empty (5178) and after it has been removed, the parent directory's link count must be decremented, to reflect that the child's .. entry is gone.

Sys_link creates a new name for an existing inode. Create (5201) 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 (5211). 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 (5215). Otherwise, it is an error (5216-5217). If the name does not already exist, create now allocates a new inode with ialloc (5220). 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 (5233). 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 (5251) 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 (5261). Otherwise, it calls namei (5264). 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 (5273) and then fills in the file (5281-5285). Since we have been so careful to initialize data structures before creating pointers to them, this sequence should feel wrong, but it is safe: no other process can access the partially initialized file since it is only in the current process's table, and these data structures are in memory, not on disk, so they don't persist across a machine crash.

Sys_mkdir (5290) and sys_mknod (5301) are trivial: they parse their arguments, call create, and release the inode it returns.

Sys_chdir (5318) changes the current directory, which is stored as cp->cwd rather than in the file table. It evaluates the new path, checks that it is a directory, releases

the old cp->cwd, and saves the new one in its place.

Chapter 5 examined the implementation of pipes before we even had a file system. 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. Chapter 5 did not examine pipealloc (5571) and pipeclose (5611), but they should be straightforward after walking through the examples above.

The final file system call is exec, which is the topic of the next chapter.

Real world

The file system interface in this chapter has proved remarkably durable: modern systems such as BSD and Linux continue to be based on the same core system calls. In those systems, multiple processes (sometimes called threads) can share a file descriptor table. That introduces another level of locking and complicates the reference counting here.

Xv6 has two different file implementations: pipes and inodes. Modern Unix systems have many: pipes, network connections, and inodes from many different types of file systems, including network file systems. Instead of the 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. Network file systems are now an everyday occurrence, but networking in general is beyond the scope of this book. On the other hand, the World Wide Web is in some ways a global-scale hierarchical file system.

Exercises

Exercise: why doesn't filealloc panic when it runs out of files? Why is this more common and therefore worth handling?

Exercise: 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?

Exercise: 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?

Exercise: 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?