Chapter 3c: xv6 File Abstractions: How Processes Access Files

Gene Cooperman

Contents

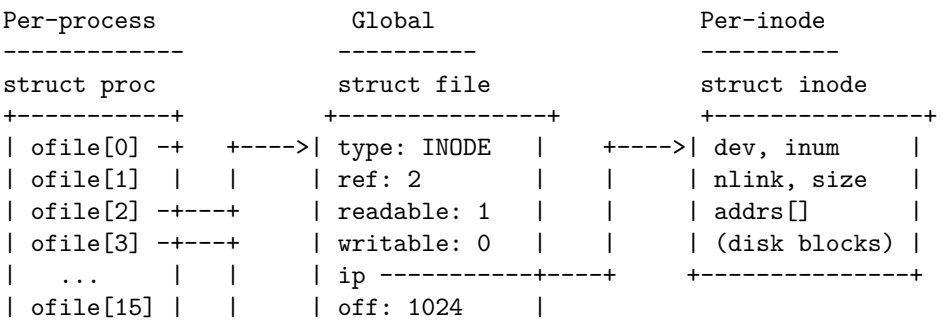
1 xv6 File Abstractions: How Processes Access Files

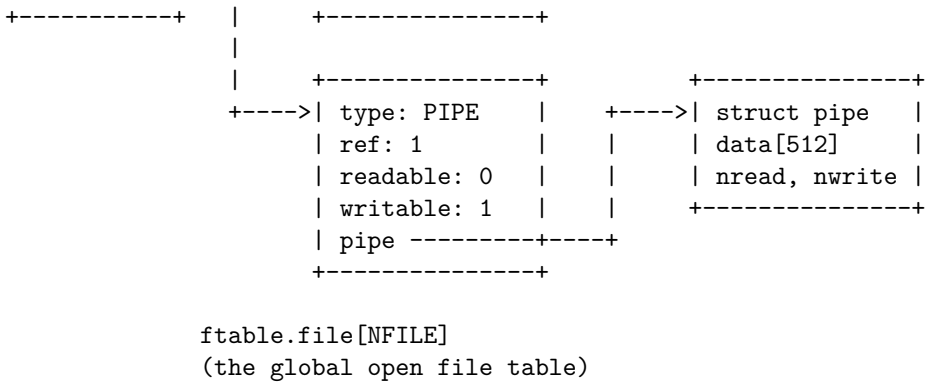
An explanation of how `proc.[ch]`, `file.[ch]`, and `stat.h` work together in xv6 to connect user programs to files, pipes, and devices.

THIS IS A WORK IN PROGRESS.

1.1 1. The Big Picture: Three Layers of Indirection

When a user program does `read(fd, buf, n)`, there are three levels of indirection between the file descriptor integer and the actual data on disk:





This three-layer design exists for good reasons:

- **Multiple file descriptors** in the same process (e.g., after `dup()`) can point to the **same `struct file`**, sharing the same offset.
- **Multiple processes** (e.g., parent and child after `fork()`) can also share the same `struct file`, which is why `struct file` has a reference count (`ref`).
- **Multiple `struct file` entries** can point to the **same `struct inode`** – for example, two independent `open()` calls on the same file produce two `struct file` entries (each with their own offset) but both point to the same inode.

1.2 2. Per-Process File Descriptors: `struct proc` in `proc.h`

Each process has a small array of pointers to open files:

```
struct proc {
  // ... (pid, state, page table, kernel stack, etc.)
  struct file *ofile[NOFILE];  // Open files (NOFILE is typically 16)
  struct inode *cwd;           // Current working directory
  // ...
};
```

The **file descriptor** that user programs work with (the `int fd` returned by `open()`) is simply an **index into `ofile[]`**. When the kernel handles a system call like `read(3, buf, n)`, it does:

```
struct file *f = proc->ofile[3];  // look up fd 3
```

This is why file descriptors are small non-negative integers (0-15 in xv6). File descriptors 0, 1, and 2 are conventionally stdin, stdout, and stderr – but that's only a convention. The kernel doesn't treat them specially; it's the shell (`sh.c`) and `init.c` that set this up.

1.2.1 What `fork()` does

When a process calls `fork()`, the child gets a **copy** of the parent's `ofile[]` array. Each shared `struct file` has its reference count incremented via `filedup()`. This means parent and child share file offsets – if the parent reads 100 bytes, the child's next read starts at byte 100. This is the mechanism behind shell output redirection working across `fork`/`exec`.

1.2.2 What `close()` does

Closing a file descriptor sets `ofile[fd] = NULL` and calls `fileclose()`, which decrements the reference count. Only when the count reaches zero is the `struct file` actually freed.

1.3 3. The Global Open File Table: `ftable` in `file.c`

All open files in the entire system live in a single global table:

```
struct {
  struct spinlock lock;
  struct file file[NFILE];   // NFILE is typically 100
} ftable;
```

The spinlock protects concurrent access (multiple processes may be opening/closing files simultaneously). The functions that manage this table are:

Function	Purpose
`fileinit()`	Initializes the lock at boot (line 19)
`filealloc()`	Scans `ftable` for an unused entry (`ref == 0`), sets `ref = 1`, returns a pointer (line 26)
`filedup(f)`	Increments `f->ref` (line 44)
`fileclose(f)`	Decrements `f->ref`; if zero, releases the underlying pipe or inode (line 56)
`filestat(f, st)`	Fills in a `struct stat` for the file (line 83)
`fileread(f, addr, n)`	Reads from the file – dispatches to `piperead` or `readi` depending on type (line 96)
`filewrite(f, addr, n)`	Writes to the file – dispatches to `pipewrite` or `writei` depending on type (line 117)

1.4 4. `struct file` – the Open File Description

Each entry in the global file table is a `struct file`:

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref;            // reference count
  char readable;      // can this file be read?
  char writable;      // can this file be written?
  struct pipe *pipe;  // FD_PIPE only
  struct inode *ip;   // FD_INODE only
  uint off;           // current read/write offset (FD_INODE only)
};
```

1.4.1 The `type` field: pipes vs. inodes

A `struct file` represents one of two fundamentally different things:

- `FD_INODE` – The file is backed by an inode: a regular file, a directory, or a device. The `ip` field points to a `struct inode`, and the `off` field tracks the current position for sequential reads and writes.
- `FD_PIPE` – The file is one end of a pipe. The `pipe` field points to a `struct pipe` (defined in `pipe.c`), which contains a 512-byte circular buffer (`PIPESIZE`), read/write positions, and a lock. Pipes have no offset or inode – data flows in one direction and is consumed when read.

`FD_NONE` means the slot is unused.

1.4.2 The `readable` and `writable` fields

These two flags control **what operations are permitted** on this open file:

- When you call `open("file", O_RDONLY)`, the kernel sets `readable = 1, writable = 0`.
- When you call `open("file", O_WRONLY)`, it sets `readable = 0, writable = 1`.
- When you call `open("file", O_RDWR)`, both are set to 1.

For **pipes**, these flags are especially important. The `pipe()` system call creates *two* `struct file` entries that both point to the same `struct pipe`:

```
ofile[3] ---> struct file { type: FD_PIPE, readable: 1, writable: 0, pipe: p }
ofile[4] ---> struct file { type: FD_PIPE, readable: 0, writable: 1, pipe: p }
```

One end is read-only (the read end), the other is write-only (the write end). When the shell sets up `cmd1 | cmd2`, it uses `fork()` and `dup()` to give `cmd1` the write end on its stdout, and `cmd2` the read end on its stdin.

The `fileread()` and `filewrite()` functions in `file.c` check these flags before doing any work:

```
int fileread(struct file *f, char *addr, int n) {
  if (f->readable == 0)
    return -1;
  // ... proceed with piperead() or readi()
}
```

1.4.3 The `ref` field: reference counting

As described above, `ref` tracks how many file descriptors (across all processes) currently point to this `struct file`. It is incremented by `filedup()` (called during `fork()` and `dup()`) and decremented by `fileclose()`. The underlying resource (pipe or inode) is only released when `ref` drops to zero.

1.5 5. `struct inode` – the In-Memory Representation of a File

The `struct inode` is defined in `file.h` and represents a file (or directory or device) **as cached in kernel memory**:

```
struct inode {
  uint dev;           // device number
  uint inum;          // inode number (index on disk)
  int ref;            // reference count (in-memory references)
```

```
  int flags;          // I_BUSY, I_VALID

  // Copied from the disk inode when I_VALID is set:
  short type;         // T_DIR, T_FILE, or T_DEV
  short major;        // major device number (T_DEV only)
  short minor;        // minor device number (T_DEV only)
  short nlink;        // number of hard links
  uint size;          // file size in bytes
  uint addrs[NDIRECT+1];  // data block addresses (direct + 1 indirect)
};
```

1.5.1 Key points about `struct inode`

It is the kernel's internal working copy. Inodes exist on disk (as `struct dinode` in `fs.h`), but the kernel reads them into memory as `struct inode` and caches them for efficiency. The `I_VALID` flag indicates whether the in-memory fields have been loaded from disk yet.

It has its own reference count, separate from `struct file`'s `ref`. Multiple `struct file` entries can point to the same inode (two independent opens of the same file). The inode's `ref` counts how many in-memory references exist; only when it drops to zero can the inode be evicted from the cache.

The `I_BUSY` flag acts as a per-inode lock. A process sets `I_BUSY` before modifying the inode and clears it when done, sleeping if another process already holds it.

The `type` field distinguishes:

Type	Value	Meaning
`T_DIR`	1	Directory
`T_FILE`	2	Regular file
`T_DEV`	3	Device (e.g., console)

For devices (`T_DEV`), the `major` and `minor` numbers index into the **`devsw[]` table** (defined in `file.c`), which holds function pointers for device-specific read and write operations:

```
struct devsw {
  int (*read)(struct inode*, char*, int);
  int (*write)(struct inode*, char*, int);
};
struct devsw devsw[NDEV];
```

This is how the console (keyboard input / screen output) is accessed through the same `read()`/`write()` interface as regular files.

1.6 6. `struct stat` – Information Returned to User Space

The `struct stat` is defined in `stat.h`:

```
struct stat {
  short type;    // T_DIR, T_FILE, or T_DEV
  int dev;       // device number
```

```
  uint ino;      // inode number
  short nlink;   // number of hard links
  uint size;     // file size in bytes
};
```

1.6.1 How `struct stat` differs from `struct inode`

Although the fields overlap, these two structs serve very different purposes:

	`struct inode`	`struct stat`
Where it lives	Kernel memory only	Copied to user space
Who uses it	Kernel code (fs.c, file.c, etc.)	User programs (via `fstat()`)
What it contains	Full operational data: block addresses (`addrs[]`), lock flags (`I_BUSY`), cache validity (`I_VALID`)	A **safe subset** of metadata – just what user programs need to know
Lifetime	Persists in the inode cache as long as referenced	Transient snapshot, filled on demand

The kernel **intentionally does not expose** internal fields to user space. The user has no business knowing about disk block addresses or kernel lock state. `struct stat` provides just the identity (device + inode number), type, link count, and size.

1.6.2 How `fstat()` connects them

When a user program calls `fstat(fd, &st)`, the kernel calls `filestat()` in `file.c` (line 83), which does:

```
int filestat(struct file *f, struct stat *st) {
  if (f->type == FD_INODE) {
    ilock(f->ip);
    stati(f->ip, st);   // copy selected fields from inode to stat
    iunlock(f->ip);
    return 0;
  }
  return -1;
}
```

The `stati()` function in `fs.c` copies the relevant fields:

```
void stati(struct inode *ip, struct stat *st) {
  st->dev  = ip->dev;
  st->ino  = ip->inum;
  st->type = ip->type;
  st->nlink = ip->nlink;
  st->size = ip->size;
}
```

Note that `fstat()` returns -1 for pipes – pipes have no inode, no size, and no disk presence, so there's nothing meaningful to report.

1.7 7. Putting It All Together: The Complete Chain

Here is what connects a user's file descriptor to data on disk:

```
User program       Kernel (per-process)     Kernel (global)        Kernel (inode cache)   Disk
------------      ----------------------    ---------------       ---------------------  ------
int fd = 3  --->  proc->ofile[3]  -------> ftable.file[k]  ---->  inode (dev=1,inum=42)  --> dinode #42
                   (just an index)          (struct file:          (struct inode:              (struct di
                                             type, ref,             type, size,                 on-disk
                                             readable,              addrs[],
                                             writable,              nlink, ...)
                                             ip, off)

fstat(fd, &st)  <-------------------------------------------------------  stati(ip, st)
                      struct stat { type, dev, ino, nlink, size }
                      (copied to user space -- safe subset only)
```

1.7.1 Example: `cat < input.txt | wc`

In this pipeline, the shell sets up:

1. `open("input.txt", O_RDONLY)` -> creates a `struct file` with `type = FD_INODE`, `readable = 1`, `writable = 0`, pointing to the inode for `input.txt`
2. `pipe(fds)` -> creates **two** `struct file` entries pointing to the same `struct pipe`, one readable, one writable
3. For `cat`: `dup` the input.txt file onto fd 0 (stdin), `dup` the pipe write end onto fd 1 (stdout)
4. For `wc`: `dup` the pipe read end onto fd 0 (stdin)

Both `cat` and `wc` just call `read(0, ...)` and `write(1, ...)` – they don't know or care whether they're talking to a file or a pipe. The `struct file` type field handles the dispatch transparently in `fileread()` and `filewrite()`.

1.8 8. A Note on Later UNIX Systems

xv6 is based on the original Sixth Edition UNIX (V6), and is deliberately simplified for teaching. One notable simplification is the **absence of file permissions**.

In xv6, `struct inode` and `struct stat` have **no permission fields**. Any process can read, write, or execute any file. There is also no concept of file ownership (no `uid` or `gid` fields).

In later UNIX systems (and all modern systems including Linux), both the on-disk inode and `struct stat` include additional fields:

```c
// Typical fields found in later UNIX but NOT in xv6:
struct stat {
  // ... (the fields xv6 has, plus:)
  uid_t st_uid;     // owner user ID
  gid_t st_gid;     // owner group ID
  mode_t st_mode;   // file type AND permissions (rwxrwxrwx)
  time_t st_atime;  // last access time
  time_t st_mtime;  // last modification time
```

```
  time_t st_ctime;  // last status change time
};
```

The `st_mode` field encodes the classic UNIX permission bits:

- **Owner** permissions: read, write, execute
- **Group** permissions: read, write, execute
- **Other** permissions: read, write, execute

These permissions are checked by the kernel on every `open()`, `exec()`, and directory traversal. The kernel compares the calling process's `uid`/`gid` against the file's owner and group to decide which set of permission bits applies.

xv6 omits all of this to keep the code small and focused on the core concepts of processes, virtual memory, and file systems – but it's important to know that any real-world UNIX system enforces access control at this level.

1.9 Summary

Concept	Struct	Defined in	Purpose
File descriptor	`int` (index into `ofile[]`)	`proc.h`	Per-process handle for an open file
Open file description	`struct file`	`file.h` / `file.c`	Shared state: type, ref count, permissions, offset, pointer to pipe or inode
In-memory inode	`struct inode`	`file.h`	Kernel's cached copy of on-disk file metadata and block locations
User-visible file info	`struct stat`	`stat.h`	Safe subset of inode metadata, copied to user space by `fstat()`
Pipe	`struct pipe`	`pipe.c`	512-byte circular buffer connecting a reader and a writer
Device dispatch	`struct devsw`	`file.h` / `file.c`	Function pointers for device-specific read/write

---

1.10 9. A Closer Look at `ref` and `off` in `struct file`

Two fields in `struct file` deserve special attention, since they are the source of many subtle behaviors that students encounter in practice.

1.10.1 Reference counting with `ref`

The `ref` field tracks how many file descriptors – across all processes in the system – currently point to this `struct file` entry. It works as follows:

- **`filealloc()`** finds an unused slot (where `ref == 0`), sets `ref = 1`, and returns it. This happens during `open()` and `pipe()`.
- **`filedup(f)`** increments `f->ref` by 1. This happens during `dup()` and during `fork()` (which duplicates every entry in the parent's `ofile[]` for the child).
- **`fileclose(f)`** decrements `f->ref` by 1. If `ref` reaches zero, the `struct file` entry is truly released: for `FD_PIPE` files, it calls `pipeclose()`; for `FD_INODE` files, it releases the inode reference. If `ref` is still positive, the entry remains alive for the other file descriptors still using it.

This means a `struct file` persists as long as *anyone* still has a descriptor pointing to it. For example, after `fork()`, both parent and child share the same `struct file` (with `ref == 2`). If the parent closes its descriptor, `ref` drops to 1 – the child can still read and write normally. Only when the child also closes does `ref` reach 0 and the underlying resource get freed.

1.10.2 The file offset (`off`) and its sharing semantics

The `off` field records the current byte position for the next `read()` or `write()` on an `FD_INODE` file. It starts at 0 and advances as data is read or written. The critical point is that **`off` lives in the `struct file`, not in the per-process `ofile[]` array**. This has important consequences.

1.10.2.1 Case 1: `dup()` – shared offset

When a process calls `dup(fd)`, the kernel allocates a new file descriptor (a new slot in `ofile[]`) but points it at the **same `struct file`**. No new `struct file` is created. This means:

```
ofile[3] ---+
            +---> struct file { ref: 2, off: 0, ip: ... }
ofile[4] ---+
```

Now `read(3, buf, 100)` advances `off` to 100. A subsequent `read(4, buf, 50)` starts at byte 100 (not 0) and advances `off` to 150. The two descriptors **share the same offset** because they share the same `struct file`. The same sharing happens after `fork()` – parent and child share offsets, which is why output from both processes doesn't overwrite itself when writing to the same file.

1.10.2.2 Case 2: Two independent `open()` calls – independent offsets

If a process calls `open()` twice on the same file:

```
int fd1 = open("data.txt", O_RDONLY);   // creates struct file A, off = 0
int fd2 = open("data.txt", O_RDONLY);   // creates struct file B, off = 0
```

This produces **two separate `struct file` entries**, each with its own `off` field, even though both point to the same underlying `struct inode`:

```
ofile[3] ---> struct file A { ref: 1, off: 0, ip --+
                                                   +--> inode (data.txt)
ofile[4] ---> struct file B { ref: 1, off: 0, ip --+
```

Now `read(3, buf, 100)` advances file A's offset to 100, but file B's offset remains at 0. A subsequent `read(4, buf, 50)` reads from the **beginning** of the file, because fd 4 has its own independent offset. The two descriptors operate completely independently, even though they refer to the same file on disk.

1.10.2.3 Why this matters

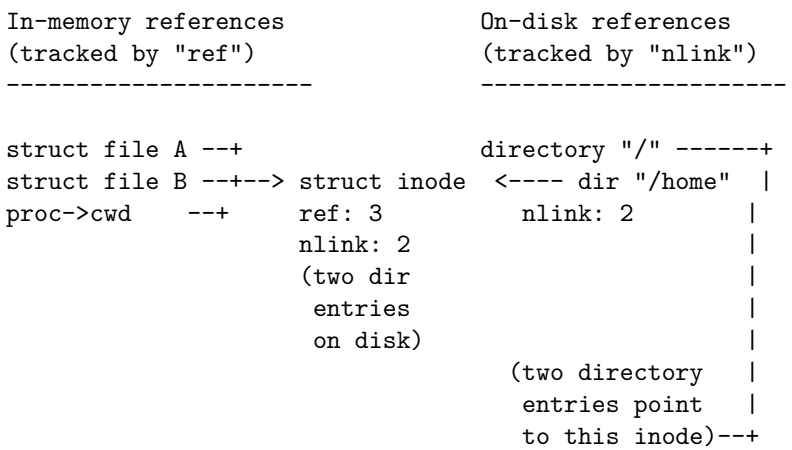
This distinction – shared `struct file` (via `dup`/`fork`) vs. independent `struct file` entries (via separate `open` calls) – explains many behaviors that would otherwise seem mysterious:

- **Shell redirection** (`> output.txt`) works correctly across `fork`/`exec` because parent and child share the offset, so their writes don't collide.
- **Two programs independently reading the same file** each get their own offset and can read at their own pace, because each called `open()` separately.
- **The `ref` count** ensures the `struct file` (and its offset) stays alive as long as anyone needs it, and is cleaned up exactly when the last descriptor is closed.

1.11 10. A Deeper Look at `struct inode`: `ref`, `nlink`, and File Types

The `struct inode` has two different reference-counting fields – `ref` and `nlink` – that serve complementary purposes. Understanding the distinction is essential to understanding how xv6 manages files and reclaims storage.

1.11.1 Two kinds of reference counting



1.11.1.1 The `ref` field: in-memory (kernel) references

The `ref` field counts how many **in-memory pointers** currently refer to this `struct inode` in kernel data structures. These references come from:

- **`struct file` entries** – every open file with `type == FD_INODE` holds a pointer `ip` to an inode
- **`proc->cwd`** – every process has a pointer to the inode of its current working directory
- **Temporary references** – kernel code that is actively traversing a path (e.g., `namei()` walking through directories) holds transient inode references

The `ref` count is managed by `iget()` (which increments it) and `iput()` (which decrements it). When `ref` reaches zero, the inode can be **evicted from the in-memory inode cache** to make room for other inodes. This does not delete the file – the inode still exists on disk. It merely means the kernel no longer needs to keep it cached in memory.

1.11.1.2 The `nlink` field: on-disk (directory) references

The `nlink` field counts how many **directory entries on disk** refer to this inode. This is the persistent, on-disk reference count. These references come from:

- **Directory entries** – each filename in a directory is a `(name, inode number)` pair. Every such entry that points to this inode contributes 1 to `nlink`.
- **Hard links** – the `link()` system call creates a new directory entry pointing to an existing inode, incrementing its `nlink`. The `unlink()` system call removes a directory entry and decrements `nlink`.

For example, if `/home/report.txt` and `/tmp/backup.txt` are hard links to the same file, the underlying inode has `nlink == 2`. Calling `unlink("/tmp/backup.txt")` removes that directory entry and decrements `nlink` to 1. The file's data is still intact – it's still accessible via `/home/report.txt`.

1.11.2 Garbage collection: when is a file truly deleted?

A file's data blocks are freed (the file is truly deleted) **only when both counts reach zero**:

- `nlink == 0` – no directory entry on disk points to this inode, so no user can open it by name
- `ref == 0` – no process currently has it open in memory

Both conditions must be met. This is why a program can keep reading a file even after another process has `unlink()`ed it. The directory entry is gone (`nlink == 0`), but the open file descriptor holds a reference (`ref > 0`), so the inode and its data blocks survive. Only when the last process closes the file does `ref` drop to zero, at which point `iput()` sees that `nlink == 0` and calls `itrunc()` to free the data blocks and mark the inode as unused on disk.

This two-phase garbage collection can be summarized as:

`nlink`	`ref`	File status
> 0	> 0	Normal: file is named on disk and actively open
> 0	0	File exists on disk but is not currently open; inode may be evicted from cache
0	> 0	File has been unlinked but is still held open by some process; data persists until last close
0	0	File is truly deleted – data blocks and inode are freed

1.11.3 File types in the inode: `T_FILE` and `T_DIR`

The `type` field in `struct inode` (and in the corresponding on-disk `struct dinode`) distinguishes different kinds of files. In xv6, the two most important types are:

1.11.3.1 `T_FILE` (value 2) – regular (plain) files

A regular file is a sequence of bytes stored in data blocks on disk. The inode's `addrs[]` array maps logical file offsets to physical disk block numbers. The `size` field records the total byte count. Regular files are created by `open()` with `O_CREATE` and can be read and written through the standard `read()`/`write()` interface.

1.11.3.2 `T_DIR` (value 1) – directory files

A directory is also a file stored on disk, but its contents have a specific structure: it is an array of `struct dirent` entries:

```
struct dirent {
  ushort inum;      // inode number (0 means empty slot)
  char name[DIRSIZ]; // filename (up to 14 characters in xv6)
};
```

Each entry maps a filename to an inode number. When the kernel needs to resolve a path like `/home/data.txt`, it:

1. Starts at the root directory's inode
2. Reads the root directory's data blocks as an array of `struct dirent`
3. Searches for an entry with `name == "home"`, finding its inode number
4. Reads that inode and repeats the process, searching for `name == "data.txt"`

Directories are the source of `nlink` references. Every `struct dirent` with a non-zero `inum` constitutes a link to the target inode. Creating a file in a directory (via `open()` with `O_CREATE`, or `mkdir()`, or `link()`) adds a `dirent` entry and increments the target inode's `nlink`. Removing a file (via `unlink()`) removes the `dirent` and decrements `nlink`.

Every directory also contains two special entries:

- `.` (dot) – a link to the directory's own inode
- `..` (dot-dot) – a link to the parent directory's inode

This is why a freshly created directory has `nlink == 2`: one from its parent's `dirent` entry, and one from its own `.` entry. Each subdirectory adds another `nlink` to the parent (via its `..` entry).

1.11.3.3 `T_DEV` (value 3) – device files

For completeness, `T_DEV` represents device special files (like the console). These are accessed through the `devsw[]` dispatch table rather than through data blocks on disk.

1.11.4 Summary of the two reference counts

Field	What it counts	Managed by	Persists on disk?	Effect when reaching zero
`ref`	In-memory kernel pointers to this inode	`iget()` / `iput()`	No (in-memory only)	Inode can be evicted from cache; if `nlink` is also 0, file is deleted
`nlink`	Directory entries pointing to this inode	`link()` / `unlink()` / `create()`	Yes (stored on disk)	File can no longer be opened by name; will be deleted when `ref` also reaches 0

Together, these two fields ensure that files are never deleted while someone still has a reference – whether that reference is a filename on disk or an open file descriptor in a running process.

1.12 11. A Classic UNIX Trick: Self-Cleaning Temporary Files

The two-phase garbage collection described above (requiring both `nlink == 0` and `ref == 0`) enables a clever programming pattern for temporary files that has been used since the early days of UNIX.

1.12.1 The pattern

```
int fd = open("/tmp/scratch", O_CREATE | O_RDWR);  // Step 1: create the file
unlink("/tmp/scratch");                              // Step 2: immediately unlink it
// Step 3: use the file normally through fd
write(fd, data, len);
lseek(fd, 0, SEEK_SET);
read(fd, buf, len);
// Step 4: when done (or on crash), the file disappears
close(fd);
```

1.12.2 Why this works

After each step, the reference counts look like this:

Step	Action	`nlink`	`ref`	File status
1	`open()` with `O_CREATE`	1	1	Normal: named on disk, open in memory
2	`unlink()`	0	1	Invisible: no directory entry, but still usable via fd
3	`read()` / `write()` via fd	0	1	Still works – the inode and data blocks are intact
4	`close(fd)`	0	0	Truly deleted – data blocks and inode freed

After step 2, the file is in a unique state: it has no name in any directory (`nlink == 0`), so no other process can discover it or open it by name. But the process that created it still holds a valid file descriptor, and can read and write normally. The inode and its data blocks remain allocated on disk because `ref > 0`.

1.12.3 The crash safety benefit

The key insight is: **what happens if the process – or the entire computer – crashes before step 4?**

- The process crashes (but the OS keeps running): The kernel cleans up all of the dead process's open file descriptors as part of process teardown (in `exit()`). Each descriptor is closed via `fileclose()`, which decrements `ref`. Once `ref` reaches 0, `iput()` sees `nlink == 0` and frees the inode and data blocks. The temporary file is deleted automatically.
- The computer crashes (power failure, kernel panic): When the system reboots and the file system is recovered, the file system scan finds an allocated inode with `nlink == 0` – an inode that no directory entry points to. Since no directory references it, it is unreachable and can be safely freed during recovery. The temporary file is cleaned up automatically.

Compare this to what happens if you **don't** unlink the file early:

- If the process crashes before it gets a chance to unlink, the file remains in `/tmp/scratch` with `nlink == 1`. Nobody cleans it up. Over time, `/tmp` fills with orphaned files from crashed programs.

By unlinking immediately after creation, you guarantee that the file cannot outlive its creator. The reference counting mechanism ensures the file remains usable for exactly as long as it is needed, and no longer.

1.12.4 Why this is safe

A natural concern is: "If I unlink the file, won't I lose my data?" The answer is no – and this is precisely the point of having two separate reference counts. `unlink()` only removes the *name* (the directory entry, decrementing `nlink`). It does not touch the *data*, and it does not close your file descriptor. As long as you hold the fd, the file is fully functional. You simply can't refer to it by pathname anymore.

1.12.5 This pattern in the real world

This technique is widely used in real UNIX systems, not just xv6:

- **Temporary file libraries** – functions like `tmpfile()` in the C standard library often use exactly this pattern internally: create a file, open it, unlink it, and return the file descriptor.
- **Shared memory and IPC** – processes sometimes create temporary files for communication, unlinking them immediately so they vanish when all participating processes exit.
- **Database systems** – temporary work files that should not survive a crash use this approach.

The elegance of this pattern comes entirely from the reference counting design in `struct inode`: `nlink` tracks who can *find* the file, `ref` tracks who is *using* the file, and the data is only freed when both answers are "nobody."

Based on xv6 rev8 (September 1, 2015) – MIT 6.828 teaching operating system.