Interlude: Files and Directories

Thus far we have seen the development of two key operating system abstractions: the process, which is a virtualization of the CPU, and the address space, which is a virtualization of memory. In tandem, these two abstractions allow a program to run as if it is in its own private, isolated world; as if it has its own processor (or processors); as if it has its own memory. This illusion makes programming the system much easier and thus is prevalent today not only on desktops and servers but increasingly on all programmable platforms including mobile phones and the like.

In this section, we add one more critical piece to the virtualization puzzle: **persistent storage**. A persistent-storage device, such as a classic **hard disk drive** or a more modern **solid-state storage device**, stores information permanently (or at least, for a long time). Unlike memory, whose contents are lost when there is a power loss, a persistent-storage device keeps such data intact. Thus, the OS must take extra care with such a device: this is where users keep data that they really care about.

> CRUX: HOW TO MANAGE A PERSISTENT DEVICE
> How should the OS manage a persistent device? What are the APIs? What are the important aspects of the implementation?

Thus, in the next few chapters, we will explore critical techniques for managing persistent data, focusing on methods to improve performance and reliability. We begin, however, with an overview of the API: the interfaces you'll expect to see when interacting with a UNIX file system.

39.1 Files And Directories

Two key abstractions have developed over time in the virtualization of storage. The first is the **file**. A file is simply a linear array of bytes, each of which you can read or write. Each file has some kind of **low-level name**, usually a number of some kind; often, the user is not aware of this
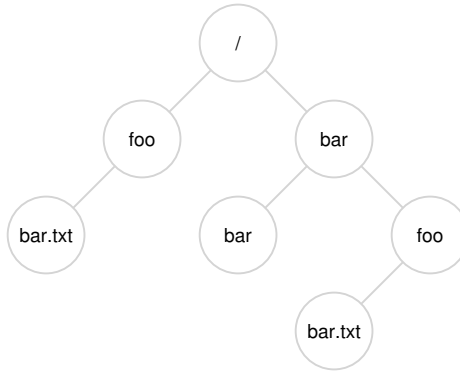
1

Figure 39.1: **An Example Directory Tree**

name (as we will see). For historical reasons, the low-level name of a file is often referred to as its **inode number** (**i-number**). We'll be learning a lot more about inodes in future chapters; for now, just assume that each file has an inode number associated with it.

In most systems, the OS does not know much about the structure of the file (e.g., whether it is a picture, or a text file, or C code); rather, the responsibility of the file system is simply to store such data persistently on disk and make sure that when you request the data again, you get what you put there in the first place. Doing so is not as simple as it seems!

The second abstraction is that of a **directory**. A directory, like a file, also has a low-level name (i.e., an inode number), but its contents are quite specific: it contains a list of (user-readable name, low-level name) pairs. For example, let's say there is a file with the low-level name "10", and it is referred to by the user-readable name of "foo". The directory that "foo" resides in thus would have an entry ("foo", "10") that maps the user-readable name to the low-level name. Each entry in a directory refers to either files or other directories. By placing directories within other directories, users are able to build an arbitrary **directory tree** (or **directory hierarchy**), under which all files and directories are stored.

The directory hierarchy starts at a **root directory** (in UNIX-based systems, the root directory is simply referred to as /) and uses some kind of **separator** to name subsequent **sub-directories** until the desired file or directory is named. For example, if a user created a directory foo in the root directory /, and then created a file bar.txt in the directory foo, we could refer to the file by its **absolute pathname**, which in this case would be /foo/bar.txt. See Figure 39.1 for a more complex directory tree; valid directories in the example are /, /foo, /bar, /bar/bar, /bar/foo and valid files are /foo/bar.txt and /bar/foo/bar.txt.

TIP: THINK CAREFULLY ABOUT NAMING

Naming is an important aspect of computer systems [SK09]. In UNIX systems, virtually everything that you can think of is named through the file system. Beyond just files, devices, pipes, and even processes [K84] can be found in what looks like a plain old file system. This uniformity of naming eases your conceptual model of the system, and makes the system simpler and more modular. Thus, whenever creating a system or interface, think carefully about what names you are using.

Directories and files can have the same name as long as they are in different locations in the file-system tree (e.g., there are two files named `bar.txt` in the figure, `/foo/bar.txt` and `/bar/foo/bar.txt`).

You may also notice that the file name in this example often has two parts: `bar` and `txt`, separated by a period. The first part is an arbitrary name, whereas the second part of the file name is usually used to indicate the **type** of the file, e.g., whether it is C code (e.g., `.c`), or an image (e.g., `.jpg`), or a music file (e.g., `.mp3`). However, this is usually just a **convention**: there is usually no enforcement that the data contained in a file named `main.c` is indeed C source code.

Thus, we can see one great thing provided by the file system: a convenient way to **name** all the files we are interested in. Names are important in systems as the first step to accessing any resource is being able to name it. In UNIX systems, the file system thus provides a unified way to access files on disk, USB stick, CD-ROM, many other devices, and in fact many other things, all located under the single directory tree.

39.2 The File System Interface

Let's now discuss the file system interface in more detail. We'll start with the basics of creating, accessing, and deleting files. You may think this is straightforward, but along the way we'll discover the mysterious call that is used to remove files, known as `unlink()`. Hopefully, by the end of this chapter, this mystery won't be so mysterious to you!

39.3 Creating Files

We'll start with the most basic of operations: creating a file. This can be accomplished with the `open` system call; by calling `open()` and passing it the O_CREAT flag, a program can create a new file. Here is some example code to create a file called "foo" in the current working directory:

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
              S_IRUSR|S_IWUSR);
```

ASIDE: THE CREAT() SYSTEM CALL

The older way of creating a file is to call creat(), as follows:

```
// option: add second flag to set permissions
int fd = creat("foo");
```

You can think of creat() as open() with the following flags: O_CREAT | O_WRONLY | O_TRUNC. Because open() can create a file, the usage of creat() has somewhat fallen out of favor (indeed, it could just be implemented as a library call to open()); however, it does hold a special place in UNIX lore. Specifically, when Ken Thompson was asked what he would do differently if he were redesigning UNIX, he replied: "I'd spell creat with an e." 哈哈哈笑死

The routine open() takes a number of different flags. In this example, the second parameter creates the file (O_CREAT) if it does not exist, ensures that the file can only be written to (O_WRONLY), and, if the file already exists, truncates it to a size of zero bytes thus removing any existing content (O_TRUNC). The third parameter specifies permissions, in this case making the file readable and writable by the owner.

One important aspect of open() is what it returns: a **file descriptor**. A file descriptor is just an integer, private per process, and is used in UNIX systems to access files; thus, once a file is opened, you use the file descriptor to read or write the file, assuming you have permission to do so. In this way, a file descriptor is a **capability** [L84], i.e., an opaque handle that gives you the power to perform certain operations. Another way to think of a file descriptor is as a pointer to an object of type file; once you have such an object, you can call other "methods" to access the file, like read() and write() (we'll see how to do so below).

As stated above, file descriptors are managed by the operating system on a per-process basis. This means some kind of simple structure (e.g., an array) is kept in the proc structure on UNIX systems. Here is the relevant piece from the xv6 kernel [CK+08]:

```
struct proc {
  ...
  struct file *ofile[NOFILE]; // Open files
  ...
};
```

A simple array (with a maximum of NOFILE open files), indexed by the file descriptor, tracks which files are opened on a per-process basis. Each entry of the array is actually just a pointer to a struct file, which will be used to track information about the file being read or written; we'll discuss this further below.

TIP: USE STRACE (AND SIMILAR TOOLS)

The strace tool provides an awesome way to see what programs are up
to. By running it, you can trace which system calls a program makes, see
the arguments and return codes, and generally get a very good idea of
what is going on.

The tool also takes some arguments which can be quite useful. For ex-
ample, -f follows any fork'd children too; -t reports the time of day
at each call; -e trace=open,close,read,write only traces calls to
those system calls and ignores all others. There are many other flags; read
the man pages and find out how to harness this wonderful tool.

39.4 Reading And Writing Files

Once we have some files, of course we might like to read or write them.
Let's start by reading an existing file. If we were typing at a command
line, we might just use the program cat to dump the contents of the file
to the screen.

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

In this code snippet, we redirect the output of the program echo to
the file foo, which then contains the word "hello" in it. We then use cat
to see the contents of the file. But how does the cat program access the
file foo?

To find this out, we'll use an incredibly useful tool to trace the sys-
tem calls made by a program. On Linux, the tool is called **strace**; other
systems have similar tools (see **dtruss** on a Mac, or **truss** on some older
UNIX variants). What strace does is trace every system call made by a
program while it runs, and dump the trace to the screen for you to see.

Here is an example of using strace to figure out what cat is doing
(some calls removed for readability):

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)      = 3
read(3, "hello\n", 4096)               = 6
write(1, "hello\n", 6)                 = 6
hello
read(3, "", 4096)                      = 0
close(3)                               = 0
...
prompt>
```

THREE
EASY
PIECES

Line 1 The <u>first</u> thing that `cat` does is <u>open the file for reading</u>. A couple of things we should note about this; first, that the file is only opened for reading (not writing), as indicated by the `O_RDONLY` flag; second, that the 64-bit offset is used (`O_LARGEFILE`); third, that the call to `open()` succeeds and returns a file descriptor, which has the value of 3.

为什么返回值是3 Why does the first call to `open()` return 3, not 0 or perhaps 1 as you might expect? As it turns out, each running process already has three files open, standard input (which the process can read to receive input), standard output (which the process can write to in order to dump information to the screen), and standard error (which the process can write error messages to). These are represented by file descriptors 0, 1, and 2, respectively. Thus, when you first open another file (as `cat` does above), it will almost certainly be file descriptor 3.

Line 2 After the open succeeds, `cat` uses the `read()` system call to repeatedly read some bytes from a file. The first argument to `read()` is the file descriptor, thus telling the file system which file to read; a process can of course have multiple files open at once, and thus the descriptor enables the operating system to know which file a particular read refers to. The second argument points to a buffer where the result of the `read()` will be placed; in the system-call trace above, strace shows the results of the read in this spot ("hello"). The third argument is the size of the buffer, which in this case is 4 KB. The call to `read()` returns successfully as well, here returning the number of bytes it read (6, which includes 5 for the letters in the word "hello" and one for an end-of-line marker).

Line 3 At this point, you see another interesting result of the strace: a single call to the `write()` system call, to the file descriptor 1. As we mentioned above, this descriptor is known as the standard output, and thus is used to write the word "hello" to the screen as the program `cat` is meant to do. But does it call `write()` directly? Maybe (if it is highly optimized). But if not, what `cat` might do is call the library routine `printf()`; internally, `printf()` figures out all the formatting details passed to it, and eventually writes to standard output to print the results to the screen.

Line 5 & 6 The `cat` program then tries to read more from the file, but since there are no bytes left in the file, the `read()` returns 0 and the program knows that this means it has read the entire file. Thus, the program calls `close()` to indicate that it is done with the file "foo", passing in the corresponding file descriptor. The file is thus closed, and the reading of it thus complete.

Writing a file is accomplished via a similar set of steps. First, a file is opened for writing, then the `write()` system call is called, perhaps repeatedly for larger files, and then `close()`. Use strace to trace writes to a file, perhaps of a program you wrote yourself, or by tracing the dd utility, e.g., dd if=foo of=bar.

> ASIDE: DATA STRUCTURE — THE OPEN FILE TABLE
> Each process maintains an array of file descriptors, each of which refers
> to an entry in the system-wide **open file table**. Each entry in this table
> tracks which underlying file the descriptor refers to, the current offset,
> and other relevant details such as whether the file is readable or writable.

39.5 Reading And Writing, But Not Sequentially

the use of lseek() to jump to anywhere you want to start reading

Thus far, we've discussed how to read and write files, but all access
has been **sequential**; that is, we have either read a file from the beginning
to the end, or written a file out from beginning to end.

Sometimes, however, it is useful to be able to read or write to a spe-
cific offset within a file; for example, if you build an index over a text
document, and use it to look up a specific word, you may end up reading
from some **random** offsets within the document. To do so, we will use
the lseek() system call. Here is the function prototype:

```
off_t lseek(int fildes, off_t offset, int whence);
```

The first argument is familiar (a file descriptor). The second argu-
ment is the offset, which positions the **file offset** to a particular location
within the file. The third argument, called whence for historical reasons,
determines exactly how the seek is performed. From the man page:

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
  location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
  the file plus offset bytes.
```

As you can tell from this description, for each file a process opens, the
OS tracks a "current" offset, which determines where the next read or
write will begin reading from or writing to within the file. Thus, part
of the abstraction of an open file is that it has a current offset, which
is updated in one of two ways. The first is when a read or write of N
bytes takes place, N is added to the current offset; thus each read or write
implicitly updates the offset. The second is *explicitly* with lseek, which
changes the offset as specified above.

The offset, as you might have guessed, is kept in that struct file
we saw earlier, as referenced from the struct proc. Here is a (simpli-
fied) xv6 definition of the structure:

```
struct file {
  int ref;
  char readable;
  char writable;
  struct inode *ip;
  uint off;
};
```

clarification of lseek() and actual disk seek:

> ASIDE: CALLING LSEEK() DOES NOT PERFORM A DISK SEEK
> The poorly-named system call `lseek()` confuses many a student try-
> ing to understand disks and how the file systems atop them work. Do
> not confuse the two! The `lseek()` call simply changes a variable in OS
> memory that tracks, for a particular process, at which offset its next read
> or write will start. A disk seek occurs when a read or write issued to the
> disk is not on the same track as the last read or write, and thus neces-
> sitates a head movement. Making this even more confusing is the fact
> that calling `lseek()` to read or write from/to random parts of a file, and
> then reading/writing to those random parts, will indeed lead to more
> disk seeks. Thus, calling `lseek()` can lead to a seek in an upcoming
> read or write, but absolutely does not cause any disk I/O to occur itself.

As you can see in the structure, the OS can use this to determine
whether the opened file is readable or writable (or both), which under-
lying file it refers to (as pointed to by the `struct inode` pointer `ip`),
and the current offset (`off`). There is also a reference count (`ref`), which
we will discuss further below.

These file structures represent all of the currently opened files in the
system; together, they are sometimes referred to as the **open file table**.
The xv6 kernel just keeps these as an array, with one lock for the entire
table:

```
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
```

Let's make this a bit clearer with a few examples. First, let's track a
process that opens a file (of size 300 bytes) and reads it by calling the
`read()` system call repeatedly, each time reading 100 bytes. Here is a
trace of the relevant system calls, along with the values returned by each
system call, and the value of the current offset in the open file table for
this file access:

System Calls	Return Code	Current Offset
fd = open("file", O_RDONLY);	3	0
read(fd, buffer, 100);	100	100
read(fd, buffer, 100);	100	200
read(fd, buffer, 100);	100	300
read(fd, buffer, 100);	0	300
close(fd);	0	–

There are a couple of items of interest to note from the trace. First,
you can see how the current offset gets initialized to zero when the file is

opened. Next, you can see how it is incremented with each `read()` by the process; this makes it easy for a process to just keep calling `read()` to get the next chunk of the file. Finally, you can see how at the end, an attempted `read()` past the end of the file returns zero, thus indicating to the process that it has read the file in its entirety.

Second, let's trace a process that opens the *same* file twice and issues a read to each of them.

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
fd1 = open("file", O_RDONLY);	3	0	−
fd2 = open("file", O_RDONLY);	4	0	0
read(fd1, buffer1, 100);	100	100	0
read(fd2, buffer2, 100);	100	100	100
close(fd1);	0	−	100
close(fd2);	0	−	−

In this example, two file descriptors are allocated (3 and 4), and each refers to a *different* entry in the open file table (in this example, entries 10 and 11, as shown in the table heading; OFT stands for Open File Table). If you trace through what happens, you can see how each current offset is updated independently.

In one final example, a process uses `lseek()` to reposition the current offset before reading; in this case, only a single open file table entry is needed (as with the first example).

System Calls	Return Code	Current Offset
fd = open("file", O_RDONLY);	3	0
lseek(fd, 200, SEEK_SET);	200	200
read(fd, buffer, 50);	50	250
close(fd);	0	−

Here, the `lseek()` call first sets the current offset to 200. The subsequent `read()` then reads the next 50 bytes, and updates the current offset accordingly.

39.6 Shared File Table Entries: `fork()` And `dup()`

In many cases (as in the examples shown above), the mapping of file descriptor to an entry in the open file table is a one-to-one mapping. For example, when a process runs, it might decide to open a file, read it, and then close it; in this example, the file will have a unique entry in the open file table. Even if some other process reads the same file at the same time, each will have its own entry in the open file table. In this way, each logical

THREE
EASY
PIECES

fork()

```
int main(int argc, char *argv[]) {
    int fd = open("file.txt", O_RDONLY);
    assert(fd >= 0);
    int rc = fork();
    if (rc == 0) {
        rc = lseek(fd, 10, SEEK_SET);
        printf("child: offset %d\n", rc);
    } else if (rc > 0) {
        (void) wait(NULL);
        printf("parent: offset %d\n",
                (int) lseek(fd, 0, SEEK_CUR));
    }
    return 0;
}
```

Figure 39.2: **Shared Parent/Child File Table Entries (`fork-seek.c`)**

reading or writing of a file is independent, and each has its own current offset while it accesses the given file.

However, there are a few interesting cases where an entry in the open file table is *shared*. One of those cases occurs when a parent process creates a child process with fork(). Figure 39.2 shows a small code snippet in which a parent creates a child and then waits for it to complete. The child adjusts the current offset via a call to lseek() and then exits. Finally the parent, after waiting for the child, checks the current offset and prints out its value.

When we run this program, we see the following output:

```
prompt> ./fork-seek
child: offset 10
parent: offset 10
prompt>
```

Figure 39.3 shows the relationships that connect each process's private descriptor array, the shared open file table entry, and the reference from it to the underlying file-system inode. Note that we finally make use of the **reference count** here. When a file table entry is shared, its reference count is incremented; only when both processes close the file (or exit) will the entry be removed.

Sharing open file table entries across parent and child is occasionally useful. For example, if you create a number of processes that are cooperatively working on a task, they can write to the same output file without any extra coordination. For more on what is shared by processes when fork() is called, please see the man pages.
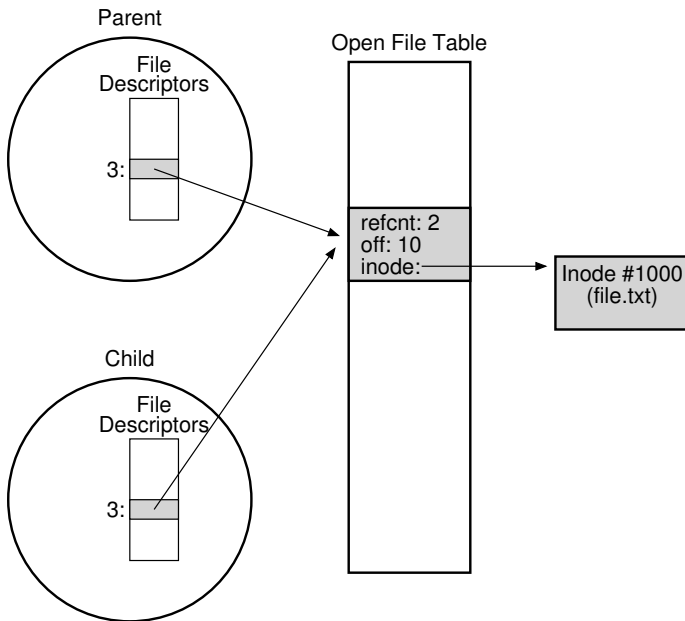
Figure 39.3: **Processes Sharing An Open File Table Entry**

One other interesting, and perhaps more useful, case of sharing occurs with the **dup()** system call (and its cousins, **dup2()** and **dup3()**).

dup()

The dup() call allows a process to create a new file descriptor that refers to the same underlying open file as an existing descriptor. Figure 39.4 shows a small code snippet that shows how dup() can be used.

The dup() call (and, in particular, dup2()) is useful when writing a UNIX shell and performing operations like output redirection; spend some time and think about why! And now, you are thinking: why didn't they tell me this when I was doing the shell project? Oh well, you can't get everything in the right order, even in an incredible book about operating systems. Sorry!

```
int main(int argc, char *argv[]) {
    int fd = open("README", O_RDONLY);
    assert(fd >= 0);
    int fd2 = dup(fd);
    // now fd and fd2 can be used interchangeably
    return 0;
}
```
Figure 39.4: **Shared File Table Entry With dup() (dup.c)**

39.7 Writing Immediately With `fsync()`

Most times when a program calls `write()`, it is just telling the file system: please write this data to persistent storage, at some point in the future. The file system, for performance reasons, will **buffer** such writes in memory for some time (say 5 seconds, or 30); at that later point in time, the write(s) will actually be issued to the storage device. From the perspective of the calling application, writes seem to complete quickly, and only in rare cases (e.g., the machine crashes after the `write()` call but before the write to disk) will data be lost.

However, some applications require something more than this eventual guarantee. For example, in a database management system (DBMS), development of a correct recovery protocol requires the ability to force writes to disk from time to time.

To support these types of applications, most file systems provide some additional control APIs. In the UNIX world, the interface provided to applications is known as `fsync(int fd)`. When a process calls `fsync()` for a particular file descriptor, the file system responds by forcing all **dirty** (i.e., not yet written) data to disk, for the file referred to by the specified file descriptor. The `fsync()` routine returns once all of these writes are complete.

Here is a simple example of how to use `fsync()`. The code opens the file `foo`, writes a single chunk of data to it, and then calls `fsync()` to ensure the writes are forced immediately to disk. Once the `fsync()` returns, the application can safely move on, knowing that the data has been persisted (if `fsync()` is correctly implemented, that is).

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
              S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

Interestingly, this sequence does not guarantee everything that you might expect; in some cases, you also need to `fsync()` the directory that contains the file `foo`. Adding this step ensures not only that the file itself is on disk, but that the file, if newly created, also is durably a part of the directory. Not surprisingly, this type of detail is often overlooked, leading to many application-level bugs [P+13,P+14].

39.8 Renaming Files

Once we have a file, it is sometimes useful to be able to give a file a different name. When typing at the command line, this is accomplished with `mv` command; in this example, the file `foo` is renamed `bar`:

ASIDE: MMAP() AND PERSISTENT MEMORY
(Guest Aside by Terence Kelly)

Memory mapping is an alternative way to access persistent data in files. The `mmap()` system call creates a correspondence between byte offsets in a file and virtual addresses in the calling process; the former is called the **backing file** and the latter its **in-memory image**. The process can then access the backing file using CPU instructions (i.e., loads and stores) to the in-memory image.

By combining the persistence of files with the access semantics of memory, file-backed memory mappings support a software abstraction called **persistent memory**. The persistent memory style of programming can streamline applications by eliminating translation between different data formats for memory and storage [K19].

```
1  p = mmap(NULL, file_size, PROT_READ|PROT_WRITE,
2          MAP_SHARED, fd, 0);
3  assert(p != MAP_FAILED);
4  for (int i = 1; i < argc; i++)
5    if (strcmp(argv[i], "pop") == 0) // pop
6      if (p->n > 0)  // stack not empty
7        printf("%d\n", p->stack[--p->n]);
8    } else {  // push
9      if (sizeof(pstack_t) + (1 + p->n) * sizeof(int)
10         <= file_size)  // stack not full
11       p->stack[p->n++] = atoi(argv[i]);
12    }
```

The program `pstack.c` (included on the OSTEP code github repo, with a snippet shown above) stores a persistent stack in file `ps.img`, which begins life as a bag of zeros, e.g., created on the command line via the `truncate` or `dd` utility. The file contains a count of the size of the stack and an array of integers holding stack contents. After `mmap()`-ing the backing file we can access the stack using C pointers to the in-memory image, e.g., `p->n` accesses the number of items on the stack, and `p->stack` the array of integers. Because the stack is persistent, data `push`'d by one invocation of `pstack` can be `pop`'d by the next.

A crash, e.g., between the increment and the assignment of the `push`, could leave our persistent stack in an inconsistent state. Applications prevent such damage by using mechanisms that update persistent memory atomically with respect to failure [K20].

```
prompt> mv foo bar
```

Using strace, we can see that mv uses the system call rename(char *old, char *new), which takes precisely two arguments: the original name of the file (old) and the new name (new).

One interesting guarantee provided by the rename() call is that it is (usually) implemented as an **atomic** call with respect to system crashes; if the system crashes during the renaming, the file will either be named the old name or the new name, and no odd in-between state can arise. Thus, rename() is critical for supporting certain kinds of applications that require an atomic update to file state.

Let's be a little more specific here. Imagine that you are using a file editor (e.g., emacs), and you insert a line into the middle of a file. The file's name, for the example, is foo.txt. The way the editor might update the file to guarantee that the new file has the original contents plus the line inserted is as follows (ignoring error-checking for simplicity):

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

What the editor does in this example is simple: write out the new version of the file under a temporary name (foo.txt.tmp), force it to disk with fsync(), and then, when the application is certain the new file metadata and contents are on the disk, rename the temporary file to the original file's name. This last step atomically swaps the new file into place, while concurrently deleting the old version of the file, and thus an atomic file update is achieved.

39.9 Getting Information About Files

Beyond file access, we expect the file system to keep a fair amount of information about each file it is storing. We generally call such data about files **metadata**. To see the metadata for a certain file, we can use the stat() or fstat() system calls. These calls take a pathname (or file descriptor) to a file and fill in a stat structure as seen in Figure 39.5.

You can see that there is a lot of information kept about each file, including its size (in bytes), its low-level name (i.e., inode number), some ownership information, and some information about when the file was accessed or modified, among other things. To see this information, you can use the command line tool stat. In this example, we first create a file (called file) and then use the stat command line tool to learn some things about the file.

```
struct stat {
    dev_t     st_dev;     // ID of device containing file
    ino_t     st_ino;     // inode number
    mode_t    st_mode;    // protection
    nlink_t   st_nlink;   // number of hard links
    uid_t     st_uid;     // user ID of owner
    gid_t     st_gid;     // group ID of owner
    dev_t     st_rdev;    // device ID (if special file)
    off_t     st_size;    // total size, in bytes
    blksize_t st_blksize; // blocksize for filesystem I/O
    blkcnt_t  st_blocks;  // number of blocks allocated
    time_t    st_atime;   // time of last access
    time_t    st_mtime;   // time of last modification
    time_t    st_ctime;   // time of last status change
};
```
Figure 39.5: **The `stat` structure.**

Here is the output on Linux:

```
prompt> echo hello > file
prompt> stat file
  File: 'file'
  Size: 6   Blocks: 8   IO Block: 4096   regular file
Device: 811h/2065d Inode: 67158084     Links: 1
Access: (0640/-rw-r-----) Uid: (30686/remzi)
  Gid: (30686/remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

Each file system usually keeps this type of information in a structure called an **inode**[1]. We'll be learning a lot more about inodes when we talk about file system implementation. For now, you should just think of an inode as a persistent data structure kept by the file system that has information like we see above inside of it. All inodes reside on disk; a copy of active ones are usually cached in memory to speed up access.

39.10 Removing Files

At this point, we know how to create files and access them, either sequentially or not. But how do you delete files? If you've used UNIX, you probably think you know: just run the program `rm`. But what system call does `rm` use to remove a file?

---

[1]Some file systems call these structures similar, but slightly different, names, such as dnodes; the basic idea is similar however.

THREE
EASY
PIECES

Let's use our old friend `strace` again to find out. Here we remove that pesky file `foo`:

```
prompt> strace rm foo
...
unlink("foo")                                = 0
...
```

We've removed a bunch of unrelated cruft from the traced output, leaving just a single call to the mysteriously-named system call `unlink()`. As you can see, `unlink()` just takes the name of the file to be removed, and returns zero upon success. But this leads us to a great puzzle: why is this system call named `unlink`? Why not just `remove` or `delete`? To understand the answer to this puzzle, we must first understand more than just files, but also directories.

39.11 Making Directories

Beyond files, a set of directory-related system calls enable you to make, read, and delete directories. Note you can never write to a directory directly. Because the format of the directory is considered file system metadata, the file system considers itself responsible for the integrity of directory data; thus, you can only update a directory indirectly by, for example, creating files, directories, or other object types within it. In this way, the file system makes sure that directory contents are as expected.

To create a directory, a single system call, `mkdir()`, is available. The eponymous `mkdir` program can be used to create such a directory. Let's take a look at what happens when we run the `mkdir` program to make a simple directory called `foo`:

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                           = 0
...
prompt>
```

When such a directory is created, it is considered "empty", although it does have a bare minimum of contents. Specifically, an empty directory has two entries: one entry that refers to itself, and one entry that refers to its parent. The former is referred to as the "." (dot) directory, and the latter as ".." (dot-dot). You can see these directories by passing a flag (`-a`) to the program `ls`:
```
prompt> ls -a
./  ../
prompt> ls -al
total 8
drwxr-x---  2 remzi remzi    6 Apr 30 16:17 ./
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

TIP: BE WARY OF POWERFUL COMMANDS

The program `rm` provides us with a great example of powerful commands, and how sometimes too much power can be a bad thing. For example, to remove a bunch of files at once, you can type something like:

```
prompt> rm *
```

where the `*` will match all files in the current directory. But sometimes you want to also delete the directories too, and in fact all of their contents. You can do this by telling `rm` to recursively descend into each directory, and remove its contents too:

```
prompt> rm -rf *
```

Where you get into trouble with this small string of characters is when you issue the command, accidentally, from the root directory of a file system, thus removing every file and directory from it. Oops!

Thus, remember the double-edged sword of powerful commands; while they give you the ability to do a lot of work with a small number of keystrokes, they also can quickly and readily do a great deal of harm.

39.12 Reading Directories

Now that we've created a directory, we might wish to read one too. Indeed, that is exactly what the program `ls` does. Let's write our own little tool like `ls` and see how it is done.

Instead of just opening a directory as if it were a file, we instead use a new set of calls. Below is an example program that prints the contents of a directory. The program uses three calls, `opendir()`, `readdir()`, and `closedir()`, to get the job done, and you can see how simple the interface is; we just use a simple loop to read one directory entry at a time, and print out the name and inode number of each file in the directory.

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino,
                d->d_name);
    }
    closedir(dp);
    return 0;
}
```

The declaration below shows the information available within each directory entry in the `struct dirent` data structure:

```
struct dirent {
  char          d_name[256]; // filename
  ino_t         d_ino;       // inode number
  off_t         d_off;       // offset to the next dirent
  unsigned short d_reclen;   // length of this record
  unsigned char d_type;      // type of file
};
```

Because directories are light on information (basically, just mapping the name to the inode number, along with a few other details), a program may want to call `stat()` on each file to get more information on each, such as its length or other detailed information. Indeed, this is exactly what `ls` does when you pass it the `-l` flag; try `strace` on `ls` with and without that flag to see for yourself.

39.13 Deleting Directories

Finally, you can delete a directory with a call to `rmdir()` (which is used by the program of the same name, `rmdir`). Unlike file deletion, however, removing directories is more dangerous, as you could potentially delete a large amount of data with a single command. Thus, `rmdir()` has the requirement that the directory be empty (i.e., only has "." and ".." entries) before it is deleted. If you try to delete a non-empty directory, the call to `rmdir()` simply will fail.

39.14 Hard Links

We now come back to the mystery of why removing a file is performed via `unlink()`, by understanding a new way to make an entry in the file system tree, through a system call known as `link()`. The `link()` system call takes two arguments, an old pathname and a new one; when you "link" a new file name to an old one, you essentially create another way to refer to the same file. The command-line program `ln` is used to do this, as we see in this example:

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

Here we created a file with the word "hello" in it, and called the file file[2]. We then create a hard link to that file using the `ln` program. After this, we can examine the file by either opening `file` or `file2`.

The way `link()` works is that it simply creates another name in the directory you are creating the link to, and refers it to the *same* inode number (i.e., low-level name) of the original file. The file is not copied in any way; rather, you now just have two human-readable names (`file` and `file2`) that both refer to the same file. We can even see this in the directory itself, by printing out the inode number of each file:

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

By passing the `-i` flag to `ls`, it prints out the inode number of each file (as well as the file name). And thus you can see what link really has done: just make a new reference to the same exact inode number (67158084 in this example).

By now you might be starting to see why `unlink()` is called `unlink()`. When you create a file, you are really doing *two* things. First, you are making a structure (the inode) that will track virtually all relevant information about the file, including its size, where its blocks are on disk, and so forth. Second, you are *linking* a human-readable name to that file, and putting that link into a directory.

After creating a hard link to a file, the file system perceives no difference between the original file name (`file`) and the newly created file name (`file2`); indeed, they are both just links to the underlying metadata about the file, which is found in inode number 67158084.

Thus, to remove a file from the file system, we call `unlink()`. In the example above, we could for example remove the file named `file`, and still access the file without difficulty:

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

The reason this works is because when the file system unlinks file, it checks a **reference count** within the inode number. This reference count (sometimes called the **link count**) allows the file system to track how many different file names have been linked to this particular inode. When `unlink()` is called, it removes the "link" between the human-readable

---

[2]Note again how creative the authors of this book are. We also used to have a cat named "Cat" (true story). However, she died, and we now have a hamster named "Hammy." Update: Hammy is now dead too. The pet bodies are piling up.

name (the file that is being deleted) to the given inode number, and decrements the reference count; only when the reference count reaches zero does the file system also free the inode and related data blocks, and thus truly "delete" the file.

You can see the reference count of a file using `stat()` of course. Let's see what it is when we create and delete hard links to a file. In this example, we'll create three links to the same file, and then delete them. Watch the link count!

```
prompt> echo hello > file
prompt> stat file
... Inode: 67158084    Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084    Links: 2 ...
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084    Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> rm file2
prompt> stat file3
... Inode: 67158084    Links: 1 ...
prompt> rm file3
```

39.15 Symbolic Links

There is one other type of link that is really useful, and it is called a **symbolic link** or sometimes a **soft link**. Hard links are somewhat limited: you can't create one to a directory (for fear that you will create a cycle in the directory tree); you can't hard link to files in other disk partitions (because inode numbers are only unique within a particular file system, not across file systems); etc. Thus, a new type of link called the symbolic link was created [MJLF84].

To create such a link, you can use the same program `ln`, but with the `-s` flag. Here is an example:

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

As you can see, creating a soft link looks much the same, and the original file can now be accessed through the file name `file` as well as the symbolic link name `file2`.

However, beyond this surface similarity, symbolic links are actually quite different from hard links. The first difference is that a symbolic link is actually a file itself, of a different type. We've already talked about regular files and directories; symbolic links are a third type the file system knows about. A `stat` on the symlink reveals all:

```
prompt> stat file
 ... regular file ...
prompt> stat file2
 ... symbolic link ...
```

Running `ls` also reveals this fact. If you look closely at the first character of the long-form of the output from `ls`, you can see that the first character in the left-most column is a − for regular files, a d for directories, and an l for soft links. You can also see the size of the symbolic link (4 bytes in this case) and what the link points to (the file named `file`).

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r-----  1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

The reason that `file2` is 4 bytes is because the way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file. Because we've linked to a file named `file`, our link file `file2` is small (4 bytes). If we link to a longer pathname, our link file would be bigger:

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi  6 May  3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May  3 19:17 file3 ->
                                            alongerfilename
```

Finally, because of the way symbolic links are created, they leave the possibility for what is known as a **dangling reference**:

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

THREE
EASY
PIECES

As you can see in this example, quite unlike hard links, removing the original file named `file` causes the link to point to a pathname that no longer exists.

39.16 Permission Bits And Access Control Lists

The abstraction of a process provided two central virtualizations: of the CPU and of memory. Each of these gave the illusion to a process that it had its own *private* CPU and its own *private* memory; in reality, the OS underneath used various techniques to share limited physical resources among competing entities in a safe and secure manner.

The file system also presents a virtual view of a disk, transforming it from a bunch of raw blocks into much more user-friendly files and directories, as described within this chapter. However, the abstraction is notably different from that of the CPU and memory, in that files are commonly *shared* among different users and processes and are not (always) private. Thus, a more comprehensive set of mechanisms for enabling various degrees of sharing are usually present within file systems.

The first form of such mechanisms is the classic UNIX **permission bits**. To see permissions for a file `foo.txt`, just type:

```
prompt> ls -l foo.txt
-rw-r--r--  1 remzi wheel  0 Aug 24 16:29 foo.txt
```

We'll just pay attention to the first part of this output, namely the `-rw-r--r--`. The first character here just shows the type of the file: `-` for a regular file (which `foo.txt` is), `d` for a directory, `l` for a symbolic link, and so forth; this is (mostly) not related to permissions, so we'll ignore it for now.

We are interested in the permission bits, which are represented by the next nine characters (`rw-r--r--`). These bits determine, for each regular file, directory, and other entities, exactly who can access it and how.

The permissions consist of three groupings: what the **owner** of the file can do to it, what someone in a **group** can do to the file, and finally, what anyone (sometimes referred to as **other**) can do. The abilities the owner, group member, or others can have include the ability to read the file, write it, or execute it.

In the example above, the first three characters of the output of `ls` show that the file is both readable and writable by the owner (`rw-`), and only readable by members of the group `wheel` and also by anyone else in the system (`r--` followed by `r--`).

The owner of the file can readily change these permissions, for example by using the `chmod` command (to change the **file mode**). To remove the ability for anyone except the owner to access the file, you could type:

```
prompt> chmod 600 foo.txt
```

ASIDE: SUPERUSER FOR FILE SYSTEMS
Which user is allowed to do privileged operations to help administer the file system? For example, if an inactive user's files need to be deleted to save space, who has the rights to do so?

On local file systems, the common default is for there to be some kind of **superuser** (i.e., **root**) who can access all files regardless of privileges. In a distributed file system such as AFS (which has access control lists), a group called `system:administrators` contains users that are trusted to do so. In both cases, these trusted users represent an inherent security risk; if an attacker is able to somehow impersonate such a user, the attacker can access all the information in the system, thus violating expected privacy and protection guarantees.

This command enables the readable bit (4) and writable bit (2) for the owner (OR'ing them together yields the 6 above), but set the group and other permission bits to 0 and 0, respectively, thus setting the permissions to `rw-------`.

The execute bit is particularly interesting. For regular files, its presence determines whether a program can be run or not. For example, if we have a simple shell script called `hello.csh`, we may wish to run it by typing:

```
prompt> ./hello.csh
hello, from shell world.
```

However, if we don't set the execute bit properly for this file, the following happens:

```
prompt> chmod 600 hello.csh
prompt> ./hello.csh
./hello.csh: Permission denied.
```

For directories, the execute bit behaves a bit differently. Specifically, it enables a user (or group, or everyone) to do things like change directories (i.e., `cd`) into the given directory, and, in combination with the writable bit, create files therein. The best way to learn more about this: play around with it yourself! Don't worry, you (probably) won't mess anything up too badly.

Beyond permissions bits, some file systems, such as the distributed file system known as AFS (discussed in a later chapter), include more sophisticated controls. AFS, for example, does this in the form of an **access control list** (**ACL**) per directory. Access control lists are a more general and powerful way to represent exactly who can access a given resource. In a file system, this enables a user to create a very specific list of who can and cannot read a set of files, in contrast to the somewhat limited owner/group/everyone model of permissions bits described above.

For example, here are the access controls for a private directory in one
author's AFS account, as shown by the `fs listacl` command:

```
prompt> fs listacl private
Access list for private is
Normal rights:
  system:administrators rlidwka
  remzi rlidwka
```

The listing shows that both the system administrators and the user
`remzi` can lookup, insert, delete, and administer files in this directory, as
well as read, write, and lock those files.

To allow someone (in this case, the other author) to access this direc-
tory, user `remzi` can just type the following command.

```
prompt> fs setacl private/ andrea rl
```

There goes `remzi`'s privacy! But now you have learned an even more
important lesson: there can be no secrets in a good marriage, even within
the file system[3].

39.17 Making And Mounting A File System

We've now toured the basic interfaces to access files, directories, and
certain special types of links. But there is one more topic we should dis-
cuss: how to assemble a full directory tree from many underlying file
systems. This task is accomplished via first making file systems, and then
mounting them to make their contents accessible.

To make a file system, most file systems provide a tool, usually re-
ferred to as `mkfs` (pronounced "make fs"), that performs exactly this task.
The idea is as follows: give the tool, as input, a device (such as a disk par-
tition, e.g., `/dev/sda1`) and a file system type (e.g., ext3), and it simply
writes an empty file system, starting with a root directory, onto that disk
partition. And mkfs said, let there be a file system!

However, once such a file system is created, it needs to be made ac-
cessible within the uniform file-system tree. This task is achieved via the
`mount` program (which makes the underlying system call `mount()` to do
the real work). What mount does, quite simply is take an existing direc-
tory as a target **mount point** and essentially paste a new file system onto
the directory tree at that point.

An example here might be useful. Imagine we have an unmounted
ext3 file system, stored in device partition `/dev/sda1`, that has the fol-
lowing contents: a root directory which contains two sub-directories, `a`
and `b`, each of which in turn holds a single file named `foo`. Let's say we
wish to mount this file system at the mount point `/home/users`. We
would type something like this:

---

[3]Married happily since 1996, if you were wondering. We know, you weren't.

TIP: BE WARY OF TOCTTOU

In 1974, McPhee noticed a problem in computer systems. Specifically, McPhee noted that "... if there exists a time interval between a validity-check and the operation connected with that validity-check, [and,] through multitasking, the validity-check variables can deliberately be changed during this time interval, resulting in an invalid operation being performed by the control program." We today call this the **Time Of Check To Time Of Use** (**TOCTTOU**) problem, and alas, it still can occur.

A simple example, as described by Bishop and Dilger [BD96], shows how a user can trick a more trusted service and thus cause trouble. Imagine, for example, that a mail service runs as root (and thus has privilege to access all files on a system). This service appends an incoming message to a user's inbox file as follows. First, it calls `lstat()` to get information about the file, specifically ensuring that it is actually just a regular file owned by the target user, and not a link to another file that the mail server should not be updating. Then, after the check succeeds, the server updates the file with the new message.

Unfortunately, the gap between the check and the update leads to a problem: the attacker (in this case, the user who is receiving the mail, and thus has permissions to access the inbox) switches the inbox file (via a call to `rename()`) to point to a sensitive file such as `/etc/passwd` (which holds information about users and their passwords). If this switch happens at just the right time (between the check and the access), the server will blithely update the sensitive file with the contents of the mail. The attacker can now write to the sensitive file by sending an email, an escalation in privilege; by updating `/etc/passwd`, the attacker can add an account with `root` privileges and thus gain control of the system.

There are not any simple and great solutions to the TOCTTOU problem [T+08]. One approach is to reduce the number of services that need root privileges to run, which helps. The `O_NOFOLLOW` flag makes it so that `open()` will fail if the target is a symbolic link, thus avoiding attacks that require said links. More radical approaches, such as using a **transactional file system** [H+18], would solve the problem, but there aren't many transactional file systems in wide deployment. Thus, the usual (lame) advice: be careful when you write code that runs with high privileges!

```
prompt> mount -t ext3 /dev/sda1 /home/users
```

If successful, the mount would thus make this new file system available. However, note how the new file system is now accessed. To look at the contents of the root directory, we would use `ls` like this:

```
prompt> ls /home/users/
a b
```

As you can see, the pathname `/home/users/` now refers to the root of the newly-mounted directory. Similarly, we could access directories `a` and `b` with the pathnames `/home/users/a` and `/home/users/b`. Finally, the files named `foo` could be accessed via `/home/users/a/foo` and `/home/users/b/foo`. And thus the beauty of mount: instead of having a number of separate file systems, mount unifies all file systems into one tree, making naming uniform and convenient.

To see what is mounted on your system, and at which points, simply run the `mount` program. You'll see something like this:

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

This crazy mix shows that a whole number of different file systems, including ext3 (a standard disk-based file system), the proc file system (a file system for accessing information about current processes), tmpfs (a file system just for temporary files), and AFS (a distributed file system) are all glued together onto this one machine's file-system tree.

39.18 Summary

The file system interface in UNIX systems (and indeed, in any system) is seemingly quite rudimentary, but there is a lot to understand if you wish to master it. Nothing is better, of course, than simply using it (a lot). So please do so! Of course, read more; as always, Stevens [SR05] is the place to begin.

ASIDE: KEY FILE SYSTEM TERMS

- A **file** is an array of bytes which can be created, read, written, and deleted. It has a low-level name (i.e., a number) that refers to it uniquely. The low-level name is often called an **i-number**.

- A **directory** is a collection of tuples, each of which contains a human-readable name and low-level name to which it maps. Each entry refers either to another directory or to a file. Each directory also has a low-level name (i-number) itself. A directory always has two special entries: the `.` entry, which refers to itself, and the `..` entry, which refers to its parent.

- A **directory tree** or **directory hierarchy** organizes all files and directories into a large tree, starting at the **root**.

- To access a file, a process must use a system call (usually, `open()`) to request permission from the operating system. If permission is granted, the OS returns a **file descriptor**, which can then be used for read or write access, as permissions and intent allow.

- Each file descriptor is a private, per-process entity, which refers to an entry in the **open file table**. The entry therein tracks which file this access refers to, the **current offset** of the file (i.e., which part of the file the next read or write will access), and other relevant information.

- Calls to `read()` and `write()` naturally update the current offset; otherwise, processes can use `lseek()` to change its value, enabling random access to different parts of the file.

- To force updates to persistent media, a process must use `fsync()` or related calls. However, doing so correctly while maintaining high performance is challenging [P+14], so think carefully when doing so.

- To have multiple human-readable names in the file system refer to the same underlying file, use **hard links** or **symbolic links**. Each is useful in different circumstances, so consider their strengths and weaknesses before usage. And remember, deleting a file is just performing that one last `unlink()` of it from the directory hierarchy.

- Most file systems have mechanisms to enable and disable sharing. A rudimentary form of such controls are provided by **permissions bits**; more sophisticated **access control lists** allow for more precise control over exactly who can access and manipulate information.

References

[BD96] "Checking for Race Conditions in File Accesses" by Matt Bishop, Michael Dilger. Computing Systems 9:2, 1996. *A great description of the TOCTTOU problem and its presence in file systems.*

[CK+08] "The xv6 Operating System" by Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich. From: https://github.com/mit-pdos/xv6-public. *As mentioned before, a cool and simple Unix implementation. We have been using an older version (2012-01-30-1-g1c41342) and hence some examples in the book may not match the latest in the source.*

[H+18] "TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions" by Y. Hu, Z. Zhu, I. Neal, Y. Kwon, T. Cheng, V. Chidambaram, E. Witchel. USENIX ATC '18, June 2018. *The best paper at USENIX ATC '18, and a good recent place to start to learn about transactional file systems.*

[K19] "Persistent Memory Programming on Conventional Hardware" by Terence Kelly. ACM Queue, 17:4, July/August 2019. *A great overview of persistent memory programming; check it out!*

[K20] "Is Persistent Memory Persistent?" by Terence Kelly. Communications of the ACM, 63:9, September 2020. *An engaging article about how to test hardware failures in system on the cheaps; who knew breaking things could be so fun?*

[K84] "Processes as Files" by Tom J. Killian. USENIX, June 1984. *The paper that introduced the /proc file system, where each process can be treated as a file within a pseudo file system. A clever idea that you can still see in modern* UNIX *systems.*

[L84] "Capability-Based Computer Systems" by Henry M. Levy. Digital Press, 1984. Available: http://homes.cs.washington.edu/~levy/capabook. *An excellent overview of early capability-based systems.*

[MJLF84] "A Fast File System for UNIX" by Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry. ACM TOCS, 2:3, August 1984. *We'll talk about the Fast File System (FFS) explicitly later on. Here, we refer to it because of all the other random fun things it introduced, like long file names and symbolic links. Sometimes, when you are building a system to improve one thing, you improve a lot of other things along the way.*

[P+13] "Towards Efficient, Portable Application-Level Consistency" by Thanumalayan S. Pillai, Vijay Chidambaram, Joo-Young Hwang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. HotDep '13, November 2013. *Our own work that shows how readily applications can make mistakes in committing data to disk; in particular, assumptions about the file system creep into applications and thus make the applications work correctly only if they are running on a specific file system.*

[P+14] "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications" by Thanumalayan S. Pillai, Vijay Chidambaram, Ramnathhan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. OSDI '14, Broomfield, Colorado, October 2014. *The full conference paper on this topic – with many more details and interesting tidbits than the first workshop paper above.*

[SK09] "Principles of Computer System Design" by Jerome H. Saltzer and M. Frans Kaashoek. Morgan-Kaufmann, 2009. *This tour de force of systems is a must-read for anybody interested in the field. It's how they teach systems at MIT. Read it once, and then read it a few more times to let it all soak in.*

[SR05] "Advanced Programming in the UNIX Environment" by W. Richard Stevens and Stephen A. Rago. Addison-Wesley, 2005. *We have probably referenced this book a few hundred thousand times. It is that useful to you, if you care to become an awesome systems programmer.*

[T+08] "Portably Solving File TOCTTOU Races with Hardness Amplification" by D. Tsafrir, T. Hertz, D. Wagner, D. Da Silva. FAST '08, San Jose, California, 2008. *Not the paper that introduced TOCTTOU, but a recent-ish and well-done description of the problem and a way to solve the problem in a portable manner.*

Homework (Code)

In this homework, we'll just familiarize ourselves with how the APIs described in the chapter work. To do so, you'll just write a few different programs, mostly based on various UNIX utilities.

Questions

1. **Stat:** Write your own version of the command line program `stat`, which simply calls the `stat()` system call on a given file or directory. Print out file size, number of blocks allocated, reference (link) count, and so forth. What is the link count of a directory, as the number of entries in the directory changes? Useful interfaces: `stat()`, naturally.

2. **List Files:** Write a program that lists files in the given directory. When called without any arguments, the program should just print the file names. When invoked with the `-l` flag, the program should print out information about each file, such as the owner, group, permissions, and other information obtained from the `stat()` system call. The program should take one additional argument, which is the directory to read, e.g., `myls -l directory`. If no directory is given, the program should just use the current working directory. Useful interfaces: `stat()`, `opendir()`, `readdir()`, `getcwd()`.

3. **Tail:** Write a program that prints out the last few lines of a file. The program should be efficient, in that it seeks to near the end of the file, reads in a block of data, and then goes backwards until it finds the requested number of lines; at this point, it should print out those lines from beginning to the end of the file. To invoke the program, one should type: `mytail -n file`, where `n` is the number of lines at the end of the file to print. Useful interfaces: `stat()`, `lseek()`, `open()`, `read()`, `close()`.

4. **Recursive Search:** Write a program that prints out the names of each file and directory in the file system tree, starting at a given point in the tree. For example, when run without arguments, the program should start with the current working directory and print its contents, as well as the contents of any sub-directories, etc., until the entire tree, root at the CWD, is printed. If given a single argument (of a directory name), use that as the root of the tree instead. Refine your recursive search with more fun options, similar to the powerful `find` command line tool. Useful interfaces: figure it out.