Introduction

So far, two important operating system abstractions have emerged: the process, which virtualizes the CPU, and the address space, which virtualizes memory. As if in its own little, secluded world, with its own processor (or processors), and memory. Thus, this illusion is now ubiquitous not only on PCs and servers, but on all programmable platforms, including mobile phones.

To the virtualization puzzle, we add persistent storage in this section. Hard disk drives or contemporary solid-state storage devices save data permanently (or at least, for a long time). Unlike memory, which loses data when the power goes out, persistent storage maintains data intact. A gadget like this requires special care from the OS since it contains sensitive data.

How to manage a persistent device

Managing persistent devices in the OS
What is an API?
What are the key elements of implementation?

Determining how to manage persistent data efficiently and reliably will be the subject of the following chapters. But first, a review of the API: the interfaces you can expect when working with a UNIX file system.

Files And Directories

Storage virtualization has evolved around two main concepts. First, the file. A file is a linear collection of bytes that can be read or written. Each file has a low-level name, usually a number; the user is often unaware of this name (as we will see). The file's low-level name is sometimes referred to as its inode number. We'll learn more about inodes in later chapters, but for now, presume that each file has an inode number.

The file system is responsible for storing data persistently on disk and ensuring that when you request the data again, you get what you put there originally. It's not as easy as it seems!

A directory is the second abstraction. Inode numbers are low-level names. A directory's contents are a list of (user-readable name, low-level name) pairs. Let's imagine a file has the low-level name "10" and the user-readable name "foo". "foo" would so have an item ("foo", "10") mapping the user-readable name to the low-level name. A directory's entries refer to files or other directories. Users can create an arbitrary directory tree (or directory hierarchy) by nesting directories within other directories.

The directory hierarchy starts at the root directory (/ in UNIX-based systems) and employs a separator to identify sub-directories until the desired file or directory is specified. The file bar.txt would be referenced by its absolute pathname if it was created in the directory foo, which in this case would be /foo/bar.txt. Figure 39.1 shows a more sophisticated directory tree with valid folders and files like /foo/bar.txt and /bar/foo/bar.txt.

/foo/bar.txt and /bar/foo/bar.txt are both files named bar.txt in the figure.

INSERT FIGURE

In this example, the file name commonly contains two parts: bar and txt, separated by a period. The first component is random, however the second part usually indicates the file type, e.g., C code (e.g.,.c), image (e.g.,.jpg), or music file (e.g.,.mp3) (e.g., .mp3). This is usually only a convention; there is no guarantee that data in a file named main.c is actually C source code.

So, the file system provides us with one fantastic feature: a convenient way to name all our files. Names are crucial in systems because they allow users to identify resources. This means that files on HDD, USB stick, CD-ROM, and other devices can all be accessed from a single directory tree under UNIX systems.

The File System Interface

Now let’s get inside the file system interface. Begin by creating, accessing, and deleting files. You may assume this is simple, however we’ll learn the strange unlink() call used to remove files.

Creating Files

Let’s start by creating a file. A program can create a new file by using open() and supplying it the O_CREAT flag. The following code creates a “foo” file in the current working directory:

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

open() accepts several flags. In this case, the second parameter creates the file (O CREAT), locks it (OWRONLY), then truncates it to zero bytes, erasing any existing content (O TRUNC). Permissions are set to allow the owner to read and write to the file.

Open() returns a file descriptor, which is useful. If you have permission to read or write to the file, you can use the file descriptor to read or write it. In this sense, a file descriptor is a capability [L84], an opaque handle that grants you specific powers. To access a file, you can use “methods” like read() and write(), which are pointers to file objects.

As stated previously, the operating system manages file descriptors per process. On UNIX platforms, the proc structure contains a basic structure (e.g., an array). Here’s the relevant xv6 kernel:

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

Open files are tracked per-process using a simple array (maximum NOFILE open files). To track information about the file being read or written, each entry of the array is a pointer to a struct file.

Reading and Writing Files

We'd like to read or write files once we receive them. Let's begin by reading a file. On a command line, we could use the program cat to print the contents of the file.

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

This code redirects the output of echo to the file foo, which includes the phrase "hello". Then we use cat to view the file. But how does cat get to file foo?

To discover out, we'll utilize a powerful tool that tracks a program's system calls. The tool is called strace on Linux and other systems (see dtruss on a Mac, or truss on some older UNIX variants). strace records every system call performed by a program and displays them on the screen.

Here's an example of using strace to figure out what cat is doing:

```
strace cat foo
```

Cat opens the file for reading first. The file is only opened for reading (not writing), as indicated by the O RDONLY flag; the 64-bit offset is used (O LARGEFILE); and finally, the call to open() succeeds and returns a file descriptor with the value 3.

Why does the initial call to open() return 3 instead of 0 or 1? Standard input (which the process can read to receive input), standard output (which the process can write to dump information to the screen), and standard error (which the process can read to detect errors) (which the process can write error messages to). File descriptors 0, 1, and 2 represent them. So, if you open a new file (as cat does), it'll almost likely be file descriptor 3.

After a successful open, cat uses the read() system function to read bytes from a file. The file descriptor is the first argument to read(), indicating the file system which file to read. A process can have numerous files open at once, therefore the descriptor lets the operating system know which file to read. As shown in the system call trace above, strace shows the results of the read in this buffer ("hello"). The third option is the buffer size, which is 4 KB. The read() function also succeeds, providing the number of bytes read (6, 5 for the letters in the word "hello" and 1 for an end-of-line marker).

Here’s another fascinating strace result: a single write() system call to file descriptor 1. As stated previously, this descriptor is used to write the word “hello” to the screen as the program cat is supposed to do. But does it use write()? Perhaps (if it is highly optimized). Aside from that, cat may call the library routine printf(), which internally sorts out the formatting requirements and eventually writes to standard output to print the results. It then tries to read more from the file, but as there are no more bytes remaining, read() returns 0, and the software knows it has read the entire file. So the software runs close() to close the file “foo”, handing in the file descriptor. So the file is closed and the reading is done.

Writing a file follows a similar path. Then, if the file is large enough, the write() system function is repeated (). For example, you can use strace to track writes to a file from a program you developed or via the dd utility, e.g.

Reading and Writing, Non-Sequentially

So far, we've only explored reading and writing files in sequential order, that is, from start to finish.

If you establish an index over a text document to look up a certain term, you may wind up reading from random offsets within the document. To do so, we'll utilize lseek(). The function prototype is:

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

The initial point is known (a file descriptor). The second argument is the offset, which places the file offset within the file. The third argument, whence, controls how the seek is executed. 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 see from the above, the OS keeps a "current" offset for each file a process opens, which defines where the next read or write will begin reading from or writing to. An open file has a current offset, which can be modified in one of two ways. The first is that each read or write of N bytes updates the offset. The second is with lseek, which modifies the offset.

The offset is stored in the struct file we saw earlier, as addressed by the struct proc. Here is a (simplified) xv6 definition:

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

It may also identify whether a file is writable (or both), which file it links to (through the struct inode pointer ip), and the current offset (off). There's also a reference count (ref), which we'll get into later.

These file structures represent the system's currently open files and are collectively referred to as the open file table. An array with one lock per entry is used by the xv6 kernel:

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

Let's first track a process that opens a 300-byte file and reads it using the read() system call, each time reading 100 bytes. Here is a list of the relevant system calls, their return values, and the current offset in the open file table for this file access:

INSERT TABLE

The trace has a few interesting points. First, notice how the current offset is reset to 0 when the file is opened. This allows a process to just keep executing read() to receive the next section of the file. Finally, a read() attempt beyond the file's end returns zero, indicating to the process that it has read the file in its entirety.

INSERT TABLE

Second, trace a process that opens and reads the same file repeatedly. These descriptors (3 and 4) relate to separate entries in the open file table (in this example, entries 10 and 11, as shown in the table heading; OFT stands for Open File Table). Observe how each current offset is updated independently.
A process uses lseek() to move the current offset before reading; only one open file table entry is required (as with the first example).

INSERT TABLE

The lseek() method here first sets the offset to 200. The next read() reads the next 50 bytes, updating the current offset.

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

fork-seek.c

In many circumstances (as in the examples above), the file descriptor to open file table mapping is one-to-one. For example, a process may elect to open, read, and close a file, giving the file a unique entry in the open file table. Other processes reading the same file will have their own entries in the open file table. Each logical read or write of a file is thus independent, and each has its own current offset.

However, there are a few unusual circumstances where an open file table item is shared. When a parent process forks a child process (). In Figure 39.2, a parent creates a kid and then waits for it to finish. The child calls lseek() to adjust the current offset and exits. Finally, the parent checks the current offset and prints it.

This program’s output is as follows:

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

As shown in Figure 39.3, each process’s private descriptor array is linked to a shared open file table entry, which links to the underlying file-system inode. Here we finally use the reference count. When two processes share a file table entry, the reference count increases until both processes close the file (or exit).

Sharing open file table entries between parent and child is useful. For example, many processes working on the same task can write to the same output file without extra coordination. You may read more about process sharing in the man pages

INSERT FIGURE

Dup() (and its cousins, dup2() and dup3()) is another intriguing and perhaps more useful case of sharing.

The dup() function allows a process to establish a new file descriptor that corresponds to an existing open file. Figure 39.4 demonstrates a brief code snippet using dup().

Consider why the dup() (and dup2()) functions are useful for constructing a UNIX shell and performing actions like output redirection. What if they had told you this when you were working on the shell project? Even in an outstanding book about operating systems, you can’t get everything in order. Sorry!

dup.c

Writing Immediately With fsync()

When a software calls write(), it is usually directing the file system to write data to persistent storage at a later date. For performance considerations, the file system will buffer such writes in memory for a period of time (say 5 seconds or 30), after which the writes will be sent to the storage device. Data is only lost when the machine fails after the write() call but before the writing to disk.

Some applications, however, require more than an even- tual assurance. For example, a database management system (DBMS) requires the ability to force writes to disk.

Most file systems include extra control APIs to assist these applications. The UNIX application interface is called fsync (int fd). For each file descriptor, the file system replies by writing all dirty (i.e., unwritten) data to disk. Once all writes are complete, fsync() returns.

An example of using fsync (). A single chunk of data is written to the file foo, and then the code runs fsync() to compel the writing to disk. So long as fsync() returns true, the application can safely proceed, knowing that the data has been persisted.

```
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 always work; in some circumstances, you must also fsync() the directory containing the file foo. This step guarantees that the file is on disk and that it is part of the directory if it is newly created. Unsurprisingly, this type of oversight leads to many application-level issues.

Renaming Files

Once we have a file, it is often helpful to rename it. On the command line, use the mv command to rename the file foo to bar:

```
prompt> mv foo bar
```

strace shows that mv uses the system function rename(char *old, char* new), that accepts two arguments: the old file name (old) and the new file name (new) (new).

Rename() is an intriguing call since it is normally designed as an atomic call, meaning that if the system crashes during renaming, the file will be renamed to either the old or the new name. Thus, rename() is crucial for applications that require atomic file state updates.

Let’s get more specific. Assume you’re using a file editor (like emacs) to add a line to a file. For example, foo.txt. The editor could do the following to ensure the new file retains the original contents plus the inserted line:

```
int fd = open("foo.txt.tmp", O WRONLY|O CREAT|O TRUNC,\s S
IRUSR|S IWUSR);
write(fd, buffer, size); / write out new version of
file\sfsync(fd);\sclose(fd);\srename("foo.txt.tmp", "foo.txt");
```

In this case, the editor writes the new file to disk as foo.txt.tmp, then renames the temporary file to the original file’s name. This final step atomically swaps the new file in while deleting the old one, completing the atomic file update.

Getting File Information

INCLUDE STAT STRUCT FILE

For example, we expect the file system to preserve certain information about each file it stores. These file data are called metadata. To see a file’s metadata, use the stat() or fstat() system functions. With these calls, you can fill in a stat structure as shown in Figure 39.5.

This includes the file’s size (in bytes), low-level name (i.e., inode number), ownership information, and information on when the file was viewed or modified. Use the stat command line tool to see this data. Create a file (named file) and then use the stat command line tool to learn about it.

On Linux, the result is:

```
prompt> echo hello > fileprompt> stat file
File: ‘file’
Size: 6 Blocks: 8 IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084 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
```

An inode is a file system structure that stores this type of data. When we talk about file system implementation, we’ll learn more about inodes. For now, think of an inode as a file system’s persistent data structure containing the data seen above. All inodes are stored on disk, with a copy in memory to speed up access.

Removing Files

We now know how to create and read files, sequentially or not. What about deleting files? If you've used UNIX, you know to run the application rm. But how does rm delete a file?
Let's use strace again. Here's how to delete file foo:

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

The tracked output has been cleaned up, leaving only a single call to the oddly titled system call unlink (). As you can see, unlink() takes the file name and returns 0 if successful. Yet another puzzle: why is this system call named unlink? Why not just delete? To solve this riddle, we need to understand more about directories.

Making Directories

You can create, read, and delete folders using a set of system calls. You can’t directly write to a directory. The file system considers directory data to be meta-data, therefore you can only update a directory by creating files, directories, or other object types within it. The file system ensures that directory contents are correct this way.

A single system call, mkdir(), can construct a directory. The mkdir program can create such a directory. How about creating a basic directory called foo with the mkdir program?

```
> strace mkdir foo
...
mkdir("foo", 0777)
...
>
```

Unfilled directories are considered “empty” when created. An empty directory contains two entries: one to itself and one to its parent. (dot) directory, and the latter as “..” (dot-dot). By using the flag (-a) with the program ls, you can see the following directories:

```
> ls -a
```

Reading Directories

With our new directory, we might as well read it. That is exactly what ls does. Let’s develop our own ls-like utility and see how it works.
Instead of opening a directory as a file, we use some new calls. This software prints the contents of a directory. The program utilizes three functions: opendir(), readdir(), and closedir(), and its interface is simple: a loop reads one directory entry at a time, printing 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”, (unsigned long) d->d_ino,
d->d_name);
        }
        closedir(dp);
        return 0;
}
```

The data structure struct dirent contains the following information for each directory entry:

INSERT DIRENT STRUCT FILE

Because directories contain little information (only a name and an inode number), a program may want to run stat() on each file to find out more about it, such as its length. Try strace on ls with and without the -l parameter to see for yourself.

Deleting Directories

Finally, rmdir() can remove a directory (which is used by the program of the same name, rmdir). Unlike file deletion, deleting directories might potentially erase huge amounts of data. Thus, before using rmdir(), the directory must be empty (i.e., only have “.” and “..” entries). rmdir() will fail if you try to delete a non-empty directory.

Hard Links

Returning to the question of why removing a file is done via unlink(), we learn about a new mechanism to create an entry in the file system tree, called link (). The link() system call accepts two arguments, an old pathname and a new pathname. As seen in this example, the command-line program ln is used.

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

We generated a file called file2 with the phrase “hello” in it. The ln application then creates a hard link to that file. Then we can open file or file2 to analyze the file.
Link() just creates a new name in the directory and refers it to the same inode number (i.e. low-level name) as the original file. The file is not copied; instead, two human-readable names (file and file2) correspond to the same file. We can see this in the directory by spitting out each file’s inode number:

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

ls prints the inode number of each file when given the -i flag (as well as the file name). The link has simply made a new reference to the same exact inode number (67158084 in this example).

You may now see why unlink() is named unlink (). Making a file accomplishes two tasks. For starters, you’ll create an inode structure that will track the file’s size, location on disk, and so on. Second, you give that file a human-readable name and put it in a directory.

The original file name (file) and the newly formed file name (file2) are both links to the underlying meta- data about the file, which is stored in inode number 67158084.
Unlinking a file from the file system removes it (). In the aforementioned example, we may remove the file named file and still access the file:

```
prompt> rm file
removed ‘file’
prompt> cat file2
hello
```

This works because when the file system unlinks a file, it checks the inode number’s reference count. This reference count (also known as the link count) keeps track of how many files have been linked to this inode. When unlink() is called, it eliminates the “link” between the human-readable name (the file being destroyed) and the provided inode number, and decreases the reference count; the file system only frees the inode and accompanying data blocks when the reference count reaches zero.

Of course, stat() can show a file’s reference count. Watch how hard links to a file are created and deleted. Create three links to the same file, then delete them. Keep an eye on the number of links

```
prompt> echo hello > file
prompt> stat file
 ... Inode: 67158084
prompt> ln file file2
prompt> stat file
... Inode: 67158084
prompt> stat file2
... Inode: 67158084
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084
prompt> rm file
prompt> stat file2
... Inode: 67158084
prompt> rm file2
prompt> stat file3
... Inode: 67158084
prompt> rm file3
```

Symbolic Links

A symbolic link, or soft link, is a very valuable sort of link. It is not possible to hard link to a directory (for fear of creating a directory tree cycle); it is not possible to hard link to files in other disk partitions (since inode numbers are unique within a file system, not across file systems); etc. Thus, the symbolic link was born.

Use the same program ln, but with the -s flag. As an example:

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

It's easy to understand how constructing a soft connection is similar to creating a hard link.
Symbolic links are comparable to physical links on the surface, but they are not identical. The first distinction is that a symbolic link is a separate sort of file. Symbolic links are a third form of file system that the file system recognizes. The symlink's stats reveal:

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

Running ls confirms this. The first character in the left-most column of the long-form output from ls is a - for ordinary files, a d for directories, and a l for soft links. You can also view the symbolic link's size (4 bytes) and where it points (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 file 2 -> file
```

The reason file2 is 4 bytes is because a symbolic link uses the pathname of the linked-to file as the link file's contents. Our link file file2 is small because we linked to file (4 bytes). Our link file grows if we link to a longer pathname:

```
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, the way symbolic links are generated allows for the potential of 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
```

Unlike hard links, removing the original file causes the link to redirect to a pathname that no longer exists.

Permission Bits and Access Control Lists

Process virtualization supplied two central virtualizations: CPU and memory. In actuality, the OS used numerous strategies to share limited physical resources among competing entities in a safe and secure manner. It also provides a virtual view of a drive, changing it from a collection of raw blocks into more user-friendly files and directories. However, unlike the CPU and memory, files are commonly shared across users and processes and are not (always) private. Thus, file systems frequently have a broader set of techniques for varying levels of sharing.
The first type is the UNIX authorization bits. To see a file's permissions, type:

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

Just focus on the first half of this output, the -rw. The first character shows the file type: This is basically unrelated to permissions, so we'll disregard it for now.
The next nine characters represent the authorization bits (rw-r—r—). These bits govern who can access a file, directory, or other entity.

Permissions are divided into three categories: what the file's owner can do, what a group can do, and what anyone (also referred to as other) can do. The owner, group member, or others can read, write, or execute the file.

To summarize, the owner (rw-) can read and write to the file, while members of the group wheel and anybody else in the system can only read it (r— followed by r—).
The file's owner can easily modify these permissions with the chmod command (to change the file mode). To prevent everyone but the owner from accessing the file, type:

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

This command enables the owner's reading and write bits (4) and (2), but sets the group and other permission bits to 0 and 0, respectively, making the permissions rw———-.
The execution bit is intriguing. Its presence decides whether a software can be run. For example, to run a simple shell script named hello.csh, type:

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

If we don’t properly set the execute bit for this file, the following happens:

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

The execute bit in directories acts differently. It allows a user (or group, or everyone) to change directories (cd) into the provided directory and, with the readable bit, create files there. Play around with it to learn more! Don’t worry, you won’t screw up too terribly.

Beyond permissions bits, some file systems, like AFS (described later in this chapter), incorporate more sophisticated controls. AFS, for example, uses an ACL per directory. Access control lists represent who can access a resource in a more general and powerful way. Unlike the owner/group/everyone permissions paradigm discussed above, this allows a user to establish a highly detailed list of who may and cannot read a set of files.

For example, the following is the fs listacl command output for a private directory under one author’s AFS account:

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

These files can be searched for, inserted, deleted, and locked by both system administrators and user remzi.

In this scenario, user remzi can simply type the following command to provide access to the other author.

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

No more privacy for Remzi! There can be no secrets in a good marriage, even within the file system.

Making and Mounting a File System

We’ve now seen how to access files, directories, and URLs. Adding a whole directory tree from several file systems is also a topic worth exploring. Making file systems and mounting them to make their contents available is how this process is achieved.

This step is performed by the mkfs (pronounced “make fs”) tool. Here’s the idea: The utility simply writes an empty file system, starting with a root directory, onto that disk partition if you provide it with a device (such a disk partition, for example). That’s what mkfs said!

It must be made accessible within the uniform file-system tree once established. By calling mount(), the underlying system accomplishes this action. Mount basically pastes a new file system onto an existing directory tree.

Here’s one example. An unmounted ext3 file system with the following data is stored in device partition /dev/sda1. Assume we want to mount it in /home/users/. We’d write:

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

If the mount succeeds, the new file system is now available for use. The new file system is now accessible. This is how we would use ls to look at the root directory’s contents:

```
prompt> ls /home/users/ ab
```

See how /home/users now points to the newly mounted directory’s root. Directory a and directory b could be reached through the path names, respectively. There are another two folders named foo, which can be reached via the following paths. Mount unites all file systems with one tree, making naming uniform and simple.

Simply run the mount software to check what is mounted and where it is located. You will see:

```
/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)
```

For example, the file system tree for this machine includes ext3, the proc file system, tmpfs, and AFS (a distributed file system).

Summary

The file system interface in UNIX (and any other system) appears simple, but there is a lot to learn. Of course, nothing beats using it (a lot).

Key Terms

- You can create, read, write, and remove files. It has a low-level name (a number) that identifies it. An i-number is a low-level name.
- A directory is a set of tuples with a human-readable name and a low-level name. Each entry links to a different directory or file. Each directory also has an i-number. A directory always contains two special entries: the self-referential. and the parent-referential..
- A directory tree or directory hierarchy arranges all files and directories into a big tree.
- A process must first ask the operating system for permission to access a file (typically open()). A file descriptor is returned if permission and intent enable read or write access.
- Each file descriptor relates to an entry in the open file table. A file's current offset (i.e., where the next read or write will occur) and other pertinent information are tracked in this entry.
- Processes can use lseek() to modify the current offset, allowing random access to different regions of the file.
- A process must use fsync() or equivalent functions to update persistent media. However, doing so correctly while keeping high performance is difficult.
- Use hard links or symbolic links to have numerous human-readable names point to the same underlying file. Consider their strengths and disadvantages before using them. Remember that removing a file just unlinks it from the directory hierarchy.
- Most file systems allow you to turn sharing on and off. Permissions bits give a basic form of such control; access control lists provide more precise control over who can access and change data.