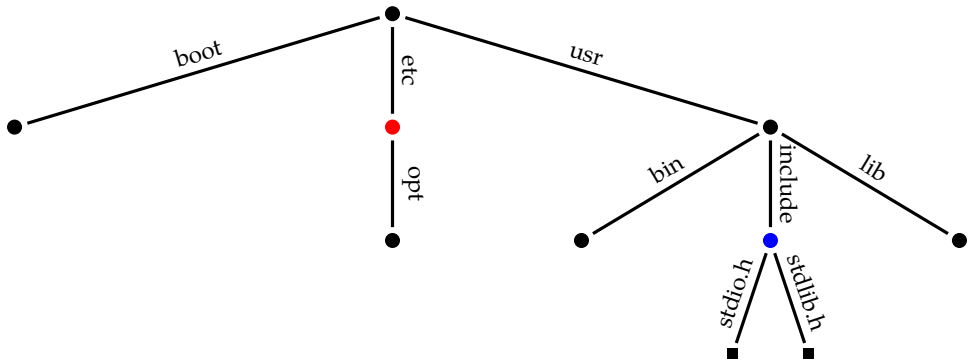
Operating Systems and Architecture (CO527) Spring 2016

Using File Systems

Fred Barnes and Radu Grigore

Tree Structure

The information in a file system is organised in a tree. Here is a (partial) tree of a UNIX-like operating system:



We can describe the path from ● to ● by '../usr/include'. The path from ● to ● could be described by '../../etc/opt/..'. The path's steps are separated by slash /. A step either goes up (..) or goes down along the named edge. (We can only go up from a ●: If a path reaches a ■, then it's stuck there.) Such paths are called *relative paths*, and their starting point is the *current working directory* of the process. *Absolute paths* use the root of the tree as their starting point, rather than the current working directory. Absolute paths are described by similar strings, except the strings start with a slash. For example, both /etc and /etc/opt/.. are absolute paths to ●.

The circles ● represent *directories*, which are files that have children. The downward pointing edges represent *directory entries*. The squares ■ represent non-directory files, which are leaves of the tree. Normal files support read and write operations.

On POSIX operating systems there is only one file system tree. If you know that one computer may have both a hard drive and a DVD, then you may wonder how come there could be only one tree. To answer this, take a look in the /dev directory:

```
rg@rg-2016:notes$ ls /dev/sda*
/dev/sda  /dev/sda1  /dev/sda2  /dev/sda3  /dev/sda4  /dev/sda5  /dev/sda6  /dev/sda7
```

The file /dev/sda contains all the content of a hard drive. The other ones are different partitions on the same hard drive. But, how do we see the files in one of those partitions? We *mount* that file:

```
mount /dev/sda1 /some/dir
```

The /some/dir must exist prior to running the command. After the command is run, /some/dir will contain the files on the partition /dev/sda1.

So, what's going on here? The operating system shows us a *virtual file system*. Files appear and disappear in the directory /dev depending on which hardware devices the operating system detects. The operating system gives us access to a wide range of devices using a uniform interface for interacting with files. We can tell the operating system, as we did above, that some of these devices contain real file systems, and thus grow subtrees in the unique virtual file system tree.

Security

POSIX operating systems support at least the following basic system of permissions. Suppose we list details about a file:

```
rg@rg-2016:notes$ ls -l 03-file-use.tex
-rw-rw-r-- 1 rg rg 3538 Mar 18 17:35 03-file-use.tex
```

The first column contains three sets of permissions, namely (1) the user permissions rw-, (2) the group permissions rw-, and (3) the other permissions r–. Each of those is a subset of rwx: r means 'allowed to read', w means 'allowed to write', x means 'allowed to execute'. Given a set of permissions, it is pretty clear what we are allowed to do. But, there are *three* sets. Which one are we supposed to use? This is determined as follows. Each file has an owner and a group: these are listed in the columns 3 and 4 above, and both are rg. Each user belongs to a set of groups; for example:

```
rg@rg-2016:notes$ groups
rg adm cdrom sudo dip plugdev lpadmin sambashare
```

If the owner of a file tries to access it, then the set (1) of permissions is used. Otherwise, if the user accessing the file belongs to the group of the file, then the set (2) of permissions is used. Otherwise, the set (3) of permissions is used.

Apart from permissions, one could also use encrypted file systems.

Reading and Writing Files

POSIX defines two interfaces for interacting with files, a high-level one and a low-level one. Here is an example of using the high-level interface:

```c
#include <stdio.h>
int main() {
  printf("Hello world!\n");
}
```

And here is an example of using the low-level interface:

```c
#include <unistd.h>
int main() {
  write(STDOUT_FILENO, "Hello world!\n", 13);
}
```

The high-level interface prints a string to the terminal. The low-level interface makes it clear that printing a string is actually done by writing to a special file, following the 'everything is a file' philosophy of POSIX. The low-level interface also makes it clear that we want to write 13 bytes. Apart from forcing us to specify some uninteresting details, the low-level has yet another quirk: there is no guarantee that the string is actually printed. If we want to make sure that 'Hello world!' is printed then we have to use a loop:

```c
#include <unistd.h>
int main() {
  const char * buf = "Hello world!\n";
  int written = 0;
  while (written < 13) {
    written += write(STDOUT_FILENO, buf + written, 13 - written);
  }
}
```

Why would write return before it writes 13 bytes? Because write cannot guarantee both (a) that it writes all bytes, and (b) that it returns quickly. The design choice for the low-level interface is to prefer to return quickly. The design choice for the high-level interface is to write all bytes. A similar trade-off is made for reading.

In both the high-level and the low-level interface, one interacts with files in three phases:

1. open the file to obtain a *file handle*
2. read from the file, write to the file, repeatedly
3. close the file

For the high-level interface, the file handle is called a *stream*, and it is a pointer to a FILE structure. For the low-level interface, the file handle is called a *descriptor*, and it is a nonnegative int. Thus, the high-level interface is sometimes called the stream-based interface, while the low-level interface is sometimes called the descriptor-based interface.

Let us implement a program that copies a file, using the descriptor-based interface. We open and close the files as follows:

```
int in_file = open(argv[1], O_RDONLY);
int out_file = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, 0664);
// TO BE FILLED
close(in_file);
close(out_file);
```

The function open takes a path, which can be absolute or relative, and returns a file descriptor. Above, we assume that the path of the source is in argv[1] and the path of the target is in argv[2]. The function open takes a second argument which specifies how the file should be opened. O_RDONLY means we will only read; O_WRONLY means we will only write; O_CREAT means that the file should be created if it does not exist; O_TRUNC means that the content of the file should be truncated (erased). The second argument is a bitmask, so we can combine flags with the bitwise logical-or operator. Optionally, the function open takes a third argument. The third argument says which permissions should be used in case the file is created. Recall that we need three sets of permissions, each set being specified by three bits. The literal 0664 is an *octal* number, because it starts with 0. (This is a lexical convention used by C and by many other languages.) Each octal digit corresponds to . . . three bits. So, each digit specifies one set of permissions: 0664 stands for `rw-rw-r--`.

After opening and before closing the files, we do the actual copying:

```
const int buffer_size = 1 << 20; // 1 MiB
char buffer[buffer_size];
while (1) {
  ssize_t r = read(in_file, buffer, buffer_size);
  if (r == 0) break;
  ssize_t w = 0;
  while (w < r) w += write(out_file, buffer + w, r - w);
}
```

You can find the full code in the file copy.c.

We can copy files using four functions:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

The function open gives us a file descriptor, provided a pathname in the virtual file system. The function close closes a file descriptor. The functions read and write let us transport data to and from the file.

Synchronous I/O Multiplexing

Suppose a process needs to read several files in parallel. It could be that several files are ready to be read while others are not. This would happen, for example, if we would implement a server and the files were network connections. If the process calls read to get data from a file that has no data ready, then the process's

execution would be blocked until some data becomes available. In the meantime, the other files, which may have data ready, are ignored. How should we handle such a situation? One solution is to have multiple threads, each thread reading from one file only — this is *asynchronous I/O multiplexing*. Another solution is to first ask which files have data ready — this is *synchronous I/O multiplexing*.

Let us see how synchronous I/O multiplexing works on an example. We shall avoid using the network for now. Instead, we use a special kind of file called 'fifo file' or 'named pipe'. Let's create two of these:

```
rg@rg-2016:files$ mkfifo one two
rg@rg-2016:files$ ls -l one two
prw-rw-r-- 1 rg rg 0 Mar 19 17:16 one
prw-rw-r-- 1 rg rg 0 Mar 19 17:16 two
```

We will put data into these files using the cat command. In one terminal we run cat - > one, in another terminal we run cat - > two. We'd like a to write program that is run by

```
rg@rg-2016:files$ ./counter one two
```

and periodically prints how many bytes came through one and two together, *no matter in which order we type in the two terminals*. You can see a complete solution in the accompanying counter.c. Here, let us look at its main bits.

The main loop looks as follows:

```
while (1) {
  // <Setup nfds and all>
  select(nfds, &all, 0, 0, 0);
  // <Read from all files in all>
}
```

The variable all is a set of file descriptors from which we wish to read. The function select returns when there exists some subset of file descriptors on which we can call read without a blocking fear. Moreover, select modifies the set all, so that it now contains only those files on which it's safe to call read.

Before entering the loop from above, we first open all files of interest.

```
int files[n];
for (int i = 0; i < n; ++i) files[i] = open(argv[i+1],O_RDONLY);
```

At the beginning of each iteration of the main loop, we set up all by inserting into it all file descriptors we still have. We will make sure that each element of files is either a file descriptor of an open file, or it is -1.

```
fd_set all;
FD_ZERO(&all);
for (int i = 0; i < n; ++i) if (files[i] != -1) FD_SET(files[i], &all);
```

(We also need to set ndfs to $1 + \max$ all. That's easy. See counter.c if you don't think so.)

After the call to select, we look at each file descriptor still in all, and we process it. Normally, we just read from it and count the bytes. But, it could also be we reached the end of file, in which case we close the file.

```
for (int i = 0; i < n; ++i) if (FD_ISSET(files[i], &all)) {
  ssize_t r = read(files[i], buffer, buffer_size);
  if (r == 0) { // end of file reached
    close(files[i]);
    files[i] = -1;
  }
  total += r;
  // <Report total>
}
```

The stream-based interface does not support synchronous I/O multiplexing.

Exercises

1. ► What is an absolute path of ●?
2. Read 'man path_resolution'.
3. The file system tree is actually a directed graph. Read about file system links (hard and soft).
4. Read 'man mount'.
5. Which command is used to change the owner of a file? Which command is used to change the group of a file? [Hint: Google.]
6. Which of the rwx permissions on a directory foo are needed to be able to change the current directory to a subdirectory of foo? [Hint: Just try it.]
7. ► Implement a function

```
boolean hasAccess(
  char accessType,
  String user, String[] userGroups,
  String fileOwner, String fileGroup, String permissions);
```

   For example, the call

```
hasAccess('w', "rg", new String[]{"foo","staff"}, "frmb", "staff", "rwxr-----");
```

   should return false. You may use any language (but you'll have to adapt the function signature).
8. The 'Hello world!' examples do not handle errors properly. See man printf and man 2 write for which errors could occur, and change the program so that such errors are gracefully reported. At the very least, ensure that the program does not crash in the event that an error occurs.
9. Can you think of a situation in which the return-early behaviour of the low-level file interface is preferable? [Hint: Think of scanf versus read.]
10. On Linux, you can see which files are open using the lsof command. How many files are open on your system right now? Run man lsof, search for 'TYPE', to see how many types of files there are.
11. ► What is a file handle? What is a file descriptor? What is a stream?
12. Another interesting operation on regular files is seeking. Read about it by running man lseek.
13. Implement counter.c using asynchronous I/O multiplexing. What extra complication arises?
14. Read 'man select_tut'.
15. ► In which situation would you use synchronous I/O multiplexing?

References

[1] Linux Foundation, Filesystem Hierarchy Standard, 2015

[2] General Concepts, POSIX, 2013