Next Previous Contents

6. File names and files

6.1 The file hierarchy

File naming in a Unix-like system can be described from the user space point of view and from the kernel point of view. On a Unix-like system, a user or user program sees a single file hierarchy, rooted at `/`. A full pathname of a file, like `/usr/games/lib/hack/scores` describes the path from the root of the tree to the given file. The *current working directory* is a marked node in the tree. Pathnames not starting with `/` are relative with respect to the cwd. (Thus, if the cwd is `/home/aeb`, then `bin/findgr` refers to `/home/aeb/bin/findgr` and `../asm` refers to `/home/asm`.) The name `.` (when initial in a pathname) refers to the cwd. The names `.` and `..` relative to some directory D refer to D and the parent directory of D, respectively.

With the original Unix, the empty string was a name for the current working directory. POSIX made this illegal.

That was the user picture. That files live in different filesystems on different devices is to a first approximation invisible. (If one looks more closely, then an NFS-mounted filesystem does not quite have Unix semantics. FAT filesystems do not have Unix attributes like owner and permissions. Etc.)

If she has the appropriate capabilities, the user can modify the big file hierarchy by mounting or unmounting block devices. Given a Zip disk or a compact flash card, or a floppy, or a CD or so, a command like

```
mount -t vfat /dev/sda4 /zip
```

will attach the file hierarchy found on `/dev/sda4` to the node named `/zip`, so that after this call `/zip/a` refers to the file `/a` on this disk, and `/zip` itself refers to `/` on this disk. In particular, owner and permissions of `/zip` will in general change. (Formulated differently: no inodes change, but after the mount the name `/zip` names a different inode.) The directory `/zip` is called the *mount point*.

Note that if the mount point was a nonempty directory, the files below it are no longer accessible using full pathnames through the mount point: If there was a `/zip/a` before the mount command, then this file can no longer be accessed by this name after the command. On the other hand, if some user process had current working directory `/zip` or `/zip/b`, this file will still be accessible using the name `./a` (or just `a`) and `../a`, respectively.

We see that the current working directory really is an inode, not a string.

Altogether, the interpretation of a filename depends on the root directory, the current working directory, and the list of mounts. The current working directory is private for each process, and a process can change its current working directory using the chdir() system call.

In ancient Unix, the root directory was the same for all processes. However, 4.2 BSD made the root directory private for each process, just like the current working directory, and introduced the chroot() system call. The chroot system call serves to specify which directory is pointed at by the users' `/`. It is often used in security-conscious environments, say by an ftp server, where the purpose is to keep anonymous users inside the *chroot jail*.

This gives a new example of a name space with interesting topology: the chroot() system call does not change the current working directory, so `.` can now lie outside the tree rooted at `/`.

In all traditional Unix versions, the list of mounts is global. Plan9 introduced the idea of namespaces where each process could see an entirely different tree (or forest) because also the mounts are private for each process. Linux is slowly following this example. Currently a special version of the fork() system call will fork off a child with a private copy of the name space of its parent, so that subsequent chdir() or chroot() or mount() calls in one do not influence the other. More details later.

Filesystem character set

So far we have not given any thoughts to the question how a filename is coded. What sequence of bits or bytes is used for a file called "/bin/ls"?

ASCII

The classical Unix answer is that filenames are in ASCII, mostly because everything is in ASCII. The C statement

```
printf("Hello world!\n");
```

uses a string of which the first byte is `'H'` with value 0110 (octal), that is, 72 (decimal), that is, 0x48 (hexadecimal), that is 01001000 (binary). Inside the program it doesn't matter what the corresponding glyph looks like, but as soon as this value is printed we recognize the shape as that of an H.

If the string is used as a filename, it will be stored on disk in some way, such that a later lookup finds the same string again.

ISO 8859-1

Bytes have 8 bits, and when there no longer was any need to use the high order bit as (anti)parity bit, more code points became available, and Western Europe started using ISO 8859-1 (also called Latin-1), an extension of ASCII in which the high order half is used for

national characters, so that the Swedes can have *knäckebröd* and the Danes *rødgrød med fløde*. This is still the default character set on the WWW. But then in Central Europe other symbols were needed and ISO 8859-2 was defined, and the Russians invented KOI-8, and the Japanese Shift-JIS and the Chinese GB5, and Western Europe needed the Euro, etc. etc. The international situation became as bad as the situation in the English speaking world had been before ASCII, with hundreds of different codes.

Exercise *The above recipe for "rødgrød med fløde" is in ISO 8859-1. What happens if you tell your browser that it is ISO 8859-2? ISO 8859-15? What happens when you paste the text into an xterm?*

Unicode

For a global character set the situation in Japan and China is worst. Both countries have a large number of characters - large dictionaries may give 50000 or more. If one byte suffices for $2^8 = 256$ symbols, two bytes suffice for $2^{16} = 65536$. Enough for China, or for Japan, but not enough for both. However, the Japanese character set is derived from the Chinese one, and many characters are the same, or almost the same. A committee did the "unification" of both alphabets, and *Unicode* was born: a 16-bit character set with symbols for every character used in any of the world's languages. These days the whole world is slowly converting to Unicode. RedHat 8.0 is the first Linux distribution that uses Unicode by default.

UTF-8

The body of Unix programs is far too large to start rewriting everything. But the 16-bit Unicode values do not fit well into the traditional Unix scheme with strings terminated by a NUL-byte. Indeed, in Unicode 'H' is coded U+0048, and at first sight that consists of the two bytes 00 and 48 (hex). To solve this problem the UTF-8 scheme of coding Unicode was developed. The convention is this:

```
0xxxxxxx
110xxxxx 10xxxxxx
1110xxxx 10xxxxxx 10xxxxxx
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
```

The x-es are the bits to be coded, and the code is shown. Values in the range 0-127 are coded by a single byte. Values below 2^{11} are coded by two bytes. Values below 2^{16} are coded by three bytes. Etc. Thus, hex ABCD, that is, binary 1010101111001101 is coded by the three bytes 11101010 10101111 10001101.

The advantage of this code is that the only way bytes in the range 0-127 occur is as themselves. Never as part of a multi-byte code. Very convenient for Americans: the

conversion from ASCII to Unicode coded in UTF-8 is trivial: do nothing. This code is called FileSystem-Safe because NULs and / and . characters, the only bytes the kernel looks at when handling filenames, do not occur in unexpected ways.

In Western Europe one has a few accented letters and these now require 2 bytes. That is not serious. In China and Japan everything that used to take 2 bytes now takes 3 bytes, and people are unhappy. So, UTF-16, the 16-bit encoding of Unicode is more popular there. (Apart from the fact that some people in Japan are unhappy because of the unification.)

Java uses Unicode internally and makes locale-dependent filenames.

```
% cat > pi.java
import java.io.*;

class Main {
        public static void main(String[] args) {
                String s = "\u03c0";    // pi
                try {
                        FileWriter out = new FileWriter(s);
                        out.write(s);
                        out.flush();
                        out.close();
                } catch (IOException ee) {
                        System.out.println("Error writing file "+s+": "+ee);
                }
        }
}
% javac pi.java
% java Main
% ls
?  Main.class  pi.java
% LC_ALL=nl_NL.utf8
% export LC_ALL
% java Main
% ls
Main.class  pi.java  π
```

Here the letter pi was converted into a question mark in the C locale, and into cf 80 (hex) in an utf8 locale. Note that cf 80 is binary 11001111 10000000 and hence codes for 01111000000, that is U+03c0. On a uxterm the filename is shown with a nice Greek pi.

Many Microsoft filesystems encode information about the character set the filenames are in. Unix does not do that, and does not really view filenames as sequences of characters - it views filenames as sequences of bytes.

6.2 The Unix filesystem model

A Unix filesystem has *inodes* as basic entities. An inode (index node) contains information about the type, owner, size, permissions and timestamps of the file, and points to where on

disk this file lives. The inode has room for a short list of block numbers, so if the file is small this suffices, but otherwise *indirect blocks* are used, that themselves contain a list of block numbers. If this still does not suffice *doubly* and *triply* indirect blocks are used.

A file is nothing but an inode. In particular, a file is not characterized by a name. There are special files, called *directories*, that contain names and the numbers of the corresponding inodes. From a name one can find, by a lookup process, which inode number belongs to that name. From an inode one cannot find a name. There may be zero, one, two, or more names for the same file.

The filesystem keeps track of the number of names a file has, and the system keeps track of the processes that currently have an open file descriptor for the file. When the last name is removed from the filesystem, and all file descriptors for the file have been closed, the blocks that the file occupied can be put onto the free list.

Links

Names of a file are also called *links* to that file. Or *hard links* if it is necessary to distinguish them from *soft links*. All hard links are equivalent.

A soft link, or symbolic link, symlink in short, is a very different animal. It is a very small file that contains the name of another file, and has the property that the system name lookup process will usually automatically follow this redirection. All hard links to a file live in the same filesystem. Symlinks to the file can live in different filesystems. The filesystem keeps track of the number of hard links to a file, but not of the number of symlinks. If you remove the file a symlink pointed to, the symlink becomes a *dangling* symlink.

Since the Unix file hierarchy should be a tree, there should be precisely one hard link to a directory. Symlinks however can point at arbitrary path names. A file tree walker should be careful not to follow symlinks (or otherwise keep track of all files visited).

Device special files

The Unix philosophy is: "everything is a file". That makes life easy, the same system calls are used to read from any device. Special device nodes are inodes that refer to a device rather than to a file. They come in two kinds: block special devices are block-structured, allow random access, and, in case they contain a filesystem, can be mounted. The typical example is a disk. All (other) devices are character special devices. One has tapes, scanners, modems, etc. Also block special devices can be often accessed via a character special device node.

Device special files are very simple: the inode contains a pair of small integers (the major and minor device numbers), and these numbers are used by the system as index in some table to find the driver for the device.

Other file types

Depending on the Unix version there may be other types of files other than regular files, directories, and device special files. Most versions know about fifos, symlinks and sockets. Try `man 2 stat` on a Linux system for a list of types.

Sparse files

A Unix filesystem may have sparse files: files with holes. The holes can be read, and then read as all zeroes. They do not take (much) space: the actual data blocks are not written. Sometimes this is important when space is scarce, like on a rescue floppy.

A demo (error handling removed for brevity):

```
/* mkhole.c */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

main(int argc, char **argv){
        char *filename = "AB";
        int fd = open(filename, O_RDWR|O_CREAT|O_TRUNC, 0666);
        int offset = (argc > 1) ? atoi(argv[1]) : 1000000;
        struct stat statbuf;

        write(fd, "A", 1);
        lseek(fd, offset, SEEK_SET);
        write(fd, "B", 1);
        close(fd);
        stat(filename, &statbuf);
        printf("The file %s has size %lld and takes %lld sectors\n", filename,
               (long long) statbuf.st_size, (long long) statbuf.st_blocks);
        return 0;
}
```

This will create a file with one byte 'A', then a hole, then another byte 'B'. Look at the size of the resulting file (measured in 512-byte sectors):

```
% ./mkhole 1
The file AB has size 2 and takes 2 sectors
% ./mkhole 1023
The file AB has size 1024 and takes 2 sectors
% ./mkhole 1024
The file AB has size 1025 and takes 4 sectors
% ./mkhole 12287
The file AB has size 12288 and takes 4 sectors
% ./mkhole 12288
The file AB has size 12289 and takes 6 sectors
% ./mkhole 274431
```

```
The file AB has size 274432 and takes 6 sectors
% ./mkhole 274432
The file AB has size 274433 and takes 8 sectors
% ./mkhole 67383295
The file AB has size 67383296 and takes 8 sectors
% ./mkhole 67383296
The file AB has size 67383297 and takes 10 sectors
% ./mkhole 2147483646
The file AB has size 2147483647 and takes 10 sectors
% ls -l AB
-rw-r--r--    1 aeb      users    2147483647 Sep 28 21:51 AB
% du AB
5       AB
% ./mkhole 2147483647
File size limit exceeded
```

Apparently this filesystem uses 1024-byte blocks, and has a maximum file size 2147483647. The file of this maximum size, but with a giant hole, takes 5 blocks.

6.3 The Linux filesystem model

Most of what was said earlier about the Unix filesystem holds for the Linux situation. But there are two important differences. On the one hand, there is a new kind of objects, *dentries*. On the other hand the situation with one large file hierarchy is left. Every process may see a different hierarchy. Also, a single filesystem may be visible several places in the tree.

Dentries

The Unix name resolution process resolves a name into an inode, and no trace of the name remains. In particular, a working directory is known only as inode, and a command like `pwd` has to go to great trouble (stat the current directory, find device and inode number, then `chdir("..")`, stat all entries in that directory until an entry with the same device and inode number is found; if found we know the last component of the name; now repeat), and may not find the name, for example because of problems with permissions.

Linux does things in two stages. First the name is resolved into a *dentry* ("directory entry"). Then the dentry is resolved into an inode. Some names are needed very frequently, such as the files used by the C library. A dentry cache speeds things up. Having dentries allows Linux to do what Unix could not, namely implement a `getcwd()` system call.

Exercise Play with `pwd` and `getcwd()` in several situations.

(i) Explain the following. (This is with `bash`.)

```
% pwd
/home/aeb
% ln -s /usr/bin ub
% cd ub
```

```
% echo $PWD
/home/aeb/ub
% pwd
/home/aeb/ub
% ls -ld ../bin
drwxr-xr-x    3 root     root        44528 2002-10-27 18:56 ../bin
% cd ..
% ls -ld bin
drwxr-xr-x    3 aeb      users        3072 2002-10-26 22:31 bin
% pwd
/home/aeb
```

How come that `bin` has different owner and size? Try the same after first doing

```
% set -o physical
%
```

(ii) `pwd` may give the wrong answer, or no answer at all. Try

```
% mkdir aa
% cd aa
% mv ../aa ../bb
% echo $PWD
/home/aeb/aa
% pwd
/home/aeb/aa
% /bin/pwd
/home/aeb/bb
```

Try the same after first doing

```
% set -o physical
%
```

and see that `pwd` is silent.

(iii) The above shows that there is a layer of user space software between the kernel image of things and the user. For testing it may be less confusing to write a small C program that more directly (there is still the C library) reports what the kernel says.

```
/* getcwd.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char buf[4096];

int main(){
        char *cwd = getcwd(buf, sizeof(buf));

        if (!cwd) {
                perror("getcwd");
                exit(1);
```

```
        }
        printf("%s\n", cwd);
        return 0;
}
```

Now do the same things using this utility `getcwd`.

```
% mkdir aa
% cd aa
% getcwd
/home/aeb/aa
% mv ../aa ../bb
% getcwd
/home/aeb/bb
% rmdir ../bb
% getcwd
getcwd: No such file or directory
```

This shows that the current working directory of a process can be deleted (when it is empty). The inode still exists as long as the process is there, but the name has gone away.

Mounts

Mounting a filesystem at a pathname means that that path now points to the root of that filesystem, instead of at the part of the tree it used to point at. This is Unix semantics. It means that a new inode will be associated with the path after the mount. In particular, the permissions of the directory (it usually is a directory one mounts on) may well change. It also means that part of the old tree may have become invisible to everybody, except for processes that had their working directory below the moint point.

This already shows that also classical Unix knows about the possibility of a file hierarchy that is a forest instead of a tree, more or less as an accident. Linux allows many things unknown in classical Unix. A "bind" mount attaches an already mounted filesystem somewhere else, so that a filesystem can be multiply mounted. (And in such a situation multiple umount calls will be required.)

Exercise *Play with* `mount` *in several situations. You may have to be root.*

```
# mkdir tmp
# mount --bind /tmp ./tmp
```

Now `./tmp` and `/tmp` are different names that refer to the same subtree.

```
# cd tmp
# umount /home/aeb/tmp
umount: /home/aeb/tmp: device is busy
# cd /tmp
# umount /home/aeb/tmp
#
```

This shows that the current working directory is not just an inode. Although `tmp` and `/home/aeb/tmp` are two names for the same directory, using one keeps the mount busy, using the other doesn't.

```
# cd /home/aeb/tmp
# mount --bind /tmp /home/aeb/tmp
# getcwd
/home/aeb/tmp
# ls
.  ..
# ls /home/aeb/tmp
.  ..  .X0-lock  .X11-unix
# umount /home/aeb/tmp
```

What happens here? The current working directory disappears from view since it is overmounted. This shows that the current working directory is not just a string: `ls .` and `ls /home/aeb/tmp` give different results because `.` and `/home/aeb/tmp` are different directories (hence have different inodes). One can check the inode number of a file using `ls -id file`.

```
# mkdir /home/aeb/foo
# mount --bind /tmp /home/aeb/tmp
# mount --move /home/aeb/tmp /home/aeb/foo
# ls /home/aeb/foo
.  ..  .X0-lock  .X11-unix
# umount /home/aeb/foo
```

This shows that it is possible to move mounted trees around.

Per-process namespaces

Since 2.4.19/2.5.2, the `clone()` system call, a generalization of Unix `fork()` and BSD `vfork()`, may have the CLONE_NEWNS flag, that says that all mount information must be copied. Afterwards, mount, chroot, pivotroot and similar namespace changing calls done by this new process do influence this process and its children, but not other processes. In particular, the virtual file `/proc/mounts` that lists the mounted filesystems, is now a symlink to `/proc/self/mounts` - different processes may live in entirely different file hierarchies.

Since 2.6.16 one can get an own namespace without starting a new process using the `unshare()` system call.

Exercise *Play with this in several situations. You may have to be root.*

```
/* newnamespace.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sched.h>
```

```
#ifndef CLONE_NEWNS
#define CLONE_NEWNS 0x00020000
#endif

int childfn(void *p) {
        setenv("PS1", "@@ ", 1);
        execl("/bin/ash", "ash", NULL);
        perror("execl");
        fprintf(stderr, "Cannot exec ash.\n");
        return 1;
}

int main(){
        char buf[10000];
        pid_t pid, p;

        pid = clone(childfn, buf+5000, CLONE_NEWNS | SIGCHLD, NULL);
        if ((int) pid == -1) {
                perror("clone");
                exit(1);
        }

        p = waitpid(pid, NULL, 0);
        if ((int) p == -1) {
                perror("waitpid");
                exit(1);
        }

        return 0;
}
```

This program starts with a namespace that is a copy of the global one, and then forks off a shell (`/bin/ash`, a simple shell, easier to handle than `/bin/sh`) and waits for it to finish. If this is under X, you can give a few commands `xterm &` to ash so as to start some shells in the new namespace. Check that mounts and umounts done in the new namespace are invisible in the old namespace.

BUGS: The program `mount` does not know about this feature yet, so updates `/etc/mtab`. Reality is visible in `/proc/mounts`.

Example of the use of `unshare()`:

```
/* Start a shell in a new namespace */
#include <stdio.h>
#include <sched.h>
#include <unistd.h>

int main() {
  if (unshare(CLONE_NEWNS | CLONE_FS | CLONE_FILES | CLONE_VM) < 0)
    perror("unshare");
  else if (execl("/bin/sh", "/bin/sh", (char *) 0) < 0)
    perror("cannot exec /bin/sh");
```

```
  return -1;
}
```

Shared and private mounts

Since 2.6.15, the tree of mounts has gotten more structure describing what mount propagation is to take place between subtrees cloned via a `mount --bind`.

Earlier, `mount --bind` would make (part of) a single filesystem tree visible at some different point, and `mount --rbind` would do this recursively, also making submounts visible in the new place, but afterwards no mount propagation would take place. This old setup is still the default. One can ask for it explicitly by use of `mount --make-private`.

The opposite is `mount --make-shared`. When a shared mount is bound elsewhere, submounts propagate.

```
# cd /home/aeb
# mkdir tmp foo
# mount --bind /tmp tmp
# mount --make-shared tmp
# mkdir tmp/usrlocal tmp/usrsbin
# mount --bind /usr/local tmp/usrlocal
# ls tmp/usrlocal
.   bin  etc    include  lib      man       sbin   src
# mount --bind tmp foo
# ls foo/usrlocal
.  ..
# mount --bind /usr/sbin tmp/usrsbin
# ls foo/usrsbin
<long list>
# umount foo
umount: /home/aeb/foo: device is busy
# umount foo/usrsbin foo
# mount --rbind tmp foo
# ls foo/usrlocal
.   bin  etc    include  lib      man       sbin   src
# umount foo/usrlocal
# ls tmp/usrlocal
.  ..
```

This shows how `mount --bind tmp foo` does not make the submount `tmp/usrsbin` visible under `foo`, while `mount --rbind tmp foo` does. Also, that since `tmp` is shared, mounts made to it after `mount --bind tmp foo` propagate to `foo`.

Apart from private and shared (a symmetric relation), there is also `mount --make-slave`. If at first two mounts A and B are shared, then after `mount --make-slave B`, mount propagation goes from A to B, but not from B to A.

Finally, there is `mount --make-unbindable` that says that a mount cannot be the source of a `mount --bind`, and must not be bound as a result of a `mount --rbind` of some parent directory.

These calls have recursive versions `--make-rprivate`, `--make-rshared`, `--make-rslave`, `--make-runbindable` that recursively change the state of all submounts that exist at the moment of invocation.

The state of a mount point (private, shared, slave, unbindable) determines what happens at the moment a bind mount occurs. But afterwards that state can change again. For example, a slave mount can be made shared again. Now it retains the slave relations it had, but when it is bound, it will share with that new mount. In other words, we have equivalence classes of mounts that propagate to each other, and on the set of equivalence classes there is a directed acyclic graph describing the slave relation.

It can be considered a bug that `--make-private` does not reset the `--make-unbindable` flag; one has to do a `--make-shared` first.

See also ibm-developerworks on namespaces.

6.4 Open files

In order to use a file (not just its name), one has to *open* it. The result of the `open()` system call is a small integer, called a *file descriptor*, index in a per-process table of open files (also called *file descriptions*) and subsequent read and write accesses will use this integer, often called `fd`.

A funny part of the socket API is that one can transmit file descriptions over a socket, using the ancillary data part of the `sendmsg()` and `recvmsg()` system calls. The sender opens a file, gets file descriptor 5 referring to the corresponding open file description, and sends the integer 5 using `sendmsg()`. The receiver already had open files numbered 0-8, and when he does `recvmsg()`, he gets the value 9 as file descriptor referring to the same open file.

Example program.

6.5 Path names relative to a file descriptor

There are various reasons why relative path names are needed on one hand, and are difficult to use on the other hand. In order to solve such problems, new system calls have been added.

Length

Path names have a maximum length, but file hierarchies do not have a maximum depth. That means that there will be files that cannot be opened by their absolute path names. One will

have to use `chdir()` to get deep enough to use a relative path name.

Races

There is a race implicit in the use of a path name: a component of the path could be changed at the same time. (This is a well-known hacker technique: quickly switch a symlink in `/tmp`.) It is avoided by first going to the right directory, and then using a relative filename.

Threads

On the other hand, all threads in a process share the same current working directory. That means that an `open()` with relative path name in one thread, and a simultaneous `chdir()` in another thread leads to undefined results.

`*at()` system calls

Linux kernel 2.6.16 introduces the system calls `openat()`, `mknodat()`, `renameat()` etc. - Solaris already had them. These have for each pathname parameter an additional parameter that gives the file descriptor for a directory relative to which a relative path name is to be interpreted. Such a file descriptor can be checked with `fstat()`, and races are avoided.

Next Previous Contents