

Theres a Lot in the Dot: Filesystem Permissions and Pathnames

Jerry Peek

Beginning with a brief tour of the filesystem, we take you deep into the dot (.) of your directory listing so you can get the most from your CLI.

Linux filesystem permissions and pathnames can cause lots of confusion. Users type long pathnames when a short one would do as well, and set wide-open access permissions when a more-secure setting would be easy if the concepts were more clear.

This article and the next take a unique approach: understanding the dot (.) that you've probably seen in directory listings.

What's "in" a Directory

Linux filesystems hold files and other objects (e.g. symbolic links, devices like disks, and directories) grouped into directories (known as folders on other operating systems). It's important to understand that a directory doesn't actually contain the objects; it holds references to those objects. The references are called hard links.

To access something in the filesystem, a program looks into a directory using a system call. From there, it finds the link to the actual object on the disk (or whatever device the object is stored on). Then it locates the object through its index number, as we'll see soon.

Each directory has at least two names. One of those names is a dot (.). Every directory on the filesystem has a name (a hard link) of dot.

Most GUI filesystem browsers don't show these dot entries. You can see them with the ls utility and its -a ("all") option.

By default, if you don't specify something to list, ls lists its current directory. You can also specify a directory, like ls /bin or ls .. (to list the dot-dot entry). Adding the -i or --inode option shows the index numbers. This directory has a parent and "grandparent" directory, as we'll see later. And pwd shows the directory's absolute pathname:

```
$ # Listing I (child directory)
$ ls
bin foo
$ ls -a
. .. bin foo
$ ls -ai
5423 . 58 .. 5425 bin 5424 foo
$ ls -ai .
5423 . 58 .. 5425 bin 5424 foo
$ pwd
/home/jpeek
```

Notice that ls -a and ls -a . give the same result. Both list the directory named dot — the current directory. If your ls shows slashes after the names of some entries, it's been aliased to use the option -F. For example:

```
$ ls -aF
./ ../ bin/ foo
```

Theres a Lot in the Dot: Filesystem Permissions and Pathnames

Jerry Peek

The listings show the directory's four entries (hard links): ., .., bin and foo. The first three are links to directories, and the fourth is a plain file. Each entry has its index number; for instance, this directory's index number (listed before its dot entry) is 5423.

If you list various directories around the filesystem, you'll see that every directory has an entry named .. They also have .. (two dots), which is a link to that directory's parent — in other words, the directory that contains the directory. Let's list the parent directory in two ways: by telling ls to list .., then by changing the shell's current directory to .. and listing the current directory:

```
$ # Listing II (parent directory)
$ ls -ai ..
 58 . 2 .. 5423 jpeek
$ cd ..
$ ls -ai
 58 . 2 .. 5423 jpeek
$ pwd
/home
```

Please compare Listing II with Listing I. What do they have in common?

- Both listings show a hard link with index number 5423. In Listing I (the child directory), that entry is named . (dot). In Listing II (the parent), the link to the same directory (index 5423) is named jpeek.
- Both also have an entry for index number 58. In the child directory, it's named .. (dot dot), and in the parent it's named . (dot).

So, the same directory can be referred to in more than one way:

- When you're in the directory itself (when it's your current directory), you can call it . (dot).
- When you're in its parent directory, you call it by its given name (for example, jpeek).

To find the parent of any directory, you can always open .. (dot dot). The filesystem's tree structure, tied together with .. and . entries, is built automatically each time you create a new directory.

One directory in the filesystem is special because it's its own parent. That's the root directory. Let's list it now (for clarity, we'll omit a lot of the entries):

```
$ # Listing III (root directory)
$ ls -ai ..
 2 . 2 .. 58 home
$ cd ..
$ ls -ai
 2 . 2 .. 58 home
$ pwd
/
```

So, if you're currently in the root directory and you type cd .., you'll stay where you are.

I-number Confusion

Each filesystem has its own set of index numbers. So, as you're looking around with ls -ai, remember that there may be more than one link with, say, number 5423. You'll also find inconsistencies where filesystems are mounted. For example, at the root directory of a system where /home is mounted on root:

Theres a Lot in the Dot: Filesystem Permissions and Pathnames

Jerry Peek

```
$ pwd
/  
$ ls -la  
2 . 2 .. 24577 home  
$ ls -la home  
123 . 123 ..
```

Of course, listing a bunch of directory entries doesn't do you much good — except to see what's in the filesystem. If you were searching for filesystem corruption, you could check these things yourself. (That's one of the things done by `fsck`, the filesystem consistency checking program.) What's more useful is to use this knowledge so you can specify filesystem objects that you want to access. To do that, you use pathnames and, sometimes, the current directory. The next column covers pathnames in detail.

Let's wrap up by looking at a diagram of a simplified filesystem, Figure One. [Click for a larger view.](#) (Opening in a separate window or tab may be easier.)

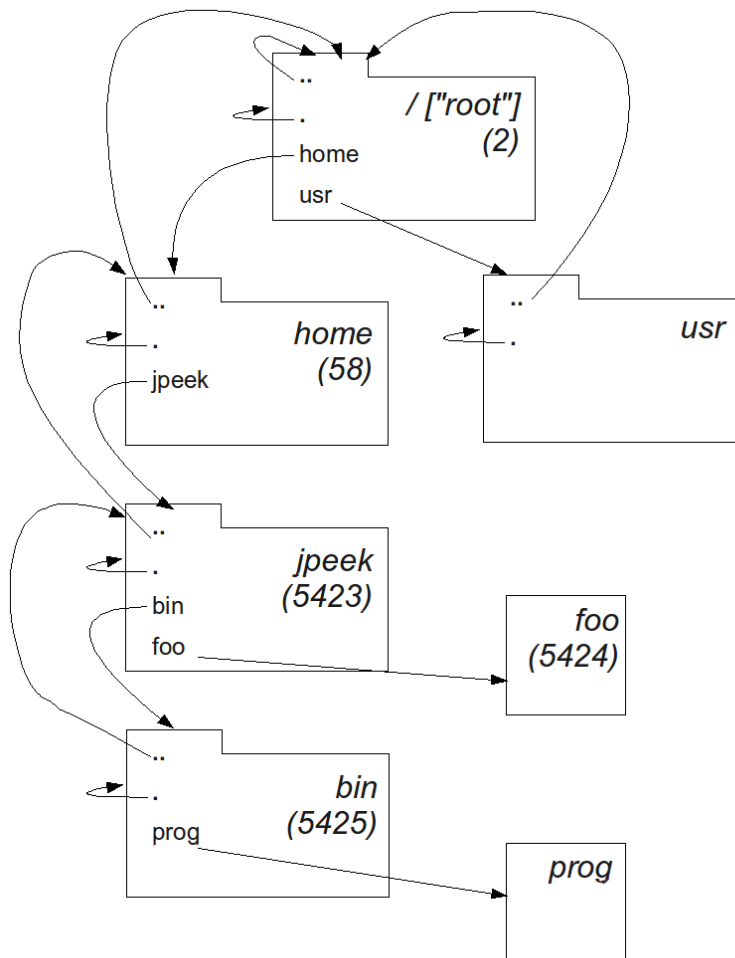


Figure 1: Simplified Filesystem Diagram

Theres a Lot in the Dot: Filesystem Permissions and Pathnames

Jerry Peek

To make things simpler, the diagram omits details such as the system of nodes and indirection that handles metadata (like file access permissions and link counts) and the pointers to the actual disk blocks. This diagram is to help you follow where the links point.

For instance, in any directory, opening `.` (dot) opens the directory itself; that's what's meant by the arrow starting at `.` pointing to the directory. As another example, you can see that the entry named `jpeek` from directory 58 points to directory 5423, as does the entry named `.` in directory 5423 and the `..` entry in directory 5425. All three of those entries refer to the same directory.

If this seems even a bit fuzzy, please look through the diagram and open a terminal window to try things on your own filesystem. Then you'll be ready for the part two on pathnames and permissions tomorrow.

In the previous article we saw how the hidden directory entries named `.` (dot) and `..` (dot dot) tie the filesystem together. Those names are hard links that reference the actual filesystem object through the index number. A directory always has at least two names: `.` and its given name. You can always reach the parent directory through the `..` entry.

Now let's dig into how pathnames and permissions work internally. (If you're familiar with all of this, try the quiz at the end.)

Two Paths to the Same Place

Pathnames can confuse users, but they're actually simple when you see how they work. A pathname gives the location of an object (a file, a directory, a socket, etc.) in the filesystem. There are two kinds of pathname: absolute (or full) and relative:

- An absolute pathname starts at the root directory (the top of the filesystem). It always starts with a `/` (slash). Example: `/home/jpeek/foo`
- A relative pathname starts at the current directory. It never starts with a slash. Example: if your current directory is `/home/jpeek`, two relative pathnames you might type are: `foo` and `../someuser`

No matter what your current directory is, you can always find an object through its absolute pathname. But a relative pathname is often shorter.

Following a Path

When you give a pathname to a program, how does it find the object you specified? For an absolute pathname, it reads the root directory and follows the path from there. Otherwise, the program opens the current directory and follows the path from there.

Figure 1 shows how the shell's system calls find a directory after you type `cd /home/jpeek`. This figure comes from part of the filesystem tree in the previous article.

Theres a Lot in the Dot: Filesystem Permissions and Pathnames

Jerry Peek

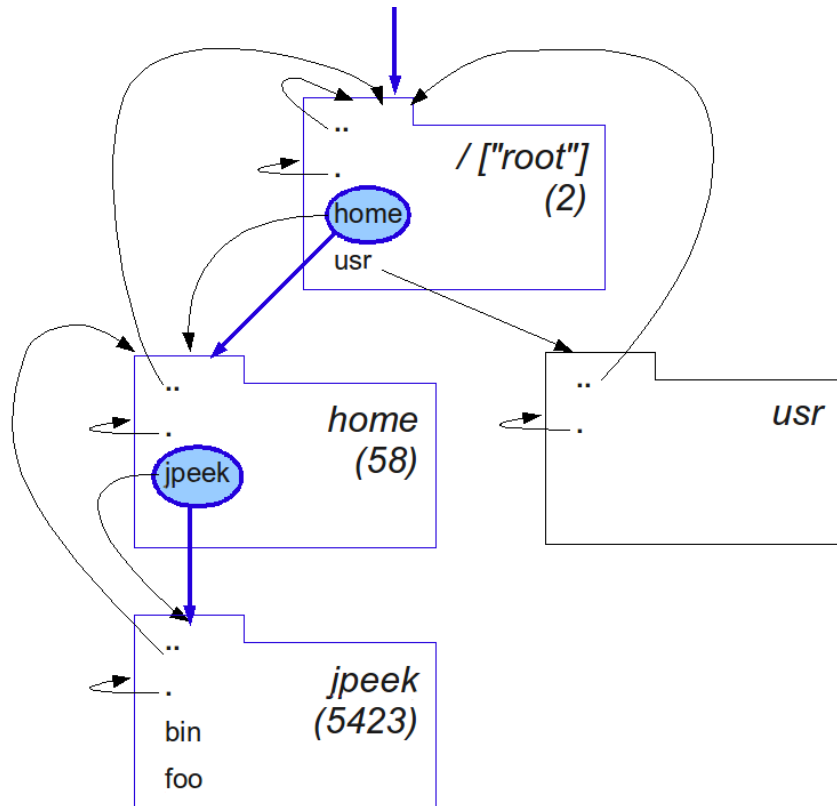


Figure 1: Finding /home/jpeek

1. The pathname starts with /, so the system opens the root directory.
2. Inside the root directory, it looks for a directory entry named home. If there's no home entry, the pathname is invalid. Otherwise, the system opens the home directory.
3. Inside the /home directory, it looks for a directory entry named jpeek.

Here are some more examples: multiple ways to reach the same directory. I'll start by changing the current directory to my home directory. (A simple cd with no pathname defaults to a user's home directory.)

```
1$ cd
2$ ls -ai
5423 . 58 .. 5425 bin 5424 foo
3$ ls -ai .
5423 . 58 .. 5425 bin 5424 foo
4$ ls -ai ../../.
5423 . 58 .. 5425 bin 5424 foo
5$ ls -ai ../jpeek
5423 . 58 .. 5425 bin 5424 foo
6$ ls -ai /home/jpeek
5423 . 58 .. 5425 bin 5424 foo
7$ ls -ai /home/jpeek/.
5423 . 58 .. 5425 bin 5424 foo
```

Theres a Lot in the Dot: Filesystem Permissions and Pathnames

Jerry Peek

```
8$ ls -ai /home/jpeek/../../jpeek
5423 .   58 ..   5425 bin  5424 foo
9$ ls -ai /jpeek
ls: cannot access /jpeek: No such file or directory
```

Every ls command lists the same directory, some through relative pathnames and some through absolute. Command 4 opens ., then opens ., then opens . again: always the same directory. Command 8 opens /home/jpeek, then the parent directory, then the jpeek directory from there — with the same result.

Why did command 9 fail? Trace it through: open the root directory, then look for an entry named jpeek. There isn't one.

Access Permissions

Once you see how pathnames work, understanding access permissions is simple. You're probably familiar with access permissions for files: read permission lets a program read the file's contents, write permission allows modification, and execute permission is for executable programs. Notice that none of those permissions control whether you can rename or delete the file. Why? We'll see soon.

A directory is actually just a special type of file, and a directory's permissions are easy to understand when you realize that a directory holds entries for other files. So:

- Read permission lets a program read the entries listed in the directory — for instance, with ls or a shell wildcard like *. (Some graphical filesystem browsers will say a directory is “unreadable” unless it also has execute permission, but that's not strictly true.)
- Write permission lets you modify the entries in a directory. Think of the directory as a file in a word processor. To edit the file — to change the spelling or words, add or delete words — you need write permission. A directory is the same way. If you don't have write permission, you can't delete, rename, or add entries to the directory. So, to protect the contents of a directory, make the directory unwritable.
- Execute permission lets you access the files linked from a directory. Note that you don't need read permission; if a directory isn't readable, you can still access an entry if you know its exact name. (That's like denying index permission on a web server folder to prevent people from knowing its contents: you can still get a web page by using its exact URI.)

Pathnames and the shell PATH

Pathnames work almost anywhere, with any command. One notable exception is specifying the program you want to use. If the program's pathname contains a /, it will be followed. Otherwise, different rules apply.

For example, let's say your current directory is /home/jpeek/bin. If you want to run the program prog from the current directory, here are three pathnames — three ways you can specify its location:

```
$ ls -l prog
-rwxr-xr-x 1 jpeek 238 2009-08-19 06:28 prog
$ /home/jpeek/bin/prog
$ ./prog
$ prog
bash: prog: command not found
$ echo $PATH
```

Theres a Lot in the Dot: Filesystem Permissions and Pathnames

Jerry Peek

```
/usr/local/bin:/usr/bin:/bin
```

From `ls -l`, we can see that `prog` is a valid pathname: it's an entry in the current directory. So why did `bash` say "command not found"? If you specify a program name only, without a slash, the shell will search through each directory in the `PATH` environment variable, looking for the program in that directory. As you can see from the last command, the current directory isn't in `$PATH`. This is for security. In this case, to run a program from the current directory, you have to tell the shell `./prog`. But, with other commands, the `./` is redundant; `prog` is simpler.

Quiz

Here's a directory listing from `ls -l`, run by the superuser (who can access everything on the filesystem):

```
# ls -la /home/zoe
drwxrwx--x 2 zoe users 8192 2010-03-14 18:33 .
drwx--x--x 9 zoe users 16384 2010-03-11 08:00 ..
-rw-r--r-- 1 zoe users 2942 2010-03-16 13:45 afile
drwx----- 2 zoe users 4096 2010-03-14 18:33 dir
-rw-r--r-- 1 zoe users 4039 2009-11-22 08:18 .profile
```

- Q: An attacker breaks into a non-superuser account that's not a member of the `users` group. What does `ls /home` show?

A: Permission denied because there's no read permission on `..` (`/home`, the parent of `/home/zoe`).

- Q: The attacker knows that there's a user named `zoe`. What does `cd /home/zoe` do?

A: It succeeds because the parent directories both have execute permission. But listing would fail with `ls`: cannot open directory `.` because there's no read permission.

- Q: The attacker tries `vim .profile` to modify `Zoe's` startup file. What happens?

A: `vim` opens the file read-only because there's read permission, but no write permission, on the file.

- Q: `jpeek`, who's a member of the `users` group, types `cd /home/zoe`, then `ls`, then `rm afile`. What happens?

A: He can access the directory because he has execute permission. He can list `.` because he has read permission. He removes `afile` because he has write permission on `.` (the directory containing the entry for `afile`).

- Q: What if `jpeek` now types `cd dir`?

A: It fails because he doesn't have execute permission on `dir`.

If anything in that quiz surprised you, this might be a good time to open a terminal window and make up some exercises for yourself. Enjoy!