NAME

path_resolution − how a pathname is resolved to a file

DESCRIPTION

Some UNIX/Linux system calls have as parameter one or more filenames. A filename (or pathname) is resolved as follows.

Step 1: start of the resolution process

If the pathname starts with the '/' character, the starting lookup directory is the root directory of the calling process. A process inherits its root directory from its parent. Usually this will be the root directory of the file hierarchy. A process may get a different root directory by use of the **chroot**(2) system call, or may temporarily use a different root directory by using **openat2**(2) with the **RESOLVE_IN_ROOT** flag set.

A process may get an entirely private mount namespace in case it—or one of its ancestors—was started by an invocation of the **clone**(2) system call that had the **CLONE_NEWNS** flag set. This handles the '/' part of the pathname.

If the pathname does not start with the '/' character, the starting lookup directory of the resolution process is the current working directory of the process — or in the case of **openat**(2)-style system calls, the *dfd* argument (or the current working directory if **AT_FDCWD** is passed as the *dfd* argument). The current working directory is inherited from the parent, and can be changed by use of the **chdir**(2) system call.

Pathnames starting with a '/' character are called absolute pathnames. Pathnames not starting with a '/' are called relative pathnames.

Step 2: walk along the path

Set the current lookup directory to the starting lookup directory. Now, for each nonfinal component of the pathname, where a component is a substring delimited by '/' characters, this component is looked up in the current lookup directory.

If the process does not have search permission on the current lookup directory, an **EACCES** error is returned ("Permission denied").

If the component is not found, an **ENOENT** error is returned ("No such file or directory").

If the component is found, but is neither a directory nor a symbolic link, an **ENOTDIR** error is returned ("Not a directory").

If the component is found and is a directory, we set the current lookup directory to that directory, and go to the next component.

If the component is found and is a symbolic link, we first resolve this symbolic link (with the current lookup directory as starting lookup directory). Upon error, that error is returned. If the result is not a directory, an **ENOTDIR** error is returned. If the resolution of the symbolic link is successful and returns a directory, we set the current lookup directory to that directory, and go to the next component. Note that the resolution process here can involve recursion if the prefix ('dirname') component of a pathname contains a filename that is a symbolic link that resolves to a directory (where the prefix component of that directory may contain a symbolic link, and so on). In order to protect the kernel against stack overflow, and also to protect against denial of service, there are limits on the maximum recursion depth, and on the maximum number of symbolic links followed. An**ELOOP** error is returned when the maximum is e xceeded ("Too many levels of symbolic links").

As currently implemented on Linux, the maximum number of symbolic links that will be followed while resolving a pathname is 40. Before Linux 2.6.18, the limit on the recursion depth was 5. Starting with Linux 2.6.18, this limit was raised to 8. In Linux 4.2, the kernel's pathname-resolution code was reworked to eliminate the use of recursion, so that the only limit that remains is the maximum of 40 resolutions for the entire pathname.

The resolution of symbolic links during this stage can be blocked by using **openat2**(2), with the **RESOLVE_NO_SYMLINKS** flag set.

Step 3: find the final entry

The lookup of the final component of the pathname goes just like that of all other components, as described in the previous step, with two differences: (i) the final component need not be a directory (at least as far as the path resolution process is concerned—it may have to be a directory, or a nondirectory, because of the requirements of the specific system call), and (ii) it is not necessarily an error if the component is not found—maybe we are just creating it. The details on the treatment of the final entry are described in the manual pages of the specific system calls.

. and ..

By convention, every directory has the entries "." and "..", which refer to the directory itself and to its parent directory, respectively.

The path resolution process will assume that these entries have their conventional meanings, regardless of whether they are actually present in the physical filesystem.

One cannot walk up past the root: "/.." is the same as "/".

Mount points

After a *mount dev path* command, the pathname "path" refers to the root of the filesystem hierarchy on the device "dev", and no longer to whatever it referred to earlier.

One can walk out of a mounted filesystem: "path/.." refers to the parent directory of "path", outside of the filesystem hierarchy on "dev".

Traversal of mount points can be blocked by using **openat2**(2), with the **RESOLVE_NO_XDEV** flag set (though note that this also restricts bind mount traversal).

Trailing slashes

If a pathname ends in a '/', that forces resolution of the preceding component as in Step 2: the component preceding the slash either exists and resolves to a directory or it names a directory that is to be created immediately after the pathname is resolved. Otherwise, a trailing '/' is ignored.

Final symbolic link

If the last component of a pathname is a symbolic link, then it depends on the system call whether the file referred to will be the symbolic link or the result of path resolution on its contents. For example, the system call **lstat**(2) will operate on the symbolic link, while **stat**(2) operates on the file pointed to by the symbolic link.

Length limit

There is a maximum length for pathnames. If the pathname (or some intermediate pathname obtained while resolving symbolic links) is too long, an **ENAMETOOLONG** error is returned ("Filename too long").

Empty pathname

In the original UNIX, the empty pathname referred to the current directory. Nowadays POSIX decrees that an empty pathname must not be resolved successfully. Linux returns**ENOENT** in this case.

Permissions

The permission bits of a file consist of three groups of three bits; see **chmod**(1) and **stat**(2). The first group of three is used when the effective user ID of the calling process equals the owner ID of the file. The second group of three is used when the group ID of the file either equals the effective group ID of the calling process, or is one of the supplementary group IDs of the calling process (as set by **setgroups**(2)). When neither holds, the third group is used.

Of the three bits used, the first bit determines read permission, the second write permission, and the last execute permission in case of ordinary files, or search permission in case of directories.

Linux uses the fsuid instead of the effective user ID in permission checks. Ordinarily the fsuid will equal the effective user ID, but the fsuid can be changed by the system call **setfsuid**(2).

(Here "fsuid" stands for something like "filesystem user ID". The concept was required for the implementation of a user space NFS server at a time when processes could send a signal to a process with the same effective user ID. It is obsolete now. Nobody should use**setfsuid**(2).)

Similarly, Linux uses the fsgid ("filesystem group ID") instead of the effective group ID. See **setfsgid**(2).

Bypassing permission checks: superuser and capabilities

On a traditional UNIX system, the superuser (*root*, user ID 0) is all-powerful, and bypasses all permissions restrictions when accessing files.

On Linux, superuser privileges are divided into capabilities (see **capabilities**(7)). Two capabilities are relevant for file permissions checks: **CAP_DAC_OVERRIDE** and **CAP_DAC_READ_SEARCH**. (A process has these capabilities if its fsuid is 0.)

The **CAP_DAC_OVERRIDE** capability overrides all permission checking, but grants execute permission only when at least one of the file's three execute permission bits is set.

The **CAP_DAC_READ_SEARCH** capability grants read and search permission on directories, and read permission on ordinary files.

SEE ALSO

readlink(2), **capabilities**(7), **credentials**(7), **symlink**(7)