Naming in file systems

Required reading: nami(), and all other file system code.

Overview

To help users to remember where they stored their data, most systems allow users to assign their own names to their data. Typically the data is organized in files and users assign names to files. To deal with many files, users can organize their files in directories, in a hierarchical manner. Each name is a pathname, with the components separated by "/".

To avoid that users have to type long abolute names (i.e., names starting with "/" in Unix), users can change their working directory and use relative names (i.e., naming that don't start with "/").

User file namespace operations include create, mkdir, mv, ln (link), unlink, and chdir. (How is "mv a b" implemented in xv6? Answer: "link a b"; "unlink a".) To be able to name the current directory and the parent directory every directory includes two entries "." and "..". Files and directories can reclaimed if users cannot name it anymore (i.e., after the last unlink).

Recall from last lecture, all directories entries contain a name, followed by an inode number. The inode number names an inode of the file system. How can we merge file systems from different disks into a single name space?

A user grafts new file systems on a name space using mount. Umount removes a file system from the name space. (In DOS, a file system is named by its device letter.) Mount takes the root inode of the to-be-mounted file system and grafts it on the inode of the name space entry where the file system is mounted (e.g., /mnt/disk1). The in-memory inode of /mnt/disk1 records the major and minor number of the file system mounted on it. When namei sees an inode on which a file system is mounted, it looks up the root inode of the mounted file system, and proceeds with that inode.

Mount is not a durable operation; it doesn't surive power failures. After a power failure, the system administrator must remount the file system (i.e., often in a startup script that is run from init).

Links are convenient, because with users can create synonyms for file names. But, it creates the potential of introducing cycles in the naning tree. For example, consider link ("a/b/c", "a"). This makes c a synonym for a. This cycle can complicate matters; for example:

- If a user subsequently calls unlink ("a"), then the user cannot name the directory "b" and the link "c" anymore, but how can the file system decide that?

This problem can be solved by detecting cycles. The second problem can be solved by computing with files are reacheable from "/" and reclaim all the ones that aren't reacheable. Unix takes a simpler approach: avoid cycles by disallowing users to create links for directories. If there are no cycles, then reference counts can be used to see if a file is still referenced. In the inode maintain a field for counting references (nlink in xv6's dinode). link increases the reference count, and unlink decreases the count; if the count reaches zero the inode and disk blocks can be reclaimed.

How to handle symbolic links across file systems (i.e., from one mounted file system to another)? Since inodes are not unique across file systems, we cannot create a link across file systems; the directory entry only contains an inode number, not the inode number and the name of the disk on which the inode is located. To handle this case, Unix provides a second type of link, which are called soft links.

Soft links are a special file type (e.g., T_SYMLINK). If namei encounters a inode of type T_SYMLINK, it resolves the the name in the symlink file to an inode, and continues from there. With symlinks one can create cycles and they can point to non-existing files.

The design of the name system can have security implications. For example, if you tests if a name exists, and then use the name, between testing and using it an adversary can have change the binding from name to object. Such problems are called TOCTTOU.

An example of TOCTTOU is follows. Let's say root runs a script every night to remove file in /tmp. This gets rid off the files that editors might left behind, but we will never be used again. An adversary can exploit this script as follows:

```
Root                            Attacker
                                mkdir ("/tmp/etc")
                          creat ("/tmp/etc/passw")
readdir ("tmp");
lstat ("tmp/etc");
readdir ("tmp/etc");
                                rename ("tmp/etc", "/tmp/x");
                          symlink ("etc", "/tmp/etc");
unlink ("tmp/etc/passwd");
```

Lstat checks whether /tmp/etc is not symbolic link, but by the time it runs unlink the attacker had time to creat a symbolic link in the place of /tmp/etc, with a password file of the adversary's choice.

This problem could have been avoided if every user or process group had its own private /tmp, or if access to the shared one was mediated.

V6 code examples

namei (sheet 46) is the core of the Unix naming system. namei can be called in several ways: NAMEI_LOOKUP (resolve a name to an inode and lock inode), NAMEI_CREATE (resolve a name, but lock parent inode), and NAMEI_DELETE (resolve a name, lock parent inode, and return offset in the directory). The reason is that namei is complicated is that we want to atomically test if a name exist and remove/create it, if it does; otherwise, two concurrent processes could interfere with each other and directory could end up in an inconsistent state.

Let's trace open("a", O_RDWR), focussing on namei:

- 5263: we will look at creating a file in a bit.
- 5277: call namei with NAMEI_LOOKUP
- 4629: if path name start with "/", lookup root inode (1).
- 4632: otherwise, use inode for current working directory.
- 4638: consume row of "/", for example in "/////a////b"
- 4641: if we are done with NAMEI_LOOKUP, return inode (e.g., namei("/")).
- 4652: if the inode we are searching for a name isn't of type directory, give up.
- 4657-4661: determine length of the current component of the pathname we are resolving.
- 4663-4681: scan the directory for the component.
- 4682-4696: the entry wasn't found. if we are the end of the pathname and NAMEI_CREATE is set, lock parent directory and return a pointer to the start of the component. In all other case, unlock inode of directory, and return 0.
- 4701: if NAMEI_DELETE is set, return locked parent inode and the offset of the to-be-deleted component in the directory.
- 4707: lookup inode of the component, and go to the top of the loop.

Now let's look at creating a file in a directory:

- 5264: if the last component doesn't exist, but first part of the pathname resolved to a directory, then dp will be 0, last will point to the beginning of the last component, and ip will be the locked parent directory.
- 5266: create an entry for last in the directory.
- 4772: mknod1 allocates a new named inode and adds it to an existing directory.
- 4776: ialloc. skan inode block, find unused entry, and write it. (if lucky 1 read and 1 write.)
- 4784: fill out the inode entry, and write it. (another write)
- 4786: write the entry into the directory (if lucky, 1 write)

Why must the parent directory be locked? If two processes try to create the same name in the same directory, only one should succeed and the other one, should receive an error (file exist).

Link, unlink, chdir, mount, umount could have taken file descriptors instead of their path argument. In fact, this would get rid of some possible race conditions (some of which have security implications, TOCTTOU). However, this would require that the current working directory be remembered by the process, and UNIX didn't have good ways of maintaining static state shared among all processes belonging to a given user. The easiest way is to create shared state is to place it in the kernel.

We have one piece of code in xv6 that we haven't studied: exec. With all the ground work we have done this code can be easily understood (see sheet 54).