Naming in File Systems

COMP 321

Dave Johnson

1

What is Naming in File Systems?

A "directory" maps file names to files

Two alternatives for storing file names

- The file name is ***in*** each inode

- The file name is ***separate from and linked to*** the inode

This second approach easily allows multiple names for the same file

2

Other Naming Implementation Choices

How is the space for a directory allocated and managed?

- Could be a preallocated contiguous chunk of disk blocks
- Or could be managed the same as the space for any other file
  - A regular file contains bytes of data
  - In the same way, a directory file contains bytes of the directory entries

What is the maximum length of a file name?

- A fixed maximum length
  - Example: Classical Unix format used fixed 14-character area for the name
- Or a variable length
  - Makes managing the space allocation for the names more difficult

Other Naming Implementation Choices

What is the format of a file name?

- Fixed format
  - Example original MS-DOS file system: Maximum of 8 characters "." maximum of 3 characters, and for letters, only uppercase allowed
- Arbitrary string of any (or almost any) characters
  - Example Unix: Literally any character other than `'\0'` and `'/'`

What is the structure and organization of a directory?

- An unsorted list of entries, or a sorted list
- A hash table (on disk, of course)
- Something else?

Example: MS-DOS Directory Entry (Like An Inode)

```
struct direntry
{
        char  name[8];              /* file name, before the "." */
        char  extension[3];         /* file name extension, after the "." */
        char  attributes;           /* protection and other attributes */
        char  unused[10];           /* unused bytes in directory entry */
        char  time[2];              /* last modified time */
        char  date[2];              /* last modified date */
        unsigned short  first;      /* first block number of file's data */
        unsigned int  size;         /* file's size in bytes*/
};
```


Example: Classical Unix Directory Entry

```
#define DIRSIZ          14

struct  direct
{
        ino_t   d_ino;              /* inode number of the file itself */
        char    d_name[DIRSIZ];     /* this name for the file */
};
```

Note that the name is <u>not</u> null-terminated (`'\0'`) in d_name

- The name ends at the first null byte `'\0'` in d_name
- ***<u>Or</u>*** after DIRSIZ number of characters
- Whichever occurs ***first*** (a 14-character name has no `'\0'`)
- In MS-DOS, the name and extension are also not null-terminated

Example: Classical Unix Inode

```
struct dinode
{
        unsigned short  di_mode;   /* mode and type of file */
        short     di_nlink;        /* number of links to file */
        short     di_uid;          /* owner's user id */
        short     di_gid;          /* owner's group id */
        off_t     di_size;         /* number of bytes in file */
        char      di_addr[40];     /* disk block addresses */
        time_t    di_atime;        /* time last accessed */
        time_t    di_mtime;        /* time last modified */
        time_t    di_ctime;        /* time created */
};
```

di_addr bytes: 39 bytes used; 13 addresses, 3 bytes each

Adding Subdirectories: A Tree-Structured Directory

There needs to be a designated "root" directory

- Could be a preallocated contiguous chunk of disk blocks
  - But how do you know where that preallocated chunk of disk blocks is?
- Or could be managed the same as any other file
  - But how do you find that file, since it's name must be interpreted relative to the root directory?

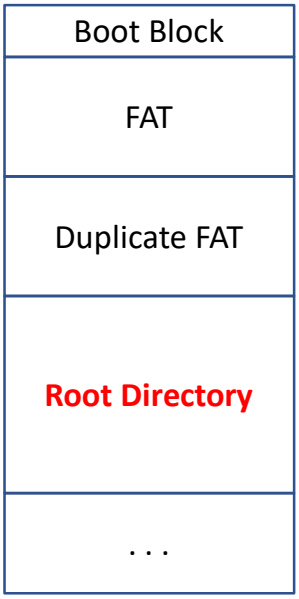
We need to be able to link subdirectories together in a tree

- This part is easy: Just need to be able to mark the "type" of a file as either a "regular" type file or a directory type file

Example: MS-DOS Root Directory

A preallocated contiguous chunk of disk blocks

- The boot block contains code to boot the kernel
- It also describes the size of the FAT and the number (1 or 2) of FATs
- It also describes the size of the root directory
- The root directory immediate follows the last FAT
- The root directory must then be a single contiguous chunk of disk space



Example: MS-DOS Linking Subdirectories Together

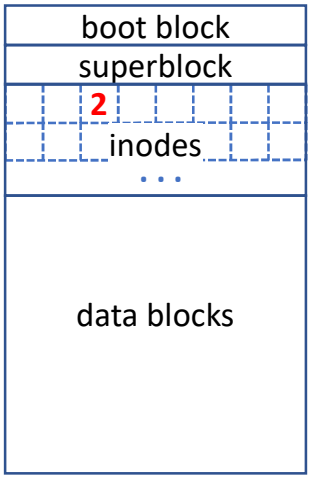
```
struct direntry
{
        char  name[8];              /* file name, before the "." */
        char  extension[3];         /* file name extension, after the "." */
        char  attributes;           /* protection and other attributes */
        char  unused[10];           /* unused bytes in directory entry */
        char  time[2];              /* last modified time */
        char  date[2];              /* last modified date */
        unsigned short  first;      /* first block number of file's data */
        unsigned int  size;         /* file's size in bytes*/
};
```

Attribute bit 0x10 means file is a directory (its data is subdirectory entries)

Example: Unix Root Directory

Managed the same as any other file

- That file is always described by inode number 2
- A fixed, constant, well-known inode number
- The directory entries are then stored in the same way as the data for any other file
  - In the data blocks "hanging off of" inode 2
- Why inode number ***2***?
  - Inode number 0 is not used, so that a 0 in an inode number field can be a special value
  - Historically, inode number 1 was used for a "file" containing the bad blocks on the disk

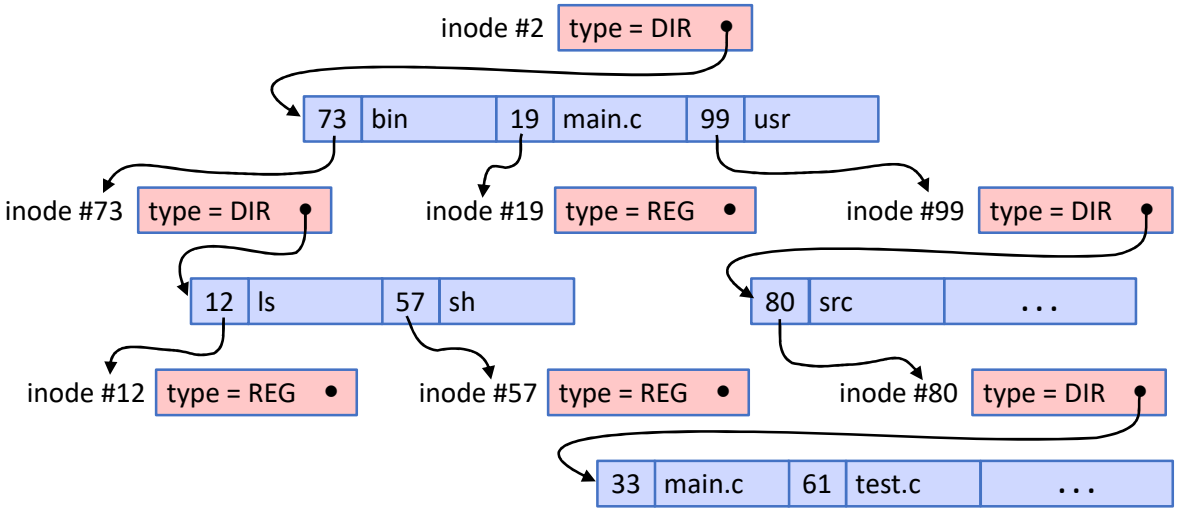


Example: Unix Linking Subdirectories Together

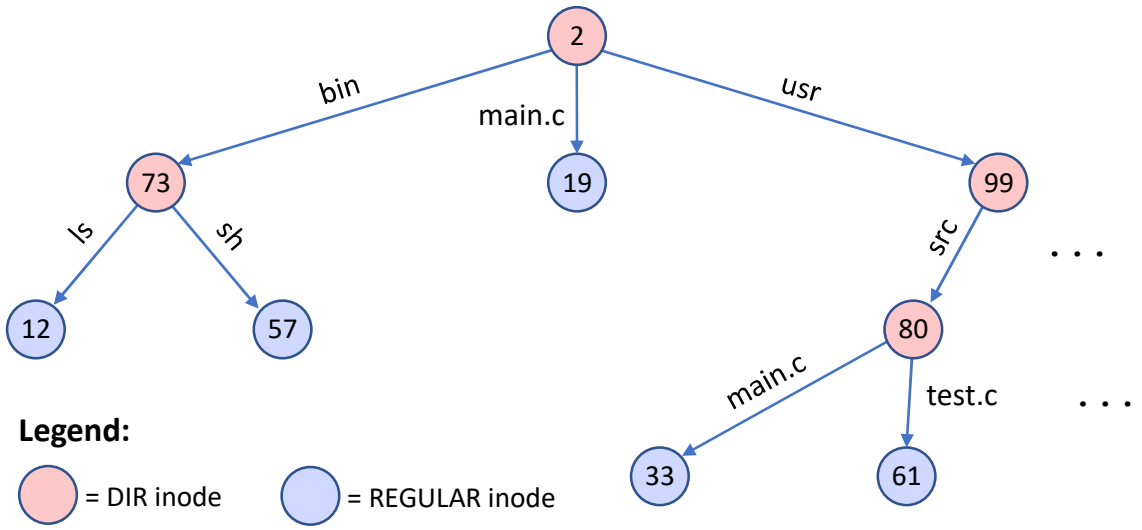
```
struct dinode
{
        unsigned short  di_mode;  /* mode and type of file */
        short     di_nlink;       /* number of links to file */
        short     di_uid;         /* owner's user id */
        short     di_gid;         /* owner's group id */
        off_t     di_size;        /* number of bytes in file */
        char      di_addr[40];    /* disk block addresses */
        time_t    di_atime;       /* time last accessed */
        time_t    di_mtime;       /* time last modified */
        time_t    di_ctime;       /* time created */
};
```

Mode bit 040000 means file is a directory (its data is subdirectory entries)

An Example Unix Directory Tree



An Example Unix Directory Tree



Shorthand for Pathnames

Each process has a current directory

- The inode number of a process's current directory is remembered in PCB
- Set to "home" directory at login (by the "login" command before exec shell)
- Changed by chdir( ) file system operation (called by "cd" command)
- Looking up a pathname:
  - If pathname[0] == '/', this is an ***absolute*** pathname
    - Start lookup at inode number 2 and the directory entries hanging off it
  - Otherwise, this is a ***relative*** pathname
    - Start lookup at inode number remembered in process's PCB and the directory entries hanging off it it
- Saves typing, but also improves efficiency since fewer steps in a relative lookup

Shorthand for Pathnames

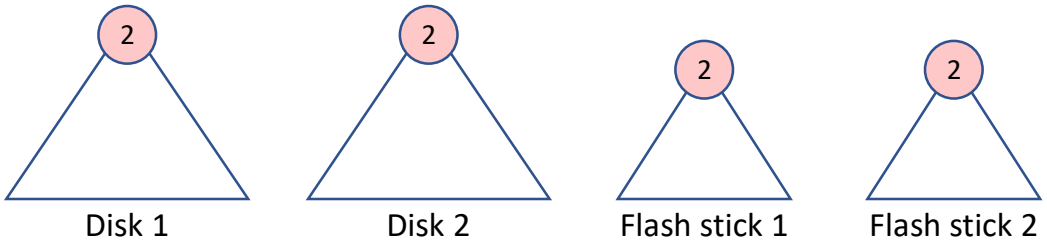
The "." and ".." names

- Doing things like "`./myprogram`" or "`cd ../otherdir`" are ***not*** special cases
- The ***first two*** entries in ***every*** directory have the names "." and ".."
  - Inode number in "." entry is the directory's own inode number
  - Inode number in ".." entry is the directory's parent's inode number
  - In the root directory (inode 2), the inode number for ".." entry is also 2
- The "." and ".." entries are always created as part of creating the directory itself and cannot be removed
  - The "." and ".." entries are ***really*** there, on disk, in ***every*** directory

Mounting File Systems

What if you have more than one disk?

- Examples: 2 separate hard disks, removable flash memory sticks
- Impossible to manage inode numbers to be unique between all of these!
- Instead, ***use the same ranges of inode numbers on each***, starting at ***inode number 2*** as the root of each
- The first file system is mounted automatically at boot as the ***root file system***

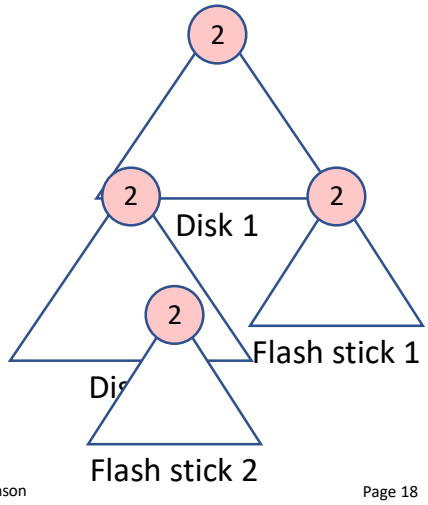


Mounting File Systems

Combine separate file systems into what appears as a single file system tree

- mount(file_system_device_name, existing_directory_name, options)
- Adds an entry to the file system ***mount table*** in the kernel
  - Such that a reference to the inode number of that existing directory name
  - … will be turned into a reference to inode 2 on that mounted device

For current directory, remember the mounted device id as well as the inode number in PCB



Background: Device IDs

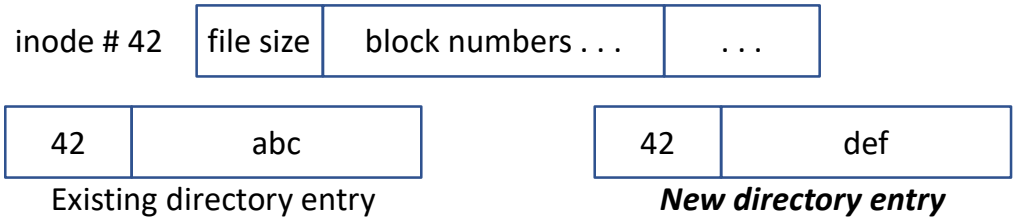
A device id is of the form (major device id, minor device id)

- Major device id indicates the type of the device
  - Indicates which device driver to use for operations on that device
  - An index into an array of structs
  - Each struct contains pointers to the function for each kind of operation for that type of device
- Minor device id indicates which device of that (major) device type
  - Generally serves as an index into an array of structs
  - Each struct contains the variables for that instance of this type of device
- The (major, minor) combination is unique among all devices
  - So device id and inode number is unique among all inodes

Creating a New "Link" to an Existing File

Creates a new directory entry with the existing file's inode number in it:

- link(oldname, newname) Example: link("abc", "def")



- Inode ***nlink*** field keeps a count of these links to that same inode
  - nlink = count of directory entries with that inode number in them
  - nlink ***incudes*** the "." and ".." directory entries (but I will ignore those for clarity in the following slides here)

The Corresponding "Unlink" Operation to Remove

Removes a name for the file, and may remove the inode and data blocks

- unlink(oldname) Example: unlink("dir1/dir2/dir3/abc")
- Always removes the directory entry for oldname
  - Example: Removes the "abc" entry in directory "dir1/dir2/dir3"
- And ***if then no other way to reach that inode***, frees the inode and the data blocks
  - Inode nlink field is ***incremented*** on Link(oldname, newname)
  - Inode nlink field is ***decremented*** on Unlink(oldname)
  - ***If then the inode nlink field == 0***, then also free that inode and the data blocks

What Does it Mean to "Remove" a Directory Entry?

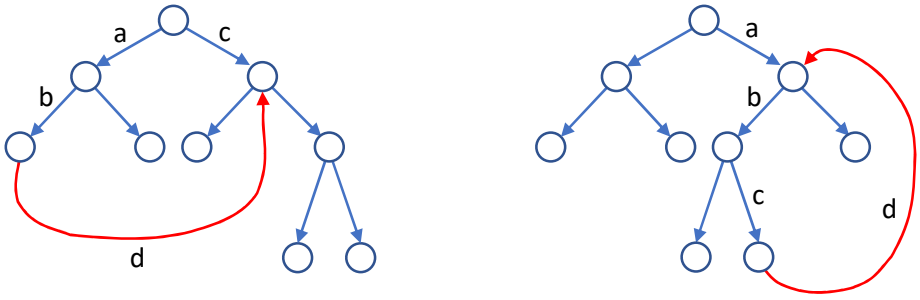
The literal interpretation (inefficient and unnecessary):

- ***Actually*** remove that directory entry
  - Copy ***every*** byte of the directory after this one entry "up" by the size of the directory entry being removed
  - ***Or*** copy the ***last*** entry in the directory on top of the one being removed
- Change the "size" field in the directory's inode to reflect the removed bytes

The better interpretation (sufficient, much more efficient, and commonly used):

- Just change the inode number in that directory entry to 0 (that's all!)
- When reading through a directory, ***ignore*** (skip over) entries with inode number == 0
- ***Do not change the "size" field in the directory's inode since those bytes are still there***

Problems with Links to a Directory

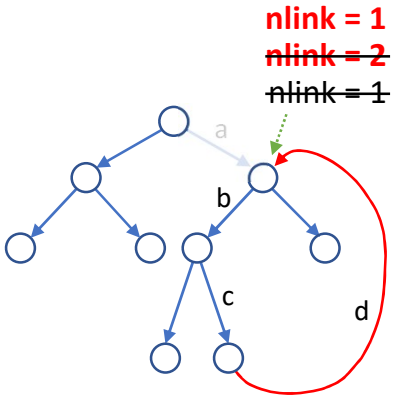


- What should ".." for "/c" be? Is it the "/" inode or the "/a/b" inode?
- Recursive traversal like "ls -R /" could now visit a same subtree more than once
- And a link to a directory could create a ***loop*** in the file system directory "tree"

Problems with Links to a Directory

A big problem with creating such a loop in the file system directory "tree"

- The inode for "/a" nlink value is initially 1
- Creating a new link "d" to this directory increases nlink from 1 to 2
- Then, unlink of "a" changes this nlink value from 2 back down to 1
  - Since nlink did not go to 0, that inode and its data blocks are ***not*** freed!
  - But this entire subtree containing 5 inodes and many data blocks is now inaccessible and lost!



Problems with Links to a Directory

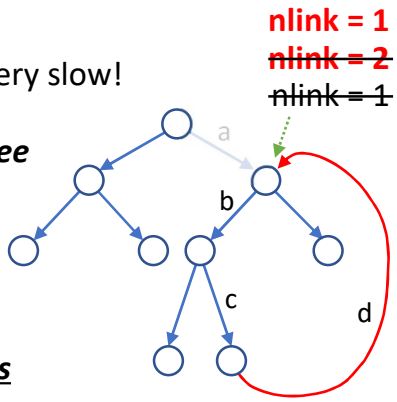
There is no practical way to check if a new directory link creates a loop

- Would require, e.g., traversing the tree to see if you can come back to the same inode
- But the tree lives on the disk, so would be very, very slow!

No practical way to check if unlink detaches subtree

- Would require, e.g., traversing the tree to see if you can still reach all inodes in use
- But the tree lives on the disk, so very, very slow!

Standard solution: <u>Prohibit new links to directories</u>



Symbolic Links

An alternative way of representing a link in the file system:

- Regular links are often called “hard links” to distinguish from symbolic links
- Problems with hard links:
  - Hard links to directories are prohibited due to loop problems
  - Cannot represent a hard link to a file on a different mounted file system
    - The hard link target is just represented as the inode number
    - And inode numbers on each mounted file system are the same!
  - Once created, no way to tell which name is the original name or the “primary” name for the file
    - All links, even the original name, all look identical
- ***Symbolic links do not have these problems***

Creating a Symbolic Link

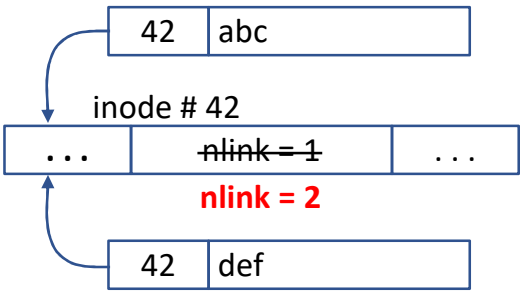
Creates a new directory entry pointing to a <u>new</u> inode:

- symlink(oldname, newname) Example: symlink("/xyz/abc", "def")
- Allocates a ***new*** inode of type = SYMLINK
- Creates a new directory entry "def" with this ***new*** inode number in it
- Data block for new inode contains the character string ***name*** of the link target "/xyz/abc", just as if it was a regular file containing those characters
- ***New*** inode's nlink value is initialized, as usual, to 1
- ***Old existing inode's nlink value is <u>not</u> changed***

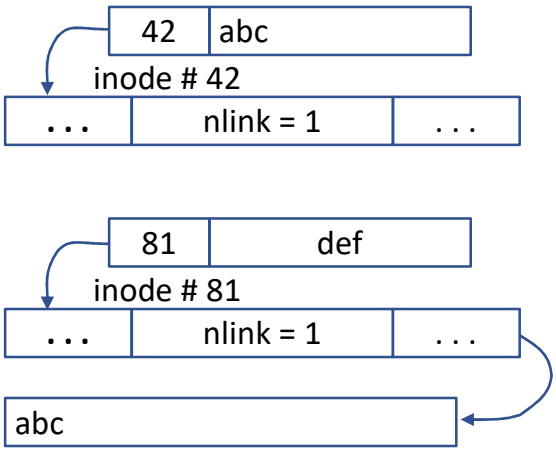
And the <u>oldname</u> does not even need to exist – it is just a character string

Comparing Hard Links vs. Symbolic Links

link("abc", "def")



symlink("abc", "def")



Using a Symbolic Link – for Example, on open()

- Look up the pathname as normal
- If you encounter a symbolic link inode, ***recurse*** to look up that target name
- This ***recursive*** call returns found inode and device id, so resume the original lookup there
  - Just like the original lookup started at (root device id, 2) or at (current directory device id, current directory inode number)
- Example: "/abc/def/ghi", but "def" directory entry points to a symbolic link inode with data "/123/456", so recursively lookup of "/123/456"
  - When that recursive call returns, resume original lookup there
  - Means look up "ghi" starting at that (device id, inode number)

Relative Pathnames as a Symbolic Link Target

- Cannot meaningfully treat this as relative to a process's current directory
  - The current directory may be different for the process that created the symbolic link and any process that uses the symbolic link
  - Would result in a different meaning for the symbolic link, which doesn't make much sense
- Instead, the relative symbolic link target is treated as relative to the directory in which it was found
- Example: "/abc/def/ghi", but "def" directory entry points to a symbolic link to relative pathname "123/456"
  - Recursive look up "123/456" starting at directory where "def" was found
  - Meaning start at the same device id and inode you were already at

What About Loops from Creating Symbolic Links?

Seemingly the same problem as for hard links, but actually <u>not</u> a problem

- When looking up a pathname, you can tell you are traversing a symbolic link rather than a hard link
- Simply place a finite limit on the number of symbolic link traversals in a ***single overall*** pathname lookup
  - Example: Linux limit is 20, FreeBSD Unix limit is 32
  - If limit is exceeded, terminate the lookup and return an error (ELOOP = "Too many levels of symbolic links")
- Such a solution for hard links would limit the maximum height of the entire file system tree, but it does not when applied only to symbolic links traversed
- And you can also decide whether or not to follow symbolic links on, e.g., ls -R