CSCI 2132: Software Development

Files and Directories

Norbert Zeh

Faculty of Computer Science
Dalhousie University

Winter 2019

Files and Directories

Much of the operation of Unix and programs running on Unix can be described as **processes** manipulating **files**.

File = **stream of bytes**

Examples:

- Data stored on disk, CD, Amazon S3, ...
- stdin, stdout, stderr
- Some interfaces to control the Unix kernel are also files

In Unix, a file is an **abstraction** for a **data source** or **data sink**.

Every file must support a certain **interface**.

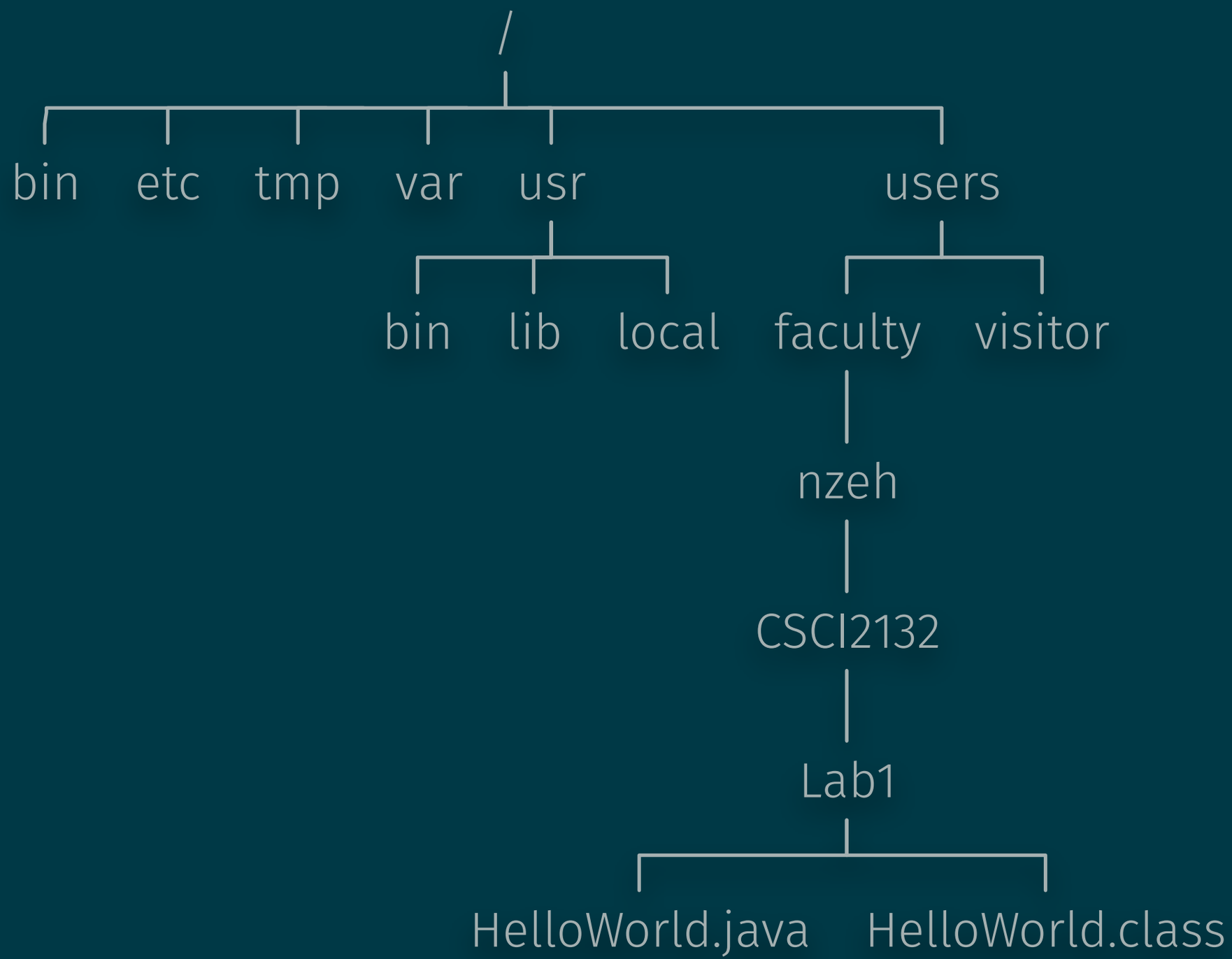
7 Types of Files

- Regular files (–)
- Directories (d)
- Buffered special files (block devices) (b)
- Unbuffered special files (character devices) (c)
- Symbolic links (l)
- Pipes (named pipes) (p)
- Sockets (s)

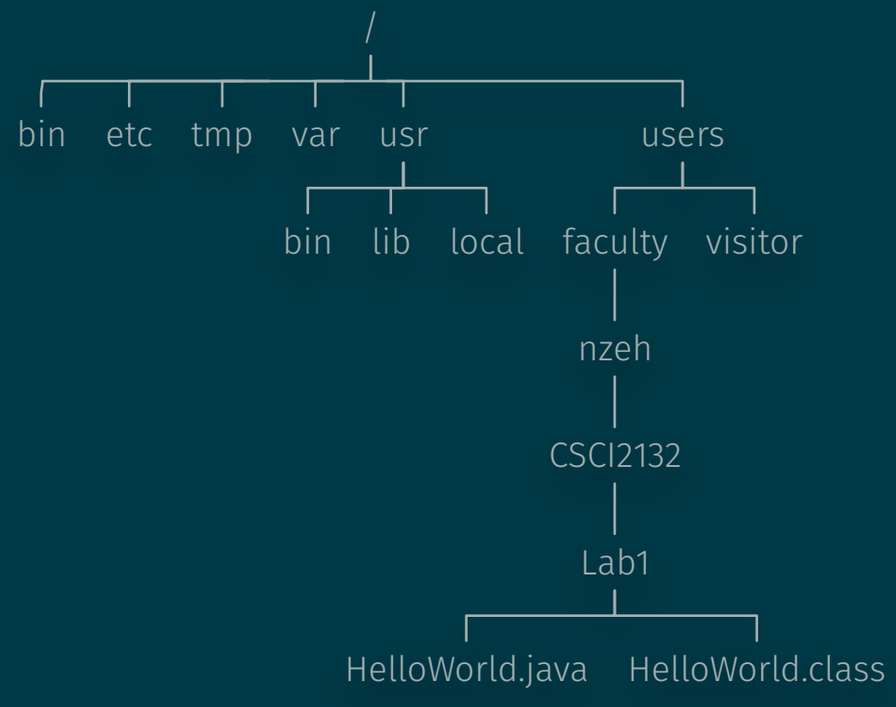
`ls -l` reveals the file type:

```
drwxr-xr-x   14 nzeh  staff      448  3 Dec 09:23 Applications
-rw-r--r--    1 nzeh  staff      695 31 Jul  2017 required-info.txt
```

Navigating the Directory Structure



The Directory Structure

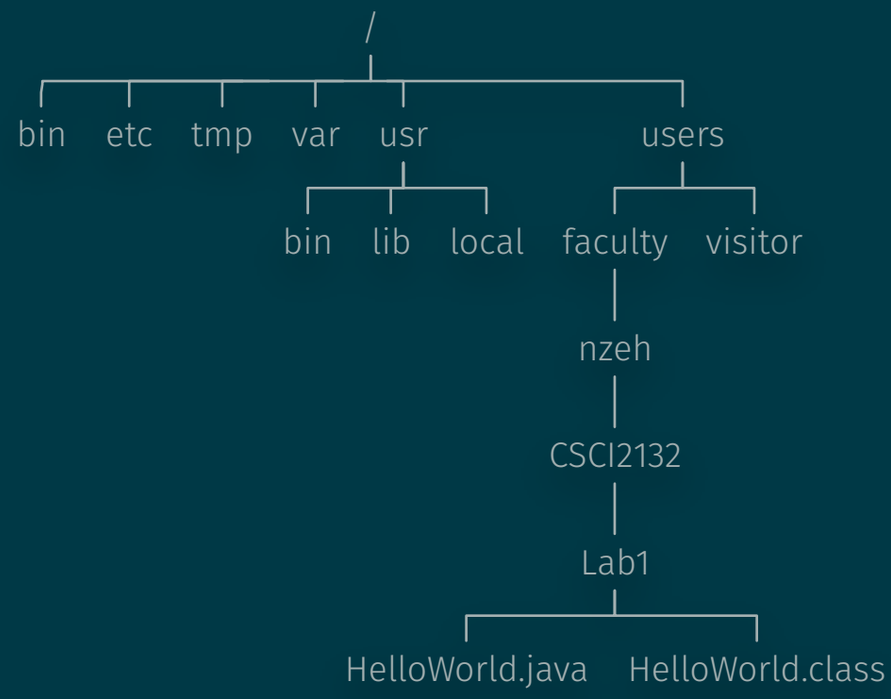


- A **root directory** (/)

- If directory A directly contains directory B, then
  - A is B's **parent directory**,
  - B is a **subdirectory** of A.

- Every directory has two special directory entries:
  - `.` = the directory itself
  - `..` = the parent directory

Pathnames (Paths)

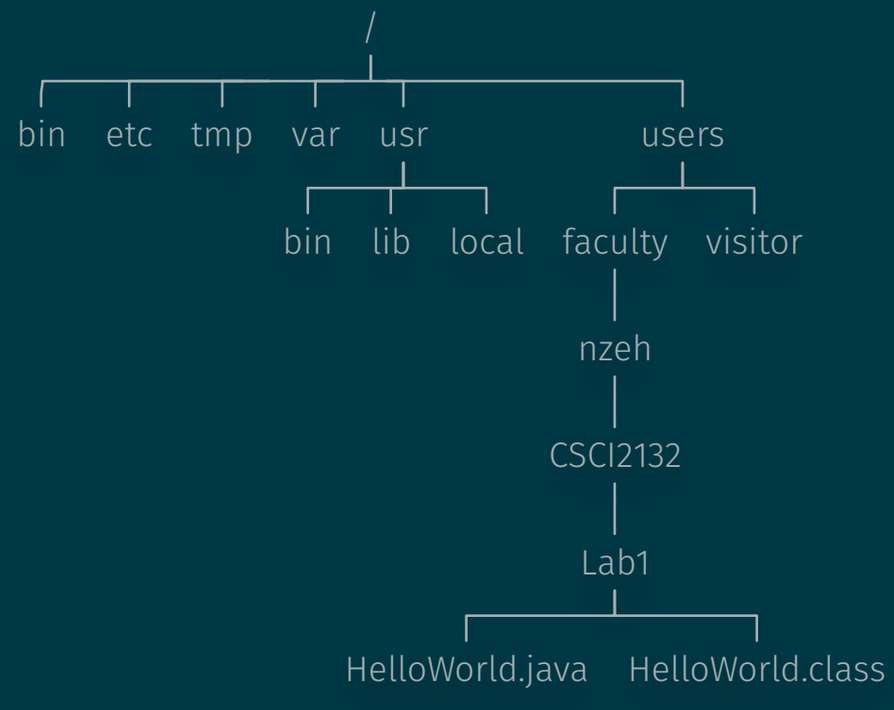


- Each file has a **name**.
- Two files in different directories can have the same name.
- Files are fully identified by their **pathnames** (paths).
- A pathname is a **sequence of directories, followed by a file name**.
- Pathname components are separated by /.

Examples:

- `/users/faculty/nzeh/CSCI2132/Lab1/HelloWorld.java`
- `/users/faculty/nzeh/CSCI2132`

Absolute and Relative Paths



- An **absolute path** starts with a `/`.
  - File is found by traversing the directory tree **from the root**.
  - **Example:** `/users/faculty/nzeh/CSCI2132/Lab1/HelloWorld.java`

- A **relative path** does not start with a `/`.
  - File is found by traversing the directory tree **from the current directory**.
  - **Examples** (current directory is `/users/faculty/nzeh`):
    - `CSCI2132`
    - `CSCI2132/Lab1/HelloWorld.java`
    - `../../visitor`
    - `./CSCI 2132/Lab1`

Components of a Pathname

Pathname = dirname + basename

Examples:

```
$ basename /home/ed/file.txt
file.txt
$ basename /home/ed/file.txt .txt
file
$ dirname /home/ed/file.txt
/home/ed
```

Useful Commands to Explore/Manipulate Directories

`ls paths`	List directory contents
`pwd`	Print current working directory
`cd path`	Change directory
`mkdir dirs`	Make directory(ies)
`mkdir -p paths`	Make directory(ies) and all ancestor directories
`rmdir dirs`	Remove empty directory(ies)
`mv path1 path2`	Move or rename file or directory
`mv -i path1 path2`	— " — (prompt before overwrite)
`rm paths`	Remove file(s) (directories with `-r`)
`tree paths`	Visualize directory contents (not a standard command)

A Small Exercise

Consider the following commands:

```
$ pwd
/home/ed
$ mkdir tmp
$ cd tmp
$ mkdir a b c
$ mkdir -p a/a1 a/a2/a21 a/a2/a22
$ cd a/a2/a22
```

What is the absolute current working directory?

What directory is `..`?

Do the following directories exist and what are their absolute paths?

`..` `../../b` `../../../c`

File Manipulation

`cat files`	show content of text file(s)
`more files` `less files`	— " —, paged
`head files`	show the first few lines of a file
`tail files`	show the last few lines of a file
`vi, emacs, pico, nano`	various text editors
`wc files`	word count(s) of the file(s) (learn about `-c`, `-w`, `-l` options)

File Permissions

Who is allowed to do what with a given file depends on the file's **owner** and **permissions**.

Users, Usernames, User IDs

Files and processes are owned by **users**.

Used to protect users working on the same system from each other.

User:

- Unique **username**, try `whoami`.
- Unique **user ID** (numeric ID corresponding to the username), try `id -u`.

Groups

A user is a member of at least one **group**:

- **Groupname** and **group ID** analogous to username and user ID.

List groups a user is a member of using `groups` or `id -G`.

Effective User and Group IDs

- Every process has an **effective user ID** and an **effective group ID**.

- Every file has a **file owner** and a **file group**.

- What a process can do with a file is determined by the file permissions and whether the effective user ID matches or effective group ID matches the file owner or file group.

File Permissions

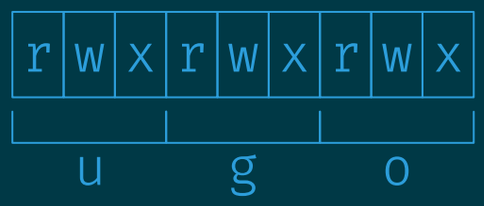
- A file can be allowed to be
  - **Read** (`r`)
  - **Written** (`w`)
  - **Executed** (`x`)
    - File: Run the file as a program
    - Directory: Change to the directory

- These permissions are set at three levels:
  - **User** (u)
  - **Group** (g)
  - **Others** (o)

File Permissions, Users, Groups

- Three sets of permissions (user, group, other)
- Which one determines what a process can do with a file?
  - If effective user ID = file owner: apply **user** permissions
  - If effective user ID ≠ file owner but effective group ID = file group: apply **group** permissions
  - If effective user ID ≠ file owner and effective group ID ≠ file group: apply **other** permissions

- File permissions written in octal:



Common permissions:

- `u=rwx,g=rx,o=rx` (`755`) (programs executable by everybody, modifiable by owner; directories accessible by everyone, modifiable by owner)
- `u=rw,g=r,o=r` (`644`) (data files readable by everybody, writable by owner)

Checking Permissions

Command: `ls -l`

Examples:

```
$ echo test > tmpfile.txt
$ ls -l tmpfile.txt
-rw-r--r-- nzeh csfac 5 Jan 8 03:01 tmpfile.txt
```

Other useful options:

- `-a`: List all files, also hidden ones (starting with `.`)
- `-t`: Order by time instead of name
- `-r`: Reverse sorting order
- Example: `ls -lt` = list files most recent file first

Changing Permissions

Command: `chmod mode files`

Examples:

`chmod 664 file.txt`	User/group: read/write Other: read
`chmod go-r file.txt`	Group/others: Remove read permission
`chmod u+x,og+r file.txt`	User: add execute permission Group/others: add read permission
`chmod u=rw,og= file.txt`	User: Set permissions to read/write Group/others: Disallow all access
`chmod a+r file.txt`	All: Add read permission
`chmod -R u+r+w+X dir1`	User: Add read/write permission Add execute permission if dir recursively for all files in dir1

Changing Owner and Group of a File

Commands: `chown user files`
`chgrp group files`

Examples:

`chown newuser file.txt`
Change owner of file.txt to newuser

`chown -R newuser files dirs`
Change owner of files and dirs to newuser, recursively for dirs

`chgrp newgroup file.txt`
Change group of file.txt to newgroup

`chgrp -R newgroup files dirs`
Change group of files and dirs to newgroup, recursively for dirs

Effective UserID and GroupID

Recall: Permissions of a process are determined based on matching effective UserID and GroupID to files' owners and groups.

How are the effective UserID and GroupID determined?

Changing Effective User and Group in the Shell

- `newgrp newgroup` logs in with a new effective group
  (user must be part of group `newgroup` for this to work)

- `su user`
  - Change effective user to `user`
  - For this to work, current user must be `root`
    (Do not try this on `bluenose`, sysadmins won't be happy.)

setuid and setgid bits

- Executable files can have two additional permission bits:
  - `setuid` (4000 oct): No matter who runs this program, the process will have effective user ID equal to the owner of the program.
  - `setgid` (2000 oct): No matter who runs this program, the process will have effective group ID equal to the group of the program.
- Another special bit:
  - `sticky` (1000 oct): Controls deletion of files in a shared directory (`man sticky`)

Further Reading

- UNIX book, chapters 1 and 2
- Read tutorials on `vi` and `emacs` in the UNIX book

Input/Output Redirection

- Default on Unix:
  - `stdin` = terminal (keyboard input)
  - `stdout`, `stderr` = terminal (screen output)

- Output redirection changes this

Output Redirection to Files

- `command > file` redirects the output of `command` to `file`.
  - `stderr` still goes to the terminal.
  - `file` is created if it does not exist.
  - If `file` exists, previous content is replaced (operation fails if `noclobber` is set).

- `command >> file` redirects the output of `command` to `file`.
  - `stderr` still goes to the terminal.
  - output is appended to `file` (old content is not replaced).

Input Redirection from Files

`command < file` reads input from file.
(E.g., useful for testing)

Examples:

- `sort < names.txt` reads lines of `names.txt` and prints them to `stdout` in sorted order.
- `sort < names.txt > sorted.txt` reads lines of `names.txt` and writes them to `sorted.txt` in sorted order.
- `mail csid < HelloWorld.java` sends the file `HelloWorld.java` to user `csid`.

Error Redirection

- `stderr` can be redirected similarly to `stdout`:

```
command 2> filename
```

  - `stdout` still goes to terminal
  - Note: "`2>`", not "`2␣>`"
  - An append version also exists: `2>>`

Redirection and File Descriptors

- `2>` may look cryptic at first, but
  - Every file has a file descriptor:
    - `0` = `stdin`
    - `1` = `stdout`
    - `2` = `stderr`
- Could have written `command 1> file` instead of `command > file`.
- `command 0< file` instead of `command < file`.

Redirecting stdout and stderr to the Same File

- `command > file` sends `stdout` to `file` but `stderr` to the terminal.
- `command 2> file` sends `stderr` to `file` but `stdout` to the terminal.

Pipes

Pipes come in two flavours:

- “Ad hoc” pipes created by joining commands using |
- Named pipes on the file system

Ad Hoc Pipes

- `command1 | command2` starts two processes running `command1` and `command2`:
  - `stdout` of `command1` goes to `stdin` of `command2`.
  - The terminal input goes to `stdin` of `command1`.
  - The `stdout` output of `command2` is written to the terminal.
  - `stderr` of both `command1` and `command2` are written to the terminal.

Named Pipes

- Named pipes are special files:
  - One process opens the file and writes to the pipe.
  - Another process opens the file and reads from the pipe.
  - The reading process reads exactly what's written by the writing process.

- Create a named pipe using `mkfifo pipename`
  (FIFO = first in-first out)

Building a Long Pipeline

- Break the problem into simple problems that can be accomplished using individual commands:
  - Sort the lines (`sort`)
  - Manipulate the contents of individual lines (`cut`, `sed`, `awk`)
  - Drop lines (`uniq`, `sed`)
  - …
- Add one stage at a time and test the output

Problem Example

The file `/etc/passwd` is in the following format:

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
user1:x:1000:1000:John Doe:/home/user1:/bin/tcsh
```

- Fields separated by colon
- 7th field is the user's shell

Problem: Count the number of distinct shells used by all users of the system (3 above).

Solution

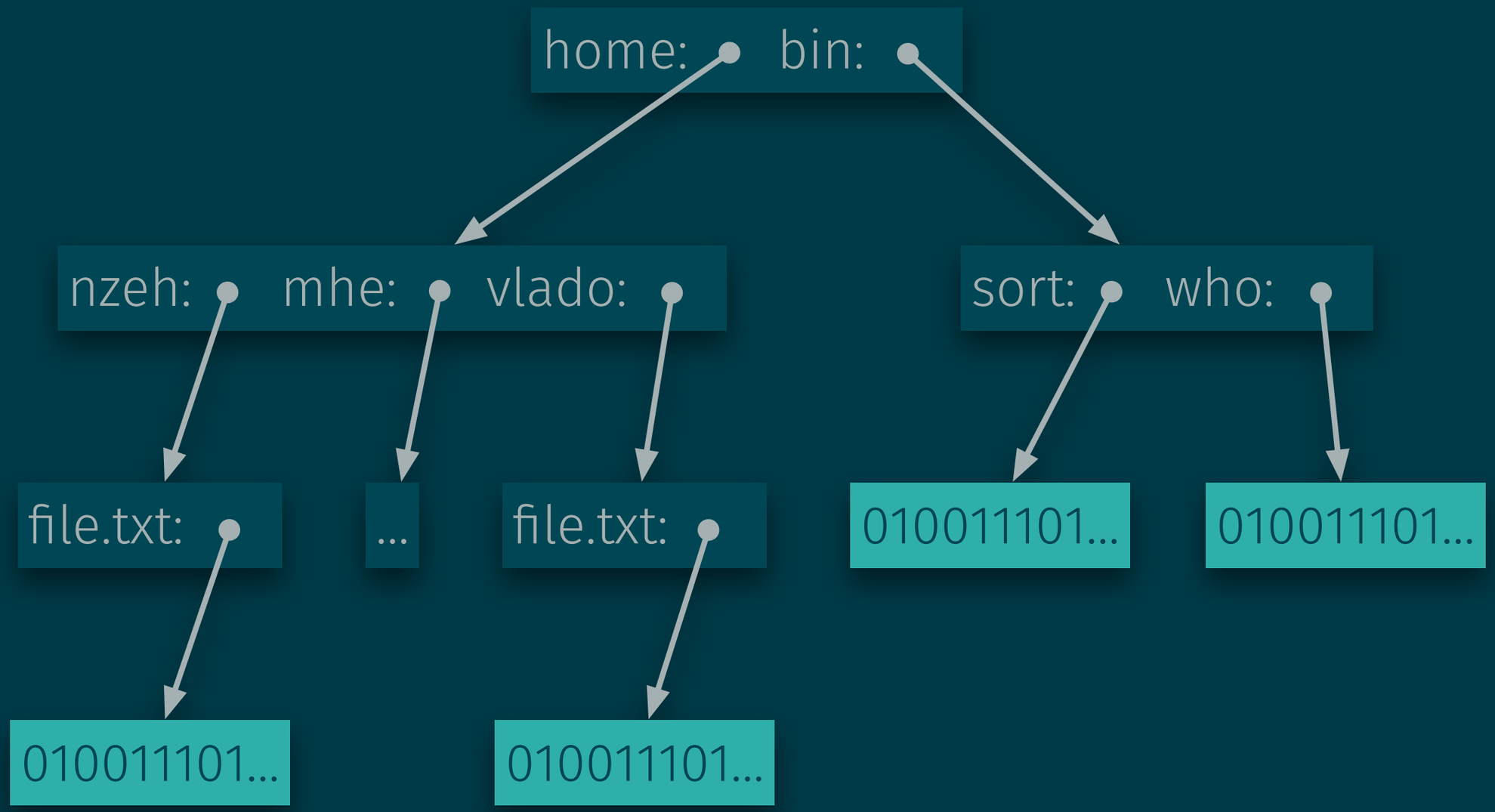
```
cut -d':' -f 7 /etc/passwd | sort | uniq | wc -l
```

or

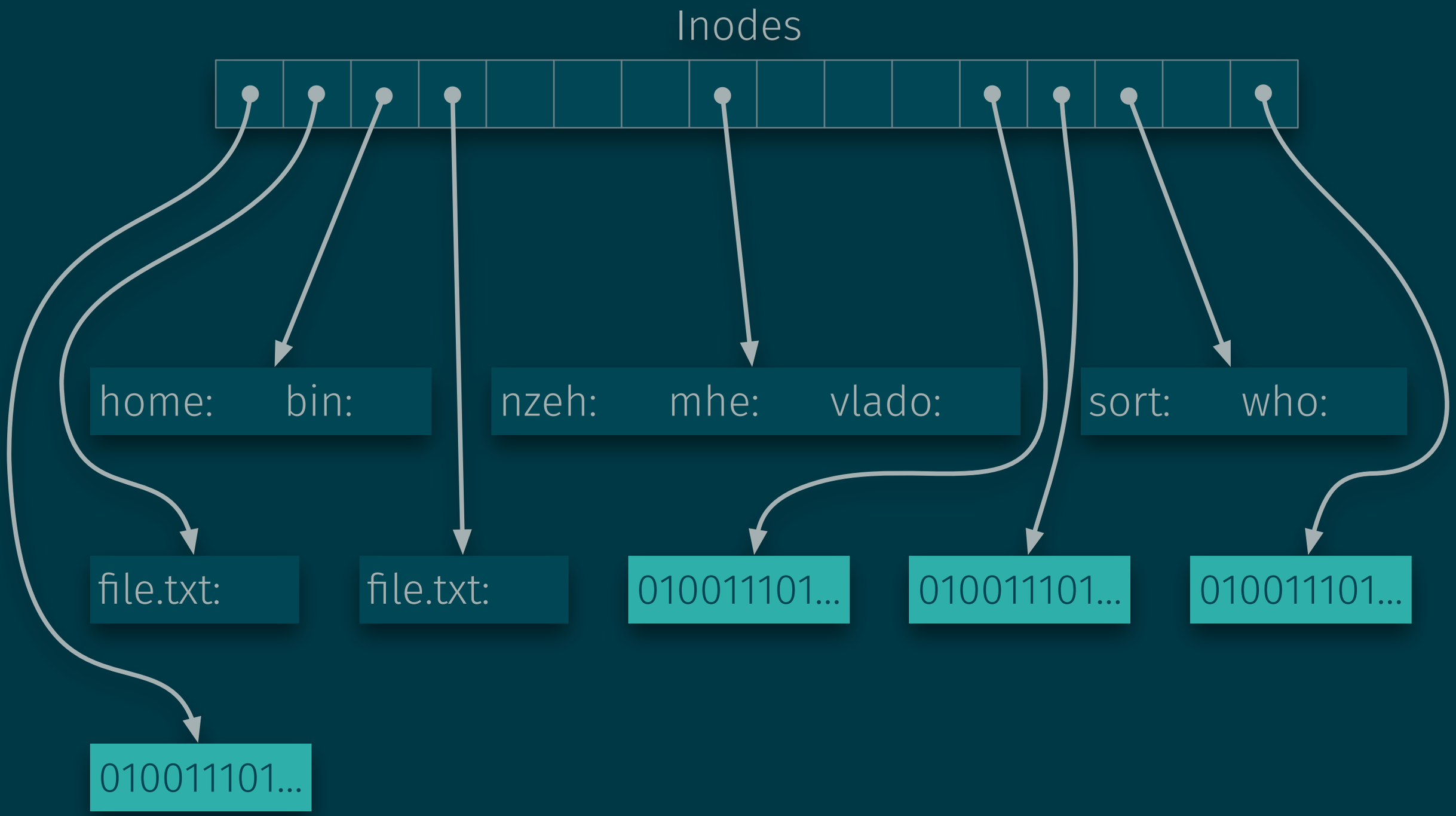
```
cut -d':' -f 7 < /etc/passwd | sort | uniq | wc -l
```

Inodes and Links

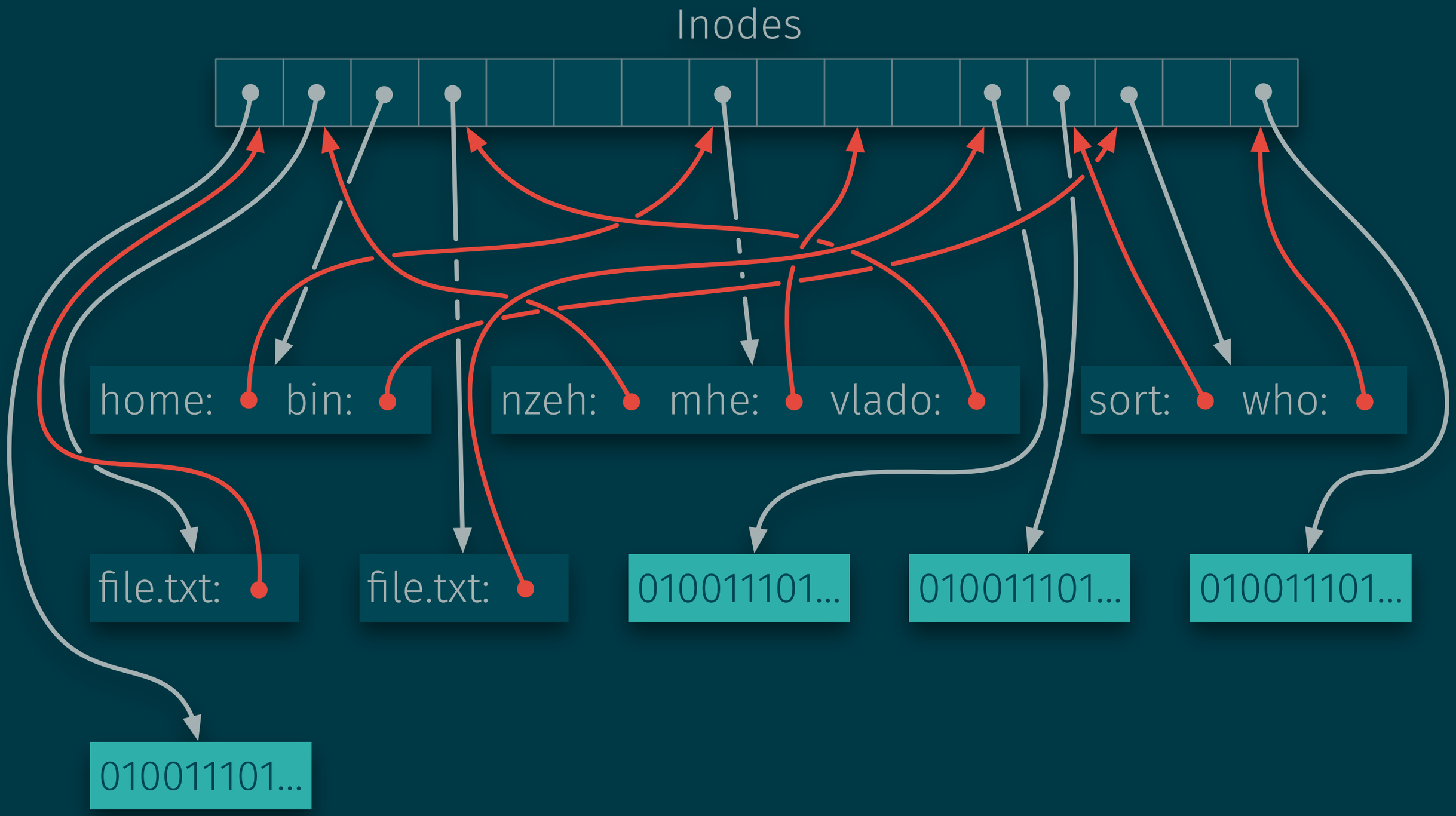
The way we have thought about the organization of the directory hierarchy so far:



Inodes and Links



Inodes and Links



Inodes

- Each file has a unique **inode number**
- One inode table per file system.
- Inode structure stores:
  - File type
  - File permissions
  - Owner and group IDs
  - Last modification and access time
  - Size of the stored object
  - Location of the data on disk

Creating Multiple Hard Links to a File

Advantage of separating directories and inodes:
A file can exist in multiples directories.

- Create additional hard link to the same file: `ln source target`
  - `source` and `target` now refer to the same inode and are indistinguishable.
  - `rm source` or `rm target` only removes link to the inode.
  - File is removed only once there is no longer any reference its inode.

Restrictions:

- `source` and `target` must exist on the same file system.
- Only one hard link to any directory.

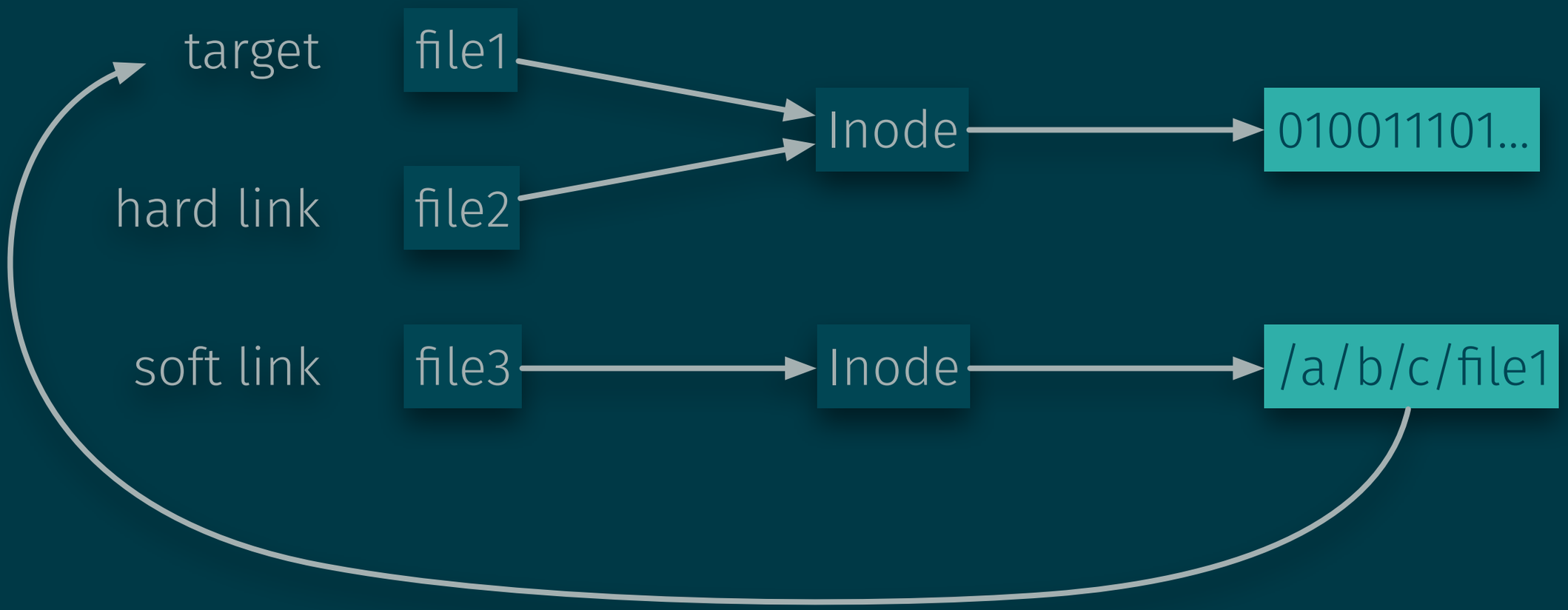
Inspecting Inode Information

- `ls -i` displays inode information
- The following example demonstrates that `ln file1 file2` makes both `file1` and `file2` refer to the exact same file:

```
$ cat “Hello world!” > file1.txt
$ ln file1.txt file2.txt
$ cat “Hallo, Welt!” >> file1.txt
$ cat file2.txt
Hello world!
Hallo, Welt!
$ ls -li
8635840546 -rw-------  2 nzeh  staff  9 25 Dec 16:02 file1.txt
8635840546 -rw-------  2 nzeh  staff  9 25 Dec 16:02 file2.txt
```

Soft Links

Soft links act as shortcuts:



Soft Links vs Hard Links

Advantages of soft links:

- Can cross file system boundaries
- Can point to directories
- Can point to another user's file/directory

Disadvantages of soft links:

- The link is not indistinguishable from the file it references.
- Less efficient in terms of time and space
- Backup and other processes need to deal with soft links carefully.
- `cp` does not copy the link but makes a copy of the referenced file.