Faculty of Computer Science, Dalhousie University *12-Sep-2018*

CSCI 2132 — Software Development

Lecture 4: Files and Directories

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Some hardware concepts
- Main UNIX concepts, Shells
- Logging in, PuTTY
- Some basic utilities and commands
  - date, clear, passwd, man
- Shell metacharacters
- 'cat' example, file redirection
- Logging out

4 Files and Directories

Slide notes:

Files and Directories

- Many concepts in Unix are either a **file** or a **process**
- **File** is a stream of bytes
- Many devices and constructs are seen as files:
  - regular files, stdin, stdout, stderr, keyboard, monitor, hard disk, CD/DVD, ...
- File is a good example of **abstraction**
- File is described by a general **interface**

Many concepts in Unix are either a file or a process. A *file* is an abstraction of data collection organized as a stream of bytes, from which we can read or to which we can write, or both. Many different concepts are seen as files, and this abstaction simplifies understanding of a complex system. For example, a file is what we consider a regular file save on disk, which can be opened to be read or written. The default communication connections of each process that we talked about—the standard input, output, and error (stdin, stdout, stderr)—are also files, the keyboard is a file, output to a terminal is a file, hard distsk, CD/DVD devices are files as well.

The concept of file is a good example of abstraction. We define file with a set of operations that can be applied to files, in other words, we describe a general interface to a file. After that, there is a large set of devices and constructs in general that can implement this interface, and we can use this interface without knowing any details about them.

The file interface includes operations such as opening and closing a file, reading a file, writing to a file, re-positioning in a file. Some files may be restricted to only some operations. For example, re-positioning is not normally available when the file is a keyboard.

There are the following seven types of files in Unix:

Seven Types of Files

1. Regular files
2. Directory files
3. Buffered special files (block devices)
4. Unbuffered special files (character devices)
5. Symbolic links
6. Pipes (named pipes)
7. Sockets

The command '`ls -l`' lists all files in a directory and shows many details about those files. (Note that the option `-l` in the command `ls -l` contains a lowercase letter L (`l`) and **not** the digit one (`1`).)

For example, the output the output of the command could be:

```
drwxr-xr-x 2 vlado csfac 4096 Sep 13 06:24 c
-rw-r--r-- 1 vlado csfac    0 Sep 13 06:34 file
```

The first letter of the output ('`d`' and '`-`' in the above list) indicates one of the seven types of files.

1. Regular files: The regular files are files containing text, executable programs, graphics, video, and similar. These are the files we usually think of by default under the name 'files'. In a long 'ls' output (`ls -l`), these files are indicated with the character '`-`'.

2. Directory files: The directories or folders are group of other files, but they are also a type of files called *directory* files. In a long 'ls' output, these files are indicated with the character '`d`'.

3. Buffered special files: are also called *block devices.* These files represent disk drives, USB keys and similar in a direct access mode. They are called buffered because a buffer in memory is used in reading and writing them. In a long 'ls' output, these file are indicated with the character '`b`'.

4. Unbuffered special files: are also called character devices, and represent devices such as the terminal and keyboard. Both, buffered and unbuffered special files are called device files. In a long 'ls' output, these files are indicated with the character '`c`'.

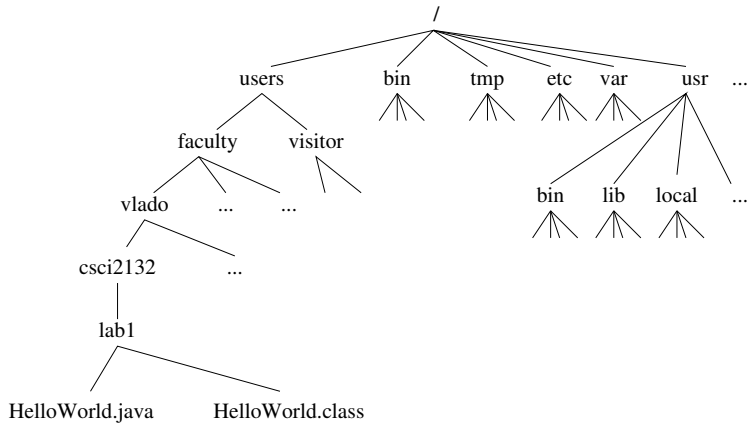
5. Symbolic links: are references to other files. Using them, we can access existing files through a different pathname (path). They are similar to shortcuts in Windows. In a long 'ls' output, these files are indicated with the character '`l`'.

6. Pipes: are used for inter-process communication, and they are also called **named pipes.** This file type should not be confused with pipes created between processes in a pipeline created as a compound shell command. In a long 'ls' output, these files are indicated with the character '`p`'.

7. Sockets: are used for inter-process communication, similarly to pipes. One of the differences is that the sockets are fully duplex-capable; i.e., processes can send data both ways, unlike pipes. Beside data, processes can also send file descriptors using sockets. In a long 'ls' output, the socket files are indicated with the character '`s`'.

4.1 Navigating Directory Structure

The figure below shows a part of the directory structure of the bluenose server, including the top, i.e., the *root* of the hierarchy. We can use it to show how a typical UNIX directory structure looks like.



The directory structure is a tree in the graph sense. We should not that this structure may not appear as a tree if we include symbolic links as edges, since the structure could potentially have cycles. This should become more clear when we talk about symbolic links.

There is a unique root node of the directory tree, called **root directory**, and it is denoted by '`/`'. The root directory typically contains several subdirectories, and maybe some other files. We can introduce the notions of parent directory and subdirectory. If a directory A contains a directory B directly, then A is the **parent directory** of B, and B is a **subdirectory** of A. The subdirectories of root further contain files and other directories, and this creates a tree structure.

Every directory contains two special directory entries: a dot (`.`), which is the directory itself; and dot-dot (`..`), which is the parent directory.

Some Notions in Directory Structure

- A tree with **root directory** (`/`)
- If a directory A contains directly directory B:
  - A is **parent directory** of B
  - B is **subdirectory** of A
- Each directory has two special directory entries:
  - dot (`.`) — the directory itself
  - dot-dot (`..`) — the parent directory

Pathname (Path)

- Each file has a **name**
- Files can have the same name if they are in different directories
  - Example: see `bin` in the previous figure
- To distinguish files with the same name, we use pathnames
- **Pathname** (or **path**) is a sequence of directories, finishing with a file name
- Directories are separated using character slash (`/`)
- Example:
  `/users/faculty/vlado/csci2132/lab1/HelloWorld.java`

Two Kinds of Paths

- **Absolute path** starts from root (initial slash `/`), examples:

```
/usr/bin
/users/faculty/vlado/csci2132/lab1/HelloWorld.java
```

– **Relative path** starts from the **current directory**; examples (if the current directory is 'vlado'):

```
csci2132
csci2132/lab1/HelloWorld.java
./csci2132/lab1/HelloWorld.java
../../visitor
./a.out
```

The notion of pathname is used to refer to each file. As we can notice in the previous figure, there can exist more than one file with the same name in different directories. For example, there are two directories with the name 'bin'. The files with the same name are distinguished by identifying the path through the directory structure that leads to them.

The path can be specified as an **absolute path,** which starts from the root directory (leading slash, `/`), or it can be a **relative path,** starting from the current directory. We recognize a relative path since it does not start with a slash. In some situations, it is important to start it with dot-slash (`./`) to explicitly indicate that the path starts from the current directory. One example such example is when we want to run a program in the current directory. For exmpale, if the program's name is `a.out` then the command `a.out` will not work unless the current directory (`.`) is a part of PATH environment variable. In that case, the program can be executed using the command `./a.out`. This remark jumps ahead of the material that we will cover, so do not worry if you do not exactly understand it at this point.

Parts of Pathname

– Pathname: dirname and basename
– Example commands:

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

– Note: blue text above is system output and red text is our input; we will use `$` or `>` as shell prompt.

Useful Commands related to Directories

– `ls` *paths* — list directory contents
– `pwd` — print working directory
– `cd` *path* — change directory
– `mkdir` *dirs* — make directory(ies)
– `mkdir -p` *paths* — whole paths, no errors
– `rmdir` *dirs* — remove empty directory(ies)
– `mv` *path1 path2* — move or rename directory or file
– `mv -i` *path1 path2* — prompt before overwrite
– `rm` *paths* — remove files but can remove directories with option `-r`; useful to consider `-f` and `-i`
– `tree` *paths* — note: not a standard Unix command

The option '`-i`' in commands `rm` and `mv` is called the "interactive" option and causes the programs rm and mv to ask user for confirmation before removing or overwriting an existing file.

A Small Exercise

Let us 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 our absolute current directory?
What directory is `..`?
Do the following directories exist and what are their absolute paths: `..`, `../../b`, and `../../../c`?

File Manipulation

- `cat` *files* — showing textual file(s) content
- `more` *files* — showing textual file content, paged
- `head` *files* — showing textual file content, first part
- `tail` *files* — showing textual file content, last part
- `vi`, `emacs`, `pico`, `nano` — file editors
- `wc` *files* — word count
  - learn about `-c`, `-w`, and `-l` options of `wc`

Editors: There are usually a number of text editors available in a Unix system. Two most popular ones are 'emacs' and 'vi'. Both of these editors are capable of handling large files and they are used by many software developers. We will learn some elements of emacs in this course, but you can learn about vi also.

4.2 File Permissions

One of the functions of an operating system is to protect access to date of a user from other users. The system of file permissions is the mechanism used in Unix for this task. We will first discuss the concepts of *users* and *groups*, and then discuss *file permissions* to understand how this access control is implemented.

Users: The Unix system keeps track of a set of users that can access the system. The users are represented with *usernames,* which are short textual identiers, and *userIDs,* which are non-negative integers. The userID 0 is reserved for the usernamer 'root', which is the special user that can access any file in the system. This user is sometimes called the root user or super-user.

The command '`whoami`' can be used to print out our username, and the command '`id -u`' to print userID.

The operating system is more efficient when working with non-negative integers as identifiers than strings, so that is why internally the system uses more userIDs than usernames.

Groups

- Every UNIX user is a member of a group
- A user can be member of multiple groups, but one is effective for a process
- Each group has a unique groupname and groupID
- Command to list groups user is member of: `groups`
- Command for more complete information: `id`
- Each process, including shell, has one effective userID and groupID
- Each file is owned by one user and one group: **file owner** and **file group**

File Permissions

- Each file has 3 sets of permissions:
  - file owner permissions (u)
  - file group permissions, (but not the user), (g)
  - permissions for others, (not user and not the group) (o)
- For each set, there are three true/false permissions:
  - read (r)
  - write (w)
  - execute (x)
- What these permissions mean for regular files and directories?