Faculty of Computer Science, Dalhousie University *17-Sep-2018*

CSCI 2132 — Software Development

Lecture 6: Pipes; Links and Inodes

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

Previous Lecture

- SVN
- wc command, pipelines
- Changing file permissions
- Changing user and group owners
- Redirection
  - standard input, output, and error

5.2 Pipes

The pipes are a way of redirecting the standard channels in such way that the standard output of one process feeds directly into the standard input of the next process. This is done in shell by writting two commands, one after another, with the pipe meta character (‘|’) between them. We can continue and connect more than two commands in such way, forming a complex command called *pipeline.*.

In order to use programs in a pipeline, then need to follow certain conventions. First of all, the programs should read their input from the standard input, and produce the output to the standard output, rather than reading or writing to files, databases, or similar resources. Such programs are called *filters,* because their processing can be described as “filtering” the standard input stream into the standard output stream. This model of computing is known as the **pipe-filter software architectural pattern.**

Slide notes:

Pipes

- Pipes are created by a shell
- Connect standard output of a process to standard input of another process
- Use of pipe metacharacter (`|`)
- A sequence of ‘piped’ processes is called a *pipeline*
- Example: Count the number of files in a directory
- Solution: `ls | wc -l`

Slide notes:

An Approach to Create Pipeline

- Break the problem into subproblems doable by individual commands
- Gradually build and test pipeline if using command line
- Consider sorting files in a directory and printing names of some of them, for example as in:
  - `ls | sort | tail`
  - `ls | sort | tail -n 3 | head -n 1`

The general steps of using pipelines in problem solving are: First, break the problem into a series of subproblems. Next, use a program to solve each subproblem.

For example, if we would like to count the number of files in the current directory, we can divide this into two subproblems: getting a list of files in the current directory and counting how many files are in this list. Thus we can write: `ls | wc -l`

Pay attention to the `-l` option. We do not use `-w` because file names can have multiple words; i.e., they make include whitespace characters in their names.

As a couple more examples, the following line makes a list of files, sorts it, and prints the bottom part of the list:
`ls | sort | tail`
or, the following line will create the same list, and print the third last line:
`ls | sort | tail -n 3 | head -n 1`

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 are separated by colon (‘`:`’)
The last (7th) field is a shell path. Write a command line to get the number of distinct shell paths in the file.
For example, this number would be `3` for the above file.

Solution

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

or

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

Some pipelines could be expressed as individual commands. For example, we could accomplish the same task in the above pipeline by using the following commands:

```
cut -d":" -f 7 /etc/passwd > tmp1
sort tmp1 > tmp2
uniq tmp2 > tmp3
wc -l tmp3
```

One note about the above solution is that we are processing the file in a step-by-step fashion in which each next step must wait until the previous step has finished. In a pipeline, the processes are running concurrently so processing is happening in the same time. In this way, we usualy use less disk space, and some processing where the data is continuously arriving we can obtain the some final results right away without waiting for all input data to be read.

By using a pipeline we also avoid creating a sequence of temporary files.

6 Inodes and Links

Slide notes:

Inodes and Links

- Each Unix file represented by *inode* (index node) internally
- Each file has a unique inode number
- Inode structure contains the following information:
  - file type
  - permissions
  - owner and group IDs
  - last modification and access time
  - size of the object being stored
  - location of the data on the disk

Each Unix file is represented by an *inode* (historically derived from *index node*), which is a data structure that contains information about:

- permissions
- owner and group IDs
- last modification and access time
- size of the object being stored
- location of the data on the disk

All inodes in a file system are stored on disk in a large inode table and each inode is associated wiith an unique inode number.

The data stored in a file is generally scattered in different locations in the file system, so we use this data structure to store information about location and order of these different file segments.

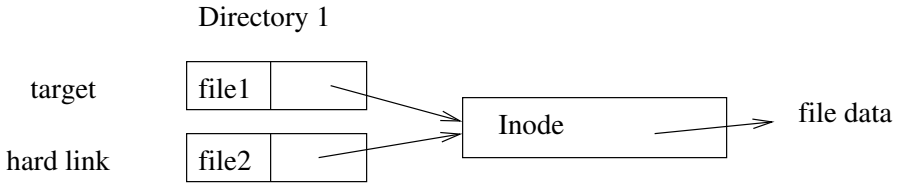
Every file has a unique inode number. A directory is a file containing a table of pairs (inode number, filename), where each pair corresponds to a file in the directory. There is one system-wide inode table, and inode numbers are indices into it.

In this way, the execution of commands such as `ls -l` that retrieve information regarding files in a directory can be efficient. This is a nice model that is effective and logical—a good example to follow when we design complex systems.

To reveal file inode numbers, we can use the option `-i` in command `ls`. For example, the command:
`ls -lid tmp`
may give an output like:
`84492732 drwx------ 2 ... tmp`
where the first number (84492732) is the inode number of the directory `tmp`. The option `-d` is used in the command ls to avoid printing the contents of the directory tmp.

6.1 Hard Links

Based on this basic knowledge of file representation, we can now learn how hard links work in Unix. A hard link is a directory entry (⟨inode number, filename⟩ pair) having a new filename, but duplicating an existing inode number. In other words, no new inode is created when we create a hard link. The following figure illustrates what a hard link is:



We need to have in mind that file1 and file2 in the above figure may reside in two different directories.

The following command can be used to create a hard link:
`ln target linkname`
For example, suppose that your current directory is the csci2132 directory you created for labs. You can create a hard link named HelloWorld.java to the file with the same name in subdirectory lab1 using:
`ln lab1/HelloWorld.java HelloWorld.java`

After this, you can compile and run the program using the hard link. Programs will not be able to tell any difference between the target and the hard link. Thus the original file is also a hard link to the same inode.

Characteristics of Hard Links

- Programs cannot tell difference between 'original' and 'target'
- Deleting a file just removes a link (unlink system call)
  - Only when number of links is 0, the space is freed
- Limitations:
  - same file system
  - no directories (exceptions possible on some systems)

A file is deleted when no hard links to its inode exist. In the above example, if we remove `~/csci2132/lab1/HelloWorld.java`, we can still use `~/csci2132/HelloWorld.java` to access the file.

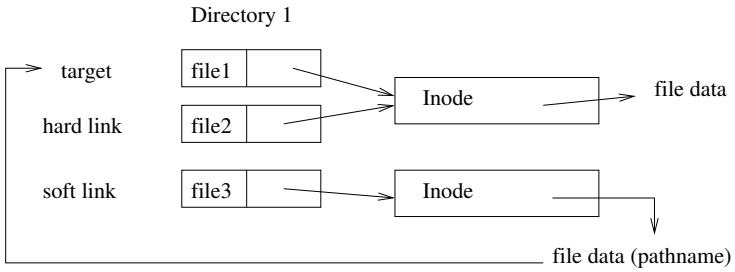
To check the hard link count of a given file, use the `ls -l` command.

We can see that hard links make it possible for us to give multiple names for the same file. If we have multiple hard links to the same file, then they all share the inode and this inode contains the number of hard links to it. Each time we remove a hard link, the hard link count is decreased by 1. The file is deleted when the count is 0.

There are some restrictions about creating hard links. First, hard links cannot span different disk drives. Second, directories cannot have hard links, as this would generally make directory structure not a tree any more, and this could possibly create an infinite recursion for the programs that traverse a directory structure. Additionally, it would be ambiguous which would be a parent directory (`..`) of a hard-linked directory. Other than these, there are no really design issues of why hard links on directories would not be allowed, so some systems may allow them in exceptional situations (e.g., some newer Mac OS systems in the Time Machine application).

6.2 Soft Links (Symbolic Links)

A soft link is a file that has the absolute or relative pathname of another file as its data part. When we create a soft link, we create a new inode. Strictly speaking, a soft link is not a link to a file, it is a link to a pathname. The pathname in a soft link could not exist any more, and the link will still be there, which could give us a wrong impression that the actual file exists, while it does not. The following figure illustrates the difference between hard links and soft links:



To create a soft link, we can use the command:
`ln -s target linkname`
If the target file is removed, the soft link still exists, but will result in an error if accessed. The '`ls -l`' command displays the pathname stored in a soft link; as as in the following output:

```
lrwxrwxrwx 1 user group 6 Sep 21 17:04 linkname -> target
-rw------- 1 user group 0 Sep 21 17:04 target
```

Soft links do not have the restrictions that apply to hard links. They also behave differently from hard links, as a hard link is essentially a different name of a file, while a soft link is a shortcut. They can span different file systems, and we can make soft links to directories.

Soft links allow users to link to other users' files, while all hard links share the same inode data, including user and group owners, and permissions.

Since soft links have fewer restrictions, why do we still create hard links? Some advantages of hard links are: They are more efficient to use: faster to access and they do not require additional inodes. We do not need to worry about which file is the 'master' file in order not to delete it by mistake. Hard links are particularly useful in some situations, such as keeping incremental backups of a file directory structure. Some programs do not treat symbolic links as 'real' files, which may or may not be desirable. For example, in a recursive copy, the cp command will copy symbolic links to regular files as symbolic links.

Side note: Emacs and Hard Links

You may notice that if you create a hard link to a texual file, and then try it edit with Emacs, the hard link will be "lost". This is what actually happens: Let us say that we have a file named `f1` with some contents and we create a hard link `f2` to this file. If we open the file `f1` with Emacs and start editing it, Emacs will rename the file to `f1~` and copy the content to a new file named `f1`. In this way, the backup file `f1~` will be now linked to `f2` and the link between `f1` and `f2` will be broken.

This is what we frequently want, but if we do not want this, we can configure Emacs to do this differently. We can put the following command in the `.emacs` configuration file:

```
(setq backup-by-copying t)
```

or

```
(setq backup-by-copying-when-linked
      t)
```

Either of these two commands will affect Emacs not to break hard links by making backups. Instead, Emacs will keep the original hard link and make backup by copying the file.

The Emacs configuration file `.emacs` is located in the user's home directory (i.e., in `~/.emacs`) and it is read and executed by Emacs each time Emacs is executed. The file is used to customize Emacs according to user preferences.

7 Wildcards and Regular Expressions

There are several situations where it is useful to be able to write a pattern that would match various strings that are similar in some way. One of those situations is when we want to apply a command to a list of files whose names follow certain pattern. We call these patterns *filename substitutions* or *wildcards.* The second example is when we want to match lines of a text file with a pattern, such as when using tools such as `grep`. This second kind of patterns are called *regular expressions.*

Slide notes:

Wildcards and Regular Expressions

- Similar patterns for string matching
  - ...but there are differences
- Wildcards or Filename Substitution
  - Used in command line, understood by shell
- Regular Expressions
  - Used with tools such as `grep`

7.1 Filename Substitution (Wildcards)

Let us consider the following example command:
`cp ../../lab1.bk/*.java .`
This command will copy all files whose names end with suffix '`.java`' from the directory '`../../lab1.bk`' into the current directory '`.`'. This is an example of a use of wildcard, and the question is how this actually gets executed? Is the '`cp`' command the only command that is "smart" in this way, or we can use this with other commands as well?

The answer is 'yes' — we can use the wildcards with other command as well since this is the shell feature and not a feature of the 'cp' command. When we issue a command like this, the shell will examine the command line for meta-characters such as '`*`' (asterisk, or star), and upon detecting it, shell will replace this string with all filenames, or paths to be more precise, that exist in the file system and whose paths match the pattern. The asterisk in the pattern can be replaced with an arbitrary part of filename.