UNIX AND LINIX IN INFOCOMMUNICATION
Week 5

O. Sadov

File Commands

UNIX tools support a standard set of commands for working with files and directories:

- `ls` — list directory contents. Let's look in '`man ls`'. We can simply specify files and directories as arguments and view the listing in different ways according to the options.

  Ok, let's take a look at our current directory — it's just '`ls`' without arguments. As we remember, after logging in, this is the home directory.

  ```
  ls
  ```

  We see some directories, but we don't see, for example, shell startup files. No problem, let's run:

  ```
  ls -a
  ```

  We can see the shell startup files and more — the directories "`.`" ("point") (current) and "`..`" ("double point") (top level) are also visible. Because that means "all" files and directories, including hidden ones. Hidden files in UNIX are just a naming convention — names must begin with a period. It is not an attribute as it is on Microsoft systems. Initially it was just a trick in the '`ls`' utility to hide the current and top directories, and then it came to be used as a naming convention to hide any file or directory.

  Also we can see directory listing recursively:

  ```
  ls -R
  ```

  Another very important option is the "long list" ("`-l`"):

  ```
  ls -l
  ```

  We see a table with information about the file/directory in the corresponding lines.

  - The first column is the file attribute. The first letter is the file type: "`-`" ("dash") is a regular file, "`d`" is a directory, and so on.

Then we can see read, write, and execute permissions for three user groups: owner, owner group, and everyone else. Once again, we see the difference between UNIX and Microsoft. In the first case it is an attribute, in the second case executability is just a naming convention: '.com', '.exe', '.bat'.

  - Some mystery column that we will discuss later.
  - Then we can see owner and owner group, size of file, time of modification and the name of file.

- `pwd` — print name of current/working directory
- `cd` — change directory. Without arguments – to home firectory.
- `cp` — copy files and directories. Most interesting option is '`-a|--archive`' with create recursive archive copy with preserving of permissions, timestamps, etc...
- `mv` — move (rename) files and directories.
- `rm` — remove files or directories.

  ```
  rm -rf ...
  ```

  means recursive delete without asking for confirmation.
- `mkdir` — make directories. If any parent directory does not exist, you will receive an error message:

  ```
  mkdir a/b/c
  mkdir: cannot create directory 'a/b/c': No such file or directory
  ```

  To avoid this, use the `-p` option:

  ```
  mkdir -p a/b/c
  ```

- `rmdir` — remove empty directories. If directory is not empty, you will receive an error message. Nowadays, running 'rm -rf something...' is sufficient in this case. But in the old days, when '`rm`' did not have a recursive option, to clean up non-empty directories, you had to create a shell script with 'rm's in each subdirectory and the corresponding 'rmdir's.

- `ln` — make links between files. Links are a very specific file type in UNIX and we will discuss them in more detail. If we look at the man page for the 'ln' command, we see a command very similar to 'cp'. But let's take a closer look:

```
cat > a
ln -s a b
ln a c
cat b; cat c
```

  At the moment everything looks like a regular copy of the file, but let's try to change something in the one of them:

```
cat >> c
cat a; cat b
```

  Wow, all the other linked files have changed too! We are just looking at the same file from different points, and changing one of them will change all the others. And in this they all seem to be alike. But let's try to delete the original file:

```
rm a
cat b; cat c
```

  In the first case, we can still see the contents of the original file, but in the second case, we see an error message. Simply because the first is a so-called hard link and the second is symbolic. We can see the difference between the two in the long ls list:

```
ls ?
```

  And we can restore access to the content for the symbolic link by simply recreating the original file:

```
ln b a
cat c
```

  Another difference between them is the impossibility of creating a hard link between different file systems and the possibility of such a linking for soft links. For more details on internal linking details, see the corresponding lecture. "Under the Hood"

Permissions

And finally, let's discuss file permissions. As we remember, we have the owner user, the owner group and all the others, as well as read, write and execute permissions for such user classes. And we have the appropriate command to change these permissions:

```
chmod [-R] [ugoa][-+=](rwx)
```

And as we understand it, permissions are just a bit field. As far as we understand, permissions are just a bitfield and in some cases it might be more useful to set them in octal mode — see for information on this.

"Under the Hood"

You can also change the owner and group for a file or directory by command '`chown`'.

```
man chown - change file owner and group
```

But keep in mind — for security reasons, only a privileged user (superuser root) can change the owner of a file. The common owner of a file can change the group of a file to any group that owner is a member of:

```
chown :group file...
chgrp group file...
```

Text Viewers

As we remember, UNIX was originally created to automate the work of the patent office, has a rich set of tools for working with text data. But what is text? Generally it is just a collection of bytes encoded according to some encoding table, originally ASCII. In a text file, you will not see any special formatting like bold text, italics, images, etc. — just text data. And this is the main communication format for UNIX utilities since the 1970s.

As you know, Microsoft operating systems have different modes of working with files — text and binary. In UNIX, all files are the same, and we have no difference between text and binary data. See details in ("Under the Hood").

Concatenating and spliting

The first creature that helps us work with text files is the "cat". Not a real "cat", but an abbreviation for concatenation. With no arguments, cat simply copies standard input to standard output. And as we understand it, we can just redirect the output to a file, and this will be the easiest way to create a new file:

```
cat > file
```

When we add filenames as arguments to our command, this will be a real concatenation — they will all be sent to the output. And if we redirect them to a file, we get all these files concatenated into an output file.

```
cat f1 f2... > all
```

If we can combine something, we must be able to split it. And we have two utilities for different types of breakdowns:

- tee — read from standard input and write to standard output and files
- split — split a file into fixed-size pieces

Text viewers and editors

What is it viewer? In the TTY interface, the man command seems like a good one — when you run it, you get paper manuals that you can combine into a book, put on a shelf, and reread as needed. On a full-screen terminal — before, Ctrl-S (stop)/Ctrl-Q(repeat) was enough for viewing, because at first the terminals were connected at low speed (9600 bits per second for ex.), and now special programs were used — viewers. Unlike text editors, viewers does not need to read the entire file before starting, resulting in faster load times with large files.

Historically, the first viewer was the "more" pager developed for the BSD project in 1978 by Daniel Halbert, a graduate student at the University of California, Berkeley. The command-syntax is:

```
more [options] [file_name]
```

If no file name is provided, 'more' looks for input from standard input.

Once 'more' has obtained input (file or stdin), it displays as much as can fit on the current screen and waits for user input. The most common methods of navigating through a file are Enter, which advances the output by one line, and Space, which advances the output by one screen. When 'more' reaches the end of a file (100%) it exits. You can exit from "more" by pressing the "q" key and the "h" key will display help. In the 'more' utility you can search with regular expressions using the 'slash' or the '+/' option. And you can search again by typing just a slash without regexp. Regexp is a very important part of UNIX culture and is used in many other programs and programming environments: (http://sdn.ifmo.ru/education/courses/free-libre-and-open-source-software/lectures/lecture-6)

The 'main' limitation of the more utility is only forward movement in the text. To solve this problem, an improved 'more' called 'less' was developed. The "less" utility buffers standard input, and we can navigate forward and backward through the buffer, for example. using the cursor keys or the PgUp/PgDown keys. A reverse search with a question mark is possible.

Text Editors

OK. We can create a file using the 'cat' utility and view the file using a viewer. But what if we need to change something, especially if we only have a TTY interface? And it is possible — we have a so-called line editor named 'ed'. The only interface for such an editor is the command line: (http://sdn.ifmo.ru/education/courses/free-libre-and-open-source-software/lectures/lecture-6).

So let's try to edit new file.

```
$ ed tst
tst: No such file or directory
```

At first — we will add some lines:

```
i
1 2
3 4
```

and we must end our input with one ‘dot’ per line.

```
.
```

Let’s take a look at our file, moving to the first line:

```
1
1 2

3 4

?
```

Seems good. Now we can add something to the end:

```
a
5 6
.
1
1 2

3 4

5 6

?
```

OK — we have 3 lines in the file now. And finally — let’s try to make a magic pass:

```
1,$s/\(.\) \(.\)/\2 \1/
1
2 1

4 3

6 5

?
```

This means: from the first to the last line, replace the lines where we have two separate letters separated by a space, exchanging those letters with places. And now 'write' and 'quit':

```
w
12
q
```

Let's check the result:

```
$ cat tst
2 1
4 3
6 5
```

But for what purposes can you use a line editor now that we have full-screen editors with a user-friendly interface? Of course, you can imagine a situation where your full-screen environment is broken and only the line editor will be the salvation. And in general I had such situations. But the main use case for `ed` is for automatic editing in scripts. Anything you need to change in the text data can be done with this editor, including sophisticated regex search and replace.

Moreover, we have a '`sed`' — stream editor, for editing text data in pipelines:

```
$ sed 's/\(.\) \(.\)/\2 \1/' < tst
1 2
3 4
5 6
```

As you can see, the original file does not change, all changes are simply sent to standard output:

```
$ cat tst
2 1
4 3
6 5
```

But UNIX-like systems also have full-screen editors, which can also be confusing for beginners. It was developed by Berkeley student Bill Joy for BSD initially as a visual mode for a line editor. It is a very fast and lightweight

editor that is part of the Single Unix Specification and the POSIX, which found on every UNIX-like system. The `VI` editor works on all types of terminals and generally requires only a conventional letter keyboard. You can work with it without the arrow keys, PgUp/Down or anything similar. There are actually very small keyboards out there that are optimized for '`vi`'.

But to work on it, you need to understand the basic concept of this editor: it can be in three states (`http://sdn.ifmo.ru/education/courses/free-libre-and-open-source-software/lectures/lecture-6`).

Immediately after launch, we find ourselves in the usual command mode and can switch to editing mode, for example, by pressing the "`[Insert]`" key. As we can see, the mode status in the lower left corner has changed to '-- INSERT --', and now we can edit our file. Pressing `Insert` again will change the state mode to '-- REPLACE --' and vice versa. Exit the editing mode by pressing `ESC`. The third mode can be accessed by pressing the colon key in command mode. This is 'ed' mode. In this mode, we can use the normal '`ed`' line editor commands and finish them with `ENTER`.

In command mode, you can find something with regex by slash and question marks, as in the '`less`' viewer. In improved VI (`vim`), you can use very useful visual mode by pressing `V`. After that you can delete the selection with '`d`' or just copy it with '`y`' (yank). Then you can paste it anywhere with '`p`' (paste). Moreover, you can use `[Ctrl-V]` to select a vertical box. To exit visual mode, simply press `ESC`.

Also you may look to `vimtutor` — a guide to Vim can be useful for beginners.

And the second most common editor is `Emacs`. This Richard Stallman's editor was the starting point for the GNU Project, along with GCC and UNIX utilities. EMACS means, for example, "Editing MACroS". An apocryphal hacker koan alleges that the program was named after Emack&Bolio's, a popular Cambridge ice cream store. But overall it is a really very powerful editor with macro extensions, allowing the user to override any keystrokes to launch the editor program. But unlike other editors that support macro-extensions, in Emacs they are implemented using the LISP programming language embedded on editor. At the time, LISP was very popular in artificial intelligence in the United States, and Stallman, who worked at the MIT Artificial Intelligence Lab, chose it as the editor extension language.

This implementation allows many LISP-based applications to be developed,

including a user-friendly interface for developers in various programming languages. Usually Emacs is a text editor with a simple graphical interface. But it can only be run in a text environment. The most commonly used keystrokes are:
`C-c C-x` — exit
`C-h t` — tutorial
`C-h i` — info

If you feel overwhelmed by the difficulty of Emaccs, you can see a personal psychoanalyst: `M-x doctor`. It would spoil the fun and hurt your recovery to say too much here about how the doctor works. But when you're ready, you may try to find the well-known Turing-test related AI program ELIZA on WikiPedia.

Also in the UNIX/Linux world, there are many other editors that may be more convenient for you, such as:

- `joe`, `nano` — simple text editors or
- `gedit`, `kate` – text editors from Gnome and KDE projects

Advanced Text Utilities

Searching

If we are talking about text data, finding some text is a common task. And in fact, these are two separate tasks — to find some text inside a file or text stream and to find a file, for example, by name in some directories.

For the first task, we have the '`grep`' utility which print lines matching a pattern.

```
man grep
```

Both fixed strings and regular expressions can be used as a pattern. Also you can do recursive search.

Another commonly used command is '`find`' — search for files in a directory hierarchy.

```
man find
```

You must set the starting point — the directory to start the search or starting points if you are interested in several directories and expressions with search criteria and actions. You may search by name with using of standard shell matching patterns, by time of modification or access, by size, by user and group, by permissins, file type, etc. You can use logical operators such as "and", "or" and "not" in your expressions.

Also you can do some actions when you find something that matches the criteria. The default action is '`print`'. You can also use formatted printing, list of found files, delete them, and execute commands with them. In '`exec`' actions, you can use curly braces to insert the name of the found file. But keep in mind — you must end your command with a semicolon, and to avoid interpreting this Shell character, you must escape it with a 'slash' ('/').

But the main drawback of '`find`' is the long execution time if you are looking in large deep directories. And to speed up this process, you can use the '`locate`' utility. It finds files by name from databases prepared by '`updatedb`' and does it incredibly fast. But you have to understand — '`updatedb`' is started automatically by the cron service at night. And if you only install the '`lookup`' toolkit or want to find something in the changed filesystem at this time — you have to update this database manually by running '`updatedb`'.

Utilities for Manipulation with a Text Data

Another operation that we often need is comparing files or directories. And we have some tools for this.

```
man cmp
```

The '`cmp`' utility compares the two files byte-by-byte and reports the position from which we have a difference. By this way we can compare binaries.

To compare text files '`diff`' utility can be used:

```
man diff
```

We can compare files, directories with the '`--recursive`' option. We can get the output as a set of commands for the '`ed`' editor or the '`patch`' utility. This

method of propagating changes was the first in the development of projects in the UNIX ecosystem and is still useful today.

Another important action with text data is sorting, and we have the '`sort`' utility which sort lines of text files:

```
man sort
```

To eliminate duplicate lines, we have the uniq utility, but first we have to sort our text stream:

```
sort file | uniq
```

We may output the first/last part of files by '`head`' and '`tail`' utilities. By default is the first or last 10 lines of standard input, or each FILE from arguments to standard output. You can set another number of lines as an option:

```
tail -15
```

Also in '`tail`' you can use the '`--follow`' option to display the appended data as the file grows.

More that, from text lines you can cut some fields, separated by some kind of separators by '`cut`' utility.

Also you can join lines of two files on a common field by '`join`' utility and merge lines of files by '`paste`'.

And finally, we have '`awk`', a scanning and templating language that can do this and other complex work on text files or streams.