Birzeit University - Faculty of Engineering & Technology
Electrical & Computer Engineering Department - ENCS313
Linux laboratory

Experiment #2
Shells - editors - pipes under Linux

0.1 Introduction

The experiment intends to make students at ease working under Linux: Expose the different shells and to the most popular text editors.

The power of Linux will be better illustrated if we combine several commands to get a more useful output. Examples will be provided on how to use these commands and better manipulate the obtained output.

0.2 Objectives

The objectives of the experiment is to learn the following:

- Get knowledge on the different available shells.
- Show how multiple commands can be combined through piping.
- Tackle more commands.
- Provide examples on how to use these commands.

0.2.1 Shells

The shell sits between you and the operating system, acting as a command interpreter. It reads your terminal input and translates the commands into actions taken by the system. The shell is analogous to **command.com** in DOS. When you log into the system you are given a default shell. When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files.

The original shell was the Bourne shell, **sh**. Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available. It has very good features for controlling input and output, but is not well suited for the interactive user. To meet the latter need the C shell, **csh**, was written and is now found on most, but not all, Unix systems.

Numerous other shells are available from the network. Almost all of them are based on either **sh** or **csh** with extensions to provide job control to sh, allow in-line editing of commands, page through previously executed commands, provide command name completion and custom prompt, etc. Some of the more well known of these may be on your favorite Unix system: the Korn shell, **ksh**, by David Korn and the Bourne Again SHell, **bash**, from the Free Software Foundations GNU project, both based on sh, the T-C shell, **tcsh**, and the extended C shell, **cshe**, both based on **csh**.

The shells have a number of **built-in**, or native commands. These commands are executed directly in the shell and dont have to call another program to be run. These built-in commands are different for the different shells.

As mentioned in the previous experiment, the following files will be *sourced* (read) when you either log on to the system or open a new command window:

- If your shell is **sh** (Bourne shell), your hidden file's name is `.profile`.
- If your shell is **bash**, your hidden file's name is `.bashrc`.
- If your shell is **csh**, your hidden file's name is `.cshrc`.
- If your shell is **tcsh**, your hidden file's name is `.tcshrc`.

The default shell for users is set in the file `passwd` that can be found under folder `/etc`. The file is in text mode and can be read by any user. However, only the `root` is allowed to change its content.

To do:

Open the file `/etc/passwd` and check on the default shell that is assigned to you.

0.3 Text editors

There are multiple text editors under Linux. Each of them present some advantages over others. However, they all do a neat job and selecting a particular text editor is just a matter of personal taste. Some text editors are described below:

vi: This is a console based plain text editor. It comes with every Linux/Unix installation which is an advantage. It is command-based.

vim: This console based plain text editor supports syntax highlighting and numerous plug-ins for specialized configurations and features.

emacs: This console based plain text editor supports the theory that more is better. It tries to support every feature possible. Emacs is a very powerful editor, though command-based.

pico: A console based plain text editor.

gedit: This is the default text editor for the Linux Gnome desktop. It supports syntax highlighting, printing, a variety of plug-ins, multi-language spell check, tabbed for multiple files, etc.

0.4 More on commands

As mentioned before, Linux is mostly command-based. You need to know as many commands as you can so as to feel at ease navigating the filesystem and manipulating directories, files and data. Keep in mind that the power of Linux doesn't come only from the rich set of commands it has but also from the different options that these commands take as argument. The output display of a specific command will be formatted differently if supplied with different options. Let's consider some of the commands that you will encounter so frequently:

- `which` command: Shows the full path of (shell) commands. It can be helpful if you need to locate command.

  Example:

  ```
  which ls
  ```

- `sort` command: Sort lines of text files.

  Example:

  ```
  touch sort.txt
  echo "13" >> sort.txt
  echo "12" >> sort.txt
  ```

```
echo "140" >> sort.txt
echo "20" >> sort.txt
echo "1" >> sort.txt
echo "12" >> sort.txt
sort sort.txt
```

You'll note that the sort occurs based on text.

To do sorting numerically, execute:

```
sort -n sort.txt
```

To redirect the output to a file:

```
sort -n sort.txt >> sort_new.txt
```

Or you can use the option `-o` with the command `sort` as follows:

```
sort -n sort.txt -o sort_new.txt
```

- `uniq` command: You would have noticed that enrty `12` is repeated twice in the above example. To remove duplicates, you can use the `uniq` command on sorted files as follows:

  ```
  uniq sort_new.txt sort_uniq.txt
  ```

- `grep` command: Print lines matching a pattern. The `grep` command is useful when you need to search in a folder, file or set of files for a particular pattern. For example, assume you have the file `sort.txt` that you created in the previous step. To search for the pattern `12` in file `sort.txt`, you execute:

  ```
  grep 12 sort.txt
  ```

  If you need to look for the pattern `12` in all files under the current foler, you execute:

  ```
  grep 12 *
  ```

  where `*` is a wild card that refers to *any* file.

  To look for the pattern `12` in all files with extension `txt`, you execute:

  ```
  grep 12 *.txt
  ```

  The behavior of the command `grep` can be inverted if used with option `-v`. In that case, all lines that do not include the pattern will get displayed. For example, if you execute:

  ```
  grep -v 12 sort.txt
  ```

  you'll get on the standard output all the lines of file `sort.txt` that do not include the pattern `12`.

  A useful option with command `grep` is the `-n` which prefixes each line of output with the line number within its input file:

  ```
  grep -n 12 sort.txt
  ```

  Below are some examples on how to use the command `grep`. You'll notice that it gets very useful once combined with *regular expressions*. We'll see more about regular expressions going forward:

`grep '[A-Z]' sort.txt`	Lines from `sort.txt` containing a capital letter
`grep '[0-9]' sort.txt`	Lines from `sort.txt` containing a number
`grep '[A-Z]...[0-9]' sort.txt`	Lines from `sort.txt` containing five-character patterns that start with a capital letter and end with a digit
`grep '\.pic$' filelist`	Lines from `sort.txt` that end in .pic

- `ps` command: Report process status. If run without options, the command `ps` will report about the current processes that are owned by the user. An example on the output of the command `ps` can be as follows:

```
PID TTY          TIME CMD
2532 pts/1    00:00:00 tcsh
2549 pts/1    00:00:00 ps
```

  In the above example, we can notice that the user has 2 running processes: `tcsh` and `ps` with PIDs `2532` and `2549` respectively.

  To see every process on the system using standard syntax, use the option `-e` (select all processes) or the combined options `-ef`. The option `-f` gives full-format listing.

  An example on the output of the command `ps -ef` can be as follows:

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 14:30 ?        00:00:03 init [5]
root         2     1  0 14:30 ?        00:00:00 [ksoftirqd/0]
root         3     1  0 14:30 ?        00:00:00 [watchdog/0]
root         4     1  0 14:30 ?        00:00:00 [events/0]
root         5     1  0 14:30 ?        00:00:00 [khelper]
root      2244  2076  0 14:32 ?        00:00:00 /usr/bin/gdm-binary -nodaemon
root      2290  2244  0 14:32 ?        00:00:00 /usr/bin/gdm-binary -nodaemon
root      2295  2290  0 14:32 ?        00:00:04 /usr/X11R6/bin/X :0 -audit 0 -au
gdm       2330  2290  0 14:32 ?        00:00:02 /usr/bin/gdmgreeter
root      2331  1926  0 14:32 ?        00:00:00 in.tftpd -s /tftpboot
root      2494  1926  0 14:37 ?        00:00:00 telnetd
root      2495  2494  0 14:37 pts/1    00:00:00 login -h bzurt -p
user      2532  2495  0 14:39 pts/1    00:00:00 -tcsh
user      2790  2532  0 14:47 pts/1    00:00:00 ps -ef
```

  Note the different columns that you get when running `ps -ef`. In particular, the column 1 refers to the process owner, column 2 shows the PID for each process and column 3 shows the parent ID for each process. Note as well that the Linux OS refers to each process by its unique PID and *not* by the process name.

- `kill` command: Send signals to processes. The default signal is the `SIGINT` which interrupts the execution of processes.

  The `kill` command takes a particular process ID as argument. It can be used as follows:

  `kill PID_num`

  where `PID_num` refers to a current process ID.

  If a particular process hangs, you need to run the command `ps` as described above in order to get its PID and then you terminate it using the command `kill`.

  Note that the command `kill` can send different signals to processes. Below is a shortlist for some of these signals (to see the list of all signals, execute `kill -l`):

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3	quit
SIGILL	4	illegal instruction (not reset when caught)
SIGTRAP	5	ctrace trap (not reset when caught)
SIGABRT	6	used by abort
SIGEMT	7	EMT instruction
SIGFPE	8	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal from kill
SIGURG	16	urgent condition on IO channel

To send a particular signal to a process, you can send it by the number. For example, you can send the signal `SIGALRM` to a process that has the PID 1234 as follows:

```
kill -14 1234
```

Alternatively, you can the signal by name after removing the prefix `SIG` as follows:

```
kill -ALRM 1234
```

Note:

To kill the last process that has been executed, you can use the shortcut `%` as follows:

```
kill %
```

The symbol % will be replaced by the PID of the last executed process.

0.4.1 The `wc` command

The command `wc` counts the number of words in a file. the command can be used as well to count the number of lines, characters, or words in a file. We'll illustrate its use in an example:

- Create a file called `wc.txt`.
- Execute the following command 5 times:

  ```
  echo "My name is coco" >> wc.txt
  ```

- Execute the following command:

  ```
  wc wc.txt
  ```

  You'll notice that you get 3 integers. These are respectively the number of lines, number of words and number of characters in the file `wc.txt`.

- If you want to get only the number of lines in file `wc.txt`, use the `-l` option of the command `wc` as follows:

  ```
  wc -l wc.txt
  ```

- If you want to get the number of words in file `wc.txt`, use the `-w` option of the command `wc` as follows:

  ```
  wc -w wc.txt
  ```

- If you want to get the number of characters in file `wc.txt`, use the `-c` option of the command `wc` as follows:

  ```
  wc -c wc.txt
  ```

0.5 Pipes

Suppose you want to count the number of files and directories contained in your home directory. One way to do that is to execute the following[1]:

- Execute the command:

  ```
  ls -l >> file.txt
  ```

- Execute the command:

  ```
  wc -l file.txt
  ```

The trick is that the command `ls -l` will return one line for each file/directory. The command `wc -l` will count the number of lines.

Another approach to determine the number of files/directories in your home directory bypasses the use of files. the Unix/Linux OS allows you to effectively connect two commands together. This connection is known as a *pipe*, and it enables you to take the output from one command and feed it directly into the input of another command. A pipe is effected by the character `|`, which is placed between the two commands. As such, the 2 above steps can be combined using pipes as follows:

```
ls -l | wc -l
```

When a pipe is set up between two commands, the standard output from the first command is connected directly to the standard input of the second command. You know that the `ls -l` command writes its list to standard output. Furthermore, you know that if no filename argument is specified to the `wc` command, it takes its input from standard input. Therefore, the list of files/directories that is output from the `ls -l` command automatically becomes the input to the `wc` command. Note that you never see the output of the `ls -l` command at the terminal because it is piped directly into the `wc` command.

A pipe can be made between *any* two programs, provided that the first program writes its output to standard output, and the second program reads its input from standard input.

It is also possible to form a pipeline consisting of more than two programs, with the output of one program feeding into the input of the next as follows:

```
ls -l | grep user | wc -l
```

0.5.1 The `cut` command

This command is used to extract various fields of data from a data file or the output of a command. The general format of the `cut` command is:

```
cut -cchars file
```

where `chars` specifies what characters you want to extract from each line of `file`. This can consist of a single number, as in `-c5` to extract character 5; a comma-separated list of numbers, as in `-c1,13,50` to extract characters 1, 13, and 50; or a dash-separated range of numbers, as in `-c20-50` to extract characters 20 through 50, inclusive. To extract characters to the end of the line, you can omit the second number of the range; so

```
cut -c5- file
```

extracts characters 5 through the end of the line from each line of `file` and writes the results

[1] Stephen Kochan, Patrick Wood, **Unix Shell programming** Third Edition. Sams Publishing, 456 pages.

to standard output.

Example

If you execute the command `who` (show who is logged on the system), you might get the following output:

```
root    console Feb 24 08:54
steve   tty02   Feb 24 12:55
george  tty08   Feb 24 09:15
ahmed   tty10   Feb 24 15:55
```

As shown, currently four people are logged in. Suppose that you just want to know the names of the logged-in users and don't care about what terminals they are on or when they logged in. You can use the `cut` command to cut out just the usernames from the `who` command's output:

```
who | cut -c1-8
```

The output will look as follows:

```
root
steve
george
ahmed
```

The `-c1-8` option to cut specifies that characters 1 through 8 are to be extracted from each line of input and written to standard output.

A more interesting command would be:

```
who | cut -c1-8 | sort
```

The output will look as follows:

```
ahmed
george
root
steve
```

The `cut` command can take a delimiter using the option `-d` and show a particular field using the `-f` option. In such a case, the format becomes:

```
cut -ddchar -ffields file
```

where `dchar` is the character that delimits each field of the data, and `fields` specifies the fields to be extracted from `file`. Field numbers start at 1, and the same type of formats can be used to specify field numbers as was used to specify character positions before (for example, `-f1,2,8, -f1-3, -f4-`).

Example

```
cut -d:  -f1 /etc/passwd
```

Note that if the `-d` option is not supplied, `cut` uses the tab character as the default field delimiter.

0.5.2 The `paste` command

The `paste` command is sort of the inverse of `cut`: Instead of breaking lines apart, it puts them together. The general format of the `paste` command is:

```
paste file1 file2 ...  filen
```

where corresponding lines from each of the specified files are *pasted* together to form single lines that are then written to standard output. The use of the command `paste` can be illustrated in the following example:

- Create the 2 files: `names.txt` and `numbers.txt`
- In the file `names.txt`, type the following:

```
Mohammad
Khalil
Fatmeh
Amin
Manal
```

- In the file `numbers.txt`, type the following:

```
123456
654321
112233
332211
615243
```

You can use `paste` to print the names and numbers side-by-side as follows:

```
paste names.txt numbers.txt
```

The output will look as follows:

```
Mohammad 123456
Khalil   654321
Fatmeh   112233
Amin     332211
Manal    615243
```

Each line from `names.txt` is displayed with the corresponding line from `numbers.txt`, separated by a tab.

Assume we have as well a file called `addresses.txt` that contain the following:

```
Jerusalem
Ramallah
Jenin
Hebron
Tulkarem
```

We can use the command `paste` on multiple files as follows:

```
paste names.txt numbers.txt addresses.txt
```

The output will look as follows:

```
Mohammad 123456   Jerusalem
Khalil   654321   Ramallah
Fatmeh   112233   Jenin
Amin     332211   Hebron
Manal    615243   Tulkarem
```

You can use the `-d` option if you want to use a separator different than the tab character. For example, if you execute the following command:

```
paste -d'+' names.txt numbers.txt addresses.txt
```

you get the following output:

```
Mohammad+123456+Jerusalem
Khalil+654321+Ramallah
Fatmeh+112233+Jenin
Amin+332211+Hebron
Manal+615243+Tulkarem
```

You can as well use the `-s` option of the command `paste` to paste together lines from the same file, not from alternate files. The effect is to merge all the lines from the file together, separated by tabs, or by the delimiter characters specified with the -d option as follows:

```
paste -s names.txt
```

The output will look as follows:

```
Mohammad Khalil Fatmeh Amin Manal
```

If you execute the following command:

```
paste -s -d '+' names.txt
```

you get the following on the standard output:

```
Mohammad+Khalil+Fatmeh+Amin+Manal
```

0.5.3 The `find` command

In the previous experiment, we've seen the `grep` command that helps users search in a file, set of files or set of directories. Regular expressions and wild cards can be used to give more power to the search operation.

The `find` command will recursively search the indicated directory tree to find files matching a type or pattern you specify. `find` can then list the files or execute arbitrary commands based on the results.

For example:

```
find .  -name coco.txt -print
```

will search for the file `coco.txt` starting the current directory as well as all subdirectories. Once it encounters a file or subdirectory that holds the name `coco.txt`, it will display the path that leads to that directory or file. Note that the option `-print` is not necessary since it represents the default behavior of the command `find`.

The command:

```
find / -name coco.txt -print
```

will search for the file `coco.txt` starting the root directory. The search will thus be exhaustive and might take a lot of time.

The search can be done on partial file names of directory names as well using wild cards. For example:

```
find .  -name *coco* -print
```

will search for any file or directory containing the string `coco`. A better example would be:

```
find .  -name a*bcd*e.txt -print
```

will search for any file or directory that start with the letter `a`, contain the string `bcd` and end with the letter `e` and has the extension `.txt` starting the current directory. You can thus notice the power of Linux commands once combined with regular expressions.

The command `find` can execute arbitrary commands on obtained results as well. For example, a user can decide to move the file `coco.txt` one directory up once it has been located as follows:

```
find -name coco.txt -exec mv {} ../ \ ;
```

Note that the command argument, {}, replaces the current path name. The end of the command

is indicated by \ ;. Other examples that relate to the command `find` are included below[2]:

- To find all files newer than the file library.txt, execute:

  ```
  find .  -newer library.txt -print
  ```

- To find all files with general read or execute permission set, and then to change the permissions on those files to disallow this:

  ```
  find .  \( -perm -004 -o -perm -001 \) -exec chmod o-rx {} \ ; -exec ls -al
  {} \ ;
  ```

0.5.4 The `tr` command

The `tr` command translates or deletes characters from standard input[3]. The general form of the command is:

```
tr from-chars to-chars
```

To get a feeling about the bahavior of the command `tr`, execute the following:

- Create a new file called `tr.txt`.
- Write the following phrase in file `tr.txt`:

  ```
  The UNIX operating system was pioneered by Ken Thompson and Dennis Ritchie at
  Bell Laboratories in the late 1960s.  One of the primary goals in the design
  of the UNIX system was to create an environment that promoted efficient program
  development.
  ```

- Run the command:

  ```
  tr e x < tr.txt
  ```

  and note the output.

The input to `tr` must be redirected from the file `tr.txt` because `tr` always expects its input to come from standard input. The results of the translation are written to standard output, leaving the original file untouched.

Example

If you run the following command:

```
cut -d:  -f1,6 /etc/passwd
```

You might get an output similar to the following:

```
root:/
cron:/
bin:/
uucp:/usr/spool/uucp
asg:/
steve:/users/steve
other:/
```

You can translate the colons into tab characters to produce a more readable output simply by tacking an appropriate `tr` command to the end of the pipeline:

```
cut -d:  -f1,6 /etc/passwd | tr :  '  '
```

The output look as follows:

[2]Extracted from the book entitled: **Introduction to Unix** by Frank G. Fiamingo, Linda DeBula, Linda Condron - University Technology Services - The Ohio State University.

[3]Stephen Kochan, Patrick Wood, **Unix Shell programming** Third Edition. Sams Publishing, 456 pages.

```
root  /
cron  /
bin   /
uucp  /usr/spool/uucp
asg   /
steve /users/steve
other /
```

The octal representation of a character can be given to `tr` command in the format

`\nnn`

where `nnn` is the octal value of the character. For example, the octal value of the tab character is 11. If you are going to use this format, be sure to enclose the character in quotes. For example, the `tr` command:

```
tr :  '\11'
```

translates all colons to tabs, just as in the preceding example. Below is the list of characters that you'll often want to specify in octal format:

Character	Octal Value
Bell	7
Backspace	10
Tab	11
Newline	12
Linefeed	12
Formfeed	14
Carriage Return	15
Escape	33

Example

The command `date` prints or sets the system date and time. To set date and time, you need to be an admin user (i.e. `root`). The output of the command `date` is as follows:

```
Mon Feb 7 12:11:49 JST 2011
```

You can translate all spaces into newline characters using the `tr` command as follows:

```
date | tr ' ' '\12'
```

The output will look as follows:

```
Mon
Feb
7
12:11:49
JST
2011
```

If you want to translate all lowercase letters in file `addresses.txt` to their uppercase equivalents, you execute the following:

```
tr '[a-z]' '[A-Z]' < addresses.txt
```

To change all uppercase letters in file `addresses.txt` to their lowercase equivalents, you execute the following:

```
tr '[A-Z]' '[a-z]' < addresses.txt
```

The `-s` option of command `tr` is used to *squeeze* out multiple occurrences of characters. In

other words, if more than one consecutive occurrence of a character occurs after the translation is made, the characters will be replaced by a single character.

For example, execute the following steps:

- Create a new file called `new_tr.txt`
- Type the following in file `new_tr.txt`:

```
my      name           is    coco.
I       am a student           at Birzeit                  University
Thank             you.
```

- Execute the following command:

```
tr -s ' ' ' ' < new_tr.txt
```

- Note the new output you get after executing the previous command.

The option `-d` of the command `tr` can be used to delete single characters from a stream of input. For example, you can execute the following command:

```
tr -d ' ' < new_tr.txt
```

and note the output that you get.

As an additional example on how useful the command `tr` is, you might want to delete all digits from a file. You can thus execute the following command:

```
tr -d '[0-9]' < new_tr.txt.
```