

Unix Fundamentals – Filters and Regular Expressions

Marek Kozłowski

Faculty of Mathematics and Information Sciences

Warsaw University of Technology

*Never use copy&paste for the following exercises. Retype all commands manually!
Don't just read those exercises and examples. Do them, please!*

1. Stream redirection

- (a) In this example ‘cat’ reads from the keyboard (*stdin*) and writes to a file (*stdout*):

```
$ cat > file1a
```

Finish by pressing ‘Ctrl d’ (generates the *end-of-file* character) or ‘Ctrl c’ (sends a SIGINT termination signal to a foreground process group). Check the result:

```
$ cat file1a
```
- (b) Remove a file contents (not a file! just its contents!) by overwriting it with an empty string:

```
$ echo -n > file1a
```

```
$ cat file1a
```

As you remember we used this syntax for removing bash history two labs ago.
- (c) Appending vs. overwriting example:

```
$ date > file1c; date > file1c; date > file1c
```

```
$ cat file1c
```

```
$ date > file1c; date >> file1c; date >> file1c
```

```
$ cat file1c
```
- (d) Keeping a log file permanently open for writing is a bad practice. A smarter way of appending control messages is illustrated by the following example:

```
$ echo "First message" >> file1d
```

```
$ echo "Second message" >> file1d
```

```
$ echo "Third message" >> file1d
```

```
$ cat file1d
```
- (e) ‘tail’ can handle appended lines. Create a file and open it with ‘tail -f’:

```
$ touch file1e; tail -f file1e
```

Open another terminal and append a few lines to this file by executing the following command repeatedly:

```
$ date >> file1e
```

End ‘tail’ by pressing ‘Ctrl c’.
- (f) The files from previous exercises won’t be necessary anymore. Keep your home tidy:

```
$ rm file1?
```

(we used this wildcard so many times it doesn’t need explanation, does it?)
- (g) Practice appending settings to a configuration file. This example shows how you can disable receiving *writes*:

```
$ echo "mesg n" >> ~/.bashrc
```
- (h) Learn ignoring output, ignoring errors and ignoring both:

```
$ date
```

```
$ date > /dev/null
```

```
$ date invalid-argument
```

```
$ date invalid-argument 2>/dev/null
```

```
$ date >/dev/null 2>&1
```

- (i) File finding revised. During the previous labs we used the following search:
- ```
$ find /etc -name "*.conf" -mtime +60 2>/dev/null
```
- It eliminates *Permission denied* messages by ignoring the *stderr* stream.

## 2. Command substitution

- (a) Create a file which name is its exact creation date and time:
- ```
$ touch "`date`; ls
```
- Note that double quotes are necessary because the `'date'` output stream contains spaces. We'd like them to be a part of the name rather than `'touch'` argument separators. If in doubt – compare the result of:
- ```
$ touch `date`; ls
```
- The cleanup must be done manually.
- (b) Listing all Linux kernel modules:
- ```
$ find /lib/modules/`uname -r`/kernel -name "*.ko.zst"
```
- (c) In case you've forgotten:
- ```
$ echo My name is: `whoami`
```

## 3. Filters

- (a) Any file viewer discussed the last week can be used as a pipeline filter:
- ```
$ set | more
$ set | less
$ set | head
$ set | tail
```
- (b) All filters can be used as standalone commands. For unknown reason most students use:
- ```
$ cat /etc/services | more
```
- instead of just:
- ```
$ more /etc/services
```
- (c) We can also mix both forms. The seventh line of a file is the last one of the first seven lines:
- ```
$ head -n7 /etc/passwd | tail -n1
```
- (d) In Arch Linux the package manager is called `'pacman'` (no, it's not a game). Let's browse all installed packages:
- ```
$ pacman -Q | less
```
- (e) `Word count` counts lines, words and characters:
- ```
$ wc /etc/passwd
```
- With the `'-l'` (`'--lines'`) option – only lines (precisely: EOL characters). Let's check the examples from the slide #18 and count shell variables:
- ```
$ set | wc -l
```
- environment variables:
- ```
$ env | wc -l
```
- entries in the working directory:
- ```
$ ls -A | wc -l
```
- packages installed in Arch Linux:
- ```
$ pacman -Q | wc -l
```
- local and remote users:
- ```
getent passwd | wc -l
```
- (f) `'wc'` as a regular command prints a file name. In a pipeline there is no file name, this may be convenient in some cases. Let's compare:
- ```
$ echo "Number of /etc/passwd lines: `cat /etc/passwd | wc -l`"
$ echo "Number of /etc/passwd lines: `wc -l /etc/passwd`"
```
- (g) By default environment variables are not sorted:
- ```
$ env
```
- but it is not a problem:

```
$ env | sort
```

We can also easily sort variables by values:

```
$ set | sort -t= -k2
```

(fields separated by '=', sorting by the second field)

- (h) Reading a random byte generator is not a good idea:

```
$ cat /dev/urandom
```

(stop by pressing 'Ctrl c')

Moreover some non-printable characters can change our terminal settings. Close your terminal emulator window and open another one. Then run this command correctly:

```
$ cat /dev/urandom | strings
```

4. *grep* – basics

- (a) What syntax for 'sort' did we use?

```
$ history | grep sort
```

- (b) On which TCP port does IMAP4 over SSL (secure mailbox access protocol) operate?

```
$ grep tcp /etc/services | grep imaps
```

- (c) Why storing secrets as string literals in code may be a bad idea?

```
$ strings /bin/bash | grep License
```

- (d) IP addresses for our servers start with '194.29.178'. Some services have been configured use them. What services? – oh, we did it such a long time ago... (unsure about the galaxy). But with *grep* sclerosis is not a problem anymore:

```
$ grep -R "194.29.178" /etc
```

It complains that we don't have permissions to access some configuration files. Let's ignore those messages as we've learned:

```
$ grep -R "194.29.178" /etc 2>/dev/null
```

- (e) 'zgrep' can operate on compressed files:

```
$ cp /etc/passwd .
```

```
$ gzip passwd
```

```
$ zgrep uszatekm passwd.gz
```

- (f) Are there utilities similar to 'zgrep' for other archive formats?

```
$ ls /usr/bin | grep grep
```

- (g) Let's print all lines containing the letters 'a' and 'c' and not containing 'u':

```
$ grep a /etc/passwd | grep c | grep -v u
```

5. *grep* with POSIX regular expressions

Before you start this exercise make sure you've read the slides about regular expressions and you understand the syntax!

- (a) Let's display a sample configuration file:

```
$ cat /etc/ssh/sshd_config
```

(the 'd' suffix indicates it is an SSH server configuration)

It is a typical configuration files. Most lines are commented out. Some are empty. Can we display only meaningful lines? At first we omit lines that start with a comment:

```
$ grep -vP '^#' /etc/ssh/sshd_config
```

But wait! If some blanks precede '#' it is still a commented out line:

```
$ grep -vP '^[ \t]*#' /etc/ssh/sshd_config
```

OK, now let's learn eliminating empty lines. An empty line contains at most some blanks:

```
$ grep -vP '^[ \t]*$' /etc/ssh/sshd_config
```

Let's combine it together:

```
$ grep -vP '^[ \t]*(#|$)' /etc/ssh/sshd_config
```

- (b) Display environment variables' names ('-o' prints only matching strings):

```
$ env | grep -oE '^[^=]*' | sort
```

or the values:

```
$ env | grep -oE '[^=]*$' | sort
```

- (c) All program files are placed in the `/usr/bin/` directory. Do any program names both start and end with vowels?
- ```
$ ls /usr/bin | grep -P '^[aeiou].*[aeiou]$'
```
- But wait! What if a program name is one character long? Some small correction is desired:
- ```
$ ls /usr/bin | grep -P '^[aeiou](.*[aeiou])?$'
```
- or
- ```
$ ls /usr/bin | grep -P '^[aeiou].*?[aeiou]$'
```
- (d) The 4<sup>th</sup> field of the `/etc/passwd` file contains a number called a GID. Are there any users with GIDs containing only odd numbers? First let's take a look (again?) at the file contents:
- ```
$ cat /etc/passwd
```
- We need to skip first 3 fields. A field can be described by the following regexp:
- ```
[^:]*:
```
- (any sequence of characters other than colons followed by a colon)
- OK, we've got all we need:
- ```
$ grep -P '^(^:)*{3}[13579]*:' /etc/passwd
```
- (e) Are there any users with GIDs from the range 900 – 1000? Remember: regexps describe strings not numbers!
- ```
$ grep -P '^(^:)*{3}9...' /etc/passwd
```
- or more elegantly:
- ```
$ grep -P '^(^:)*{3}9[0-9]{2}:' /etc/passwd
```
- (f) Try to build a regular expression that matches any valid IPv4 address. Check if it works:
- ```
$ grep -RP 'your_regexp_here' /etc 2>/dev/null
```
- and apply corrections if necessary. Honestly, it's not easy :-)
- (g) Continue with your own experiments.