

Seminar I

1. Attendance requirements (see course website)
2. Connection to campus/exam servers
3. Tricks and shortcuts:

The mouse is really neat for copy/paste. Everything you select in the console is already copied to the clipboard. Right-click (Windows) or middle-click (Linux) will paste.

- There are several keyboard shortcuts, which although identical to those used in Windows, do totally different things
 - Tab - Autocomplete
 - Up/Down arrow - navigate through the history of commands
 - Ctrl-C - Stop the currently running program; really useful when you have an infinite loop
 - Ctrl-S - Lock the console. You will be tempted to save your work with this, and then get confused.
 - Ctrl-Q - Unlock console. This is the antidote for Ctrl-S. Note that everything you typed while the console was locked will show up after you press this.
 - Ctrl-Z - Suspend the execution of the current program. Do not use this to "undo" anything
 - Ctrl-A - Jump to the beginning of the line
 - Ctrl-E - Jump to the end of the line
 - Ctrl-F - Move forward on character
 - Ctrl-B - Move backwards on character
 - Ctrl-D - End of file when providing program input. When pressed on the first position of the command line, ends the connection (similar to running `exit`).
 - Ctrl-R - Search through the history of commands
 - Ctrl-K - Cut the text from the current position to the end of the line
 - Ctrl-Y - Paste what was cut with Ctrl-K

4. Commands and paths:

A command is a program, any program

We will only use commands that work in the console, meaning the interface is exclusively text

To run a command, type its name followed by whatever arguments are necessary, everything separated by space.

Commands:

- o list the content of the current directory run `ls`
- To see the content of file `/etc/passwd`, use `cat /etc/passwd`. Here, `/etc/passwd` is an argument
- To see the content of the current directory, run `ls -l`. Here, `-l` is an argument too
- To see the content of the current directory, with all the details, and including the hidden files, run `ls -l -a` or `ls -l --all`. Here, `-a` and `--all` have the same effect, one being the short form, and the other being the long-form.
- To create a directory name `abc`, run `mkdir abc`
- To display the content of the current directory, with all the details but without the annoying colors, run `ls -l --color=never`. Here, `--color=never` is an argument with value.

Sometimes, the equal sign is not necessary, but always consult the manual (command `man`) or the `--help` option (eg `ls --help`)

- To do the same thing above, for the directory `/etc`, run `ls -l --color=never /etc`

Paths:

- UNIX file system has a single root, unlike Windows which has a root for every drive mounted (ie C:, D:, etc)
- The UNIX file system root is `/`, and all drives are mounted as directories, somewhere in the file system
- The UNIX file separator is `/`, unlike Windows, where the separator is `\`
- Every user has a home directory, which is the current directory when you connect over SSH. Run command `pwd` to find the path to your current directory.

Go through the ideas above explaining the command structure:

- Space is separator
- First word is the command
- Next words are arguments
 - Values: `ls /etc`
 - Options
 - Short form: `ls -l`
 - Long form: `ls --all`
 - Short form with value: `cut -d : -f 1,2,3 /etc/passwd`
 - Long form with value: `cut --delimiter=: --fields=1,2,3 /etc/passwd`
 - Combined short forms: `ps -e -f` is equivalent with `ps -ef`

5. Regular expressions:

Very weird and ugly, but compact and flexible language for matching text.

Why use them?

- How can you find all the lines of a text file that contains phone numbers?
- How can you remove all the extra spaces at the end of each line of a text file?
- How can you remove any duplicated spaces from a text file?
- If you ask a user to input an email in a text field, how do you verify that they input something that at least looks like an email?

Regular expression rules

- Every character that appears in a regular expression can have two meanings, normal or special, depending on the escape character `\` appearing in front of it.
- Depending on the program used to process the regular expressions, a character's special meaning is achieved with or without escaping it. Below are the meanings as required by the programs we will use for this class.

.	Matches any single character
\	Escape, changes the meaning of the character following it, between normal and special
[abc]	Matches any single character that appears in the list (in this case a or b or c)

[a-z]	Matches any single character that belongs to the range (in this case any lower-case letter)
[^0-9]	Matches any single character that does not belong to the range (in this case anything that is not a digit)
^	Beginning of line
\$	End of line
\<	Beginning of word
\>	End of word
()	Group several characters into an expression
*	Previous expression zero or more times
+	Previous expression one or more times
?	Previous expression zero or one times
{m,n}	Previous expression at least m and at most n times
	Logical OR between parts of the regular expression

Examples:

- .* - any sequence of characters
- [a-zA-Z02468] - any letter, regardless of its case, and any even digit
- [,] - space or comma
- ^[^0-9]+\$ - non-empty lines containing any characters except digits
- ([Nn][Oo])⁺ - any refusal, no matter how insistent (eg No no no no no)

1. Grep:

- Program that searches through files using regular expressions
- Why is it called `grep` and not something more intuitive? It was very intuitive to those who implemented `grep`, but not so much to us. Google "why is grep called grep" and enjoy. While at it, go learn about `ed` (the original UNIX editor). It is available in your Linux installation. Have Google handy if you play with it ...
- Option arguments that we will use a lot
 - `-E` - use extended regular expressions (POSIX ERE)
 - `-v` - display lines that do not match the given regular expression
 - `-i` - ignore upper/lower case when matching (match case-insensitively)
 - `-q` - Do not display the matching lines, just exit with 0 if found, or 1 if not found. We will use this only later, when we get to Shell Programming.
- File `/etc/passwd` contains all the users in the system, providing their usernames, full names, home directories, etc
 - We will use this file a lot to exercise text matching
 - It is structured as follows, using `:` as field separator

```
username:password:user-id:group-id:user-info:home-directory:shell
```

1

- The password field will always be x, the actual encrypted password residing in file /etc/shadow which cannot be read by regular users
- The user-info field, usually contains the full name of the user

1. Let's search for things in file /etc/passwd:

- a. Display all lines containing "dan". The solution is below
 - i. `grep -E "dan" /etc/passwd`
- b. Display the line of username "dan". The username is the first field on the line, it is not empty, and it ends at the first :. We will rely on these aspects to ensure that we only search the usernames, and not anything else
 - i. `grep -E -i "^dan:" /etc/passwd`
- c. Display the lines of all users who do not have digits in their username.
 - i. `grep -E "^[^0-9]:" /etc/passwd`
- d. Display the lines of all users who have at least two vowels in their username. This is a little tricky, because the vowels do not need to be consecutive, so we need to allow for any characters between the vowels (including none), but we cannot allow : to appear between vowels, or else we would be searching outside the username.
 - i. `grep -E -i "^[^:]*[aeiou][^:]*[aeiou][^:]*:" /etc/passwd`
 - ii. `grep -E -i "^[^:]*([aeiou][^:]*){2,}:" /etc/passwd`
- e. There will be lots of users displayed for the problem above, so let's search for usernames with at least 5 vowels in their username. The first solution above will be really long for this case, but the second will be very easy to adapt, by changing 2 into 5.
 - i. `grep -E -i "^[^:]*([aeiou][^:]*){5,}:" /etc/passwd`
- f. Display the lines of all the users not having bash as their shell. The shell is the last value on the line, so we will use that when searching.
 - i. `grep -E -v "/bash$" /etc/passwd`
- g. Display the lines of all users named Ion. We will have to search in the user-info field (the fifth field) of each line, ignore the upper/lower case of the letters, and ensure that we do not display anybody containing the sequence "ion" in their names (eg Simion, Simionescu, or Ionescu).
 - i. `grep -E -i "^[^:]*{4}[^:]*\<ion\>" /etc/passwd`

2. Let's consider a random text file a.txt, and search for things in it:

- a. Display all the non-empty lines
 - i. `grep -E "." a.txt`
- b. Display all the empty lines
 - i. `grep -E "^$" a.txt`
- c. Display all lines containing an odd number of characters
 - i. `grep -E "^(..)*.$" a.txt`
- d. Display all lines containing an ocean name

```
grep -E -i "\<atlantic\>|\<pacific\>|\<indian\>|\<arctic\>|\<antarctic\>"
a.txt
```

- e. Display all lines containing an email address
 - i. What does an email address look like? It has the following structure.
 1. username - let's assume it can contain any character, except for @, *, !, and ?
 2. @ - separator between the username and the hostname
 3. hostname:
 - a. Sequence of at least two elements separated by .

- b. Let's assume an element can contain any letter, digit, dash, or underscore

- ii. `grep -E -i "<[^*!\?]+@[a-z0-9_-]+(\.[a-z0-9_-]+)+>" a.txt`

2. Sed:

- Program for searching processing text by performing search/replace, transliterations, line deletion, etc
- By default, it does not modify the file, but displays the result of processing the input file
- We will use only the features presented below
 - a. Search/replace:
 - i. Command structure: `sed -E "s/regex/replacement/flags" a.txt`
 - ii. `s` - is the search/replace command
 - iii. `/` is the separator and can be any other character. The first character after the command `s` is considered to be the separator
 - iv. The flags at the end can be `g`, `i`, or both
 - 1. `g` - Perform the replacement everywhere on the line. Without it, only the first appearance will be replaced
 - 2. `i` - Perform a case-insensitive search
 - v. The replacement can contain reference to the expressions grouped in the regex as `\1`, `\2`, etc, the number being the order in which the groups appear in the regex
 - b. Transliterate
 - i. Command structure: `sed -E "y/characters/replacement/" a.txt`
 - ii. `y` - is the transliteration command
 - iii. `/` is the separator and can be any other character. The first character after the command `y` is considered to be the separator
 - iv. The `characters` and the `replacement` must have the same length
 - c. Delete lines matching a regular expression
 - i. Command structure: `sed -E "/regex/d" a.txt`
 - ii. `/` is the separator
 - iii. `d` - is the line deletion command

Let's manipulate the content of `/etc/passwd`

- a. Display all lines, replacing all vowels with spaces
 - i. `sed -E "s/[aeiou]/ /gi" /etc/passwd`
- b. Display all lines, converting all vowels to upper case
 - i. `sed -E "y/aeiou/AEIOU/" /etc/passwd`
- c. Display all lines, deleting those containing numbers of five or more digits:
 - i. `sed -E "[0-9]{5,}/d" /etc/passwd`
- d. Display all lines, swapping all pairs of letters
 - i. `sed -E "s/([a-z])([a-z])/\2\1/gi" /etc/passwd`
- e. Display all lines, duplicating all vowels
 - i. `sed -E "s/([aeiou])/\1\1/gi" /etc/passwd`

3. Awk:

- Given a separator character (by default it is space), treats the input text as a table, with each line being a row, and the fields of each row the tokens of the line, as determined by the separator.
- Processes the input based on a program written in a simple C-like language
- A program is a sequence of instruction blocks, prefixed by an optional selector
- Each block in the program is applied to every line of input matching its selector. If the block does not have a selector, it is applied to every line of input
- A selector is any valid conditional expression, or one of the following two special selectors
 - a. BEGIN - the block associated with this selector is executed before any input has been processed
 - b. END - the block associated with this selector is executed after all input has been processed
- Special variables
 - a. NR - number of the current line of input
 - b. NF - the number of fields on the current line
 - c. \$0 - the entire input line
 - d. \$1, \$2, ... - the fields of the current line
- The AWK program can be written in a file, or provided directly on the command line between apostrophes

Manipulate the content of /etc/passwd, using AWK with the program provided on the command line

- a. Display all the usernames, but only the usernames, and nothing else. We will use argument `-F` to tell AWK that the input file is separated by `:`, and then we will print the first field of each line, by not providing any selector for the block.
 - i. `awk -F: '{print $1}' /etc/passwd`
- b. Print the full name (the user info field) of the users on odd lines
 - i. `awk -F: 'NR % 2 == 1 {print $5}' /etc/passwd`
- c. Print the home directory of users having their usernames start with a vowel
 - i. `awk -F: '/^[aeiouAEIOU]/ {print $6}' /etc/passwd`
- d. Print the full name of users having even user ids
 - i. `awk -F: '$3 % 2 == 0 {print $5}' /etc/passwd`
- e. Display the username of all users having their last field end with "nologin"
 - i. `awk -F: '$NF ~ /nologin$/ {print $1}' /etc/passwd`
- f. Display the full names of all users having their username longer than 10 characters
 - i. `awk -F: 'length($1) > 10 {print $5}' /etc/passwd`

Keep using /etc/passwd as input file, but provide AWK programs in a file. The command will look like

```
awk -F: -f prog.awk /etc/passwd
```

- a. Provide the content of file `prog.awk` so that the command above will print all user on even line having a group id less than 20


```
NR % 2 == 0 && $4 < 20 {
    print $5
}
```
- b. Display the sum of all user ids


```
BEGIN {
```

```
        sum=0
    }
    {
        sum += $3
    }
END {
    print sum
}
```

c. Display the product of the differences between the user id and the group id

```
BEGIN {
    prod=1
}
{
    prod *= $3-$4
}
END {
    print prod
}
```