COSC 264 Problem Set
Unix is Your Friend!

Andreas Willig
andreas.willig@canterbury.ac.nz

July 7, 2021

---

1 Introduction

The goal of the first week in the lab is that you acquire basic skills for working with the Unix / Linux command line and for using basic text processing tools. You will learn about a small set of Unix/Linux command line tools like `grep`, `sed` and `awk`, both in isolation and chained together by piping. These tools operate on **text files**, i.e. on files which mainly use printable characters of the ASCII character set. You will perform the lab work entirely under the Linux operating system on the lab computers. After you have logged in, you should open a terminal/shell. Important: all of the following assumes that you work with the `bash` shell. If you have never changed any settings in your Linux account, you should be fine.

In this lab sheet you will find a range of problems. Many of these problems just involve reading `man` pages or similar documentation, some practical problems require you to invent smallish command line incantations or to even write an `awk` script. It is important that you work through the reading problems **before** the actual lab, so that you can use the lab time itself to work on the practical problems.

Important: you should not expect that the notes below will contain *everything* you need to know to solve the problems on this and on later sheets – they barely scratch the surface and should just provide you with basic ideas and starting points. It is expected that you read and work a lot on your own!

You might ask yourself: why do we have to deal with this Unix/Linux stuff anyway? There are two general answers:

- If you happen to work in the fields of system administration, network design and management, or the deployment and maintenance of web / database / application servers, it is highly likely that you will get in touch with Unix/Linux boxes. For example, many websites are hosted in data centers made up of a huge number of

blade servers. The primary way of interacting with a blade server is to log remotely onto it (using for example `ssh`) and to do your work on a Unix command line – after all, a blade server does typically not have a display and graphics card, nor does it have any software that runs graphical user interfaces.

- I believe that every computer scientist or computer / software engineer should know at least a bit of Unix, not only because it is an excellent operating system in itself, but also to show you that there is more than just Windows or Mac OS X (which under the hood actually is a Unix).

A more specific answer is that the things discussed on this problem sheet come in handy on other problem sheets.

2 BASH command line editing

After you have started a shell / terminal, it allows you to enter commands, which start one or more programs and which might also print results or error messages. When the started program(s) have finished, the shell is again ready to accept commands.

For a novice the `bash` shell with its fairly spartan user interface looks a bit intimidating. However, it has quite wonderful support for entering commands and their parameters. One of the most helpful mechanisms of the `bash` shell is the **tab completion** mechanism. For example, in your Linux box there exists a file named `/usr/share/doc/bash/README`.[1] Suppose we want to display the contents of this file in the terminal. We could achieve this by entering the command

```
more /usr/share/doc/bash/README
```

into the command line in one go, followed by pressing ENTER. Instead, we do the following:

- Start in a new line in the shell.
- Enter `more` followed by a blank
- Enter `/u` (nothing more!) and then press the TAB key. The shell expands this into the string `/usr/`. By pressing the TAB key you instruct the shell to try to complete the given prefix `/u`, assuming this is uniquely possible and that it is part of a filename (i.e. the shell searches the contents of the filesystem to find a completion). In this case it is unique.
- Next press `sh` and press TAB again, you should now have `more /usr/share/` on your screen, followed by the cursor.

[1] This name actually refers to a file named `README` in the directory `/usr/share/doc/bash/`. In Linux directories are hierarchically organised, which in this case for example means that `bash` in turn is an entry in the directory `/usr/share/doc/` etc.

- Now press `d` and enter TAB (you maybe have to press TAB twice). There are several possible candidate files/directories starting with `d` in directory `/usr/share/`, so the `bash` displays all of them.
- Now press `o` and TAB again, the number of alternatives has reduced.
- Now press `c/bash/R` and then TAB. Now the expansion is unique and you will notice that now there is a blank between the last character of the filename and the cursor.
- You can press ENTER to submit the command.

This tab completion mechanism is a blessing that Unix shells have since decades. Another very helpful mechanism is the **command history**. After you have entered some commands, just try the cursor-up and cursor-down keys and see what happens. Furthermore you can enter `Ctrl-R` and some letters of a previous command to completely recall the previous command.

It is also good to know that tab completion works not only for filename arguments (as shown above), but also for command names itself: go into a new line of the shell, enter `do` and press the TAB button (you need to press the TAB button twice). You are listed a number of commands starting with `do`.

The precise operation of tab completion can be configured, and generally it depends on systemwide defaults. Sometimes these defaults do not implement tab completion for commands. If the TAB key does not work as you expect, you can try the Control-D key (after you have entered at least one character of the command).

To leave the shell, use Control-D. From within a shell you can start a further shell by submitting the command `bash` (followed by pressing ENTER). You will enter a new bash shell with an empty command history. If you enter Control-D in this new shell, you will be back to the shell where you started.

3 Important Unix Command Line Tools

It is part of the tradition of Unix to offer a large set of programs, each of which does only one thing very comprehensively (i.e. with lots of options), and to allow users to chain them together on the command line using pipes. We will briefly go over a few relevant tools and examples below.

A very important command to know about Unix/Linux is the `man` command (related to the word *manual*). By typing

```
man grep
```

in the command line, you get (after pressing ENTER) the manual page for the `grep` command. Generally, you use

```
man xxx
```

to learn something about the command `xxx`. It is important to note that the primary intention of a manual page is *not* to be a tutorial, it is usually just a rather terse summary of what a program does, what its major command line options are, etc.

As a general rule, whenever you want to ask a tutor something related to a Unix command, please check the `man` page first and try to solve your question. When your tutor is convinced that your question can be solved by studying the `man` page, he/she will just refer you to it.

Problem 3.1 (Review problem)**.**

- Use the `man` command to learn about the following fundamental commands in Unix/Linux:
  - `ls`
  - `cd`

Solution 3.1.

3.1 Regular Expressions and `GREP`

The first important tool to learn about is the `grep` tool. It basically allows the user to specify a text pattern and to print all lines in a file that match the given pattern. The text patterns are specified as *regular expressions*. Examples:

```
grep hello file.txt
```

prints all lines of file `file.txt` which contain the sub-string `hello`. With

```
grep "hel*o" file.txt
```

you find all lines containing `helo`, `hello` `helllo` and so on. In this case it was necessary to enclose the search pattern with ¨ characters. This ensures that the shell hands over the entire string `hel*o` to the grep program. If the ¨ characters were left away, the shell itself would have interpreted the `*` character and it would have expanded this with the names of all files in the current directory named `helo`, `hello` and so on (assuming they exist). After doing this expansion (accidentally) the shell would execute the command

```
grep helo hello helllo file.txt
```

which means that `grep` is instructed to search for the string `helo` in the files `hello`, `helllo` and `file.txt`. Clearly, this is not what we wanted.

Problem 3.2 (Review problem)**.**

- Review the notion and usage of regular expressions.
- Read the man page of the `grep` command and the `egrep` command. Specifically, check out the following options:
  - `-i`
  - `-v`
  - `-A`
  - `-B`
- How differs the `egrep` command from `grep`?
- How would you remove all lines which include sequences of the character `a` that are at least three but at most five characters long? Or, as an extended example, sequences which either have three-to-five characters long sequences of the letter `a` or four-to-six character long sequences of the letter `b`? Write a `grep` or `egrep` invocation and a test file.

Solution 3.2. Only the answer to the last problem is given. One way to achieve this is to use the invocation `grep "aaa" testfile | grep -v "aaaaaa"` or better (i.e. without piping) is `egrep -v -w a{3,5} testfile`. The sign separating the expressions in brackets is the vertical bar.

3.2 The stream editor `SED`

Basically, the `sed` tool allows you to modify those parts of a file that match a regular expression (however, the file itself is not changed – instead, the modified file is sent to `stdout`, see below). You have to specify the regular expression and the text that is substituted. `sed` allows for fairly sophisticated specification of patterns and substitutions, we will stick to the easiest things.[2] For example:

```
sed -e "s/hello/byebye/" filename
```

goes through file `filename` line-by-line and replaces *the first* occurence of string `hello` in a line by `byebye`. The `s` character identifies our intention to **substitute**. If you want to replace *all* occurences of `hello` in a line, you have to say

```
sed -e "s/hello/byebye/g" filename
```

[2] An excellent and very comprehensive tutorial can be found here: `http://www.grymoire.com/Unix/Sed.html`

i.e. you have to supply the `g` (for "global") flag. The command-line parameter `-e` in the sed invocation means that the actual "sed script" (i.e. the specification of the replacement operations) is given in the command line as the next parameter (as above). With the `-f` parameter the sed script is taken from a file.

Problem 3.3 (Review problem)**.**

- Read the man page of the `sed` command.
- Write a `sed` invocation that replaces all occurences of a string of the form `prefix-xxx-suffix.cfg` by a string of the form `suffix-xxx-prefix.cfg` where `xxx` is any three-digit number. Of course, in the replacement string the same three-digit number should show up as in the original string. Create an example text file to test your solution. Advanced: reverse the three digits, e.g. if a line contains `prefix-123-suffix.cfg` then the output should be `suffix-321-prefix.cfg`.

Solution 3.3. The key trick is to store parts of the pattern using the `\(` and `\)` operator. So the invocation which solves both problems (with minor adaptation) becomes: `sed -e "s/prefix-\([0-9]\)\([0-9]\)\([0-9]\)-suffix.cfg/suffix-\3\2\1-prefix.cfg/g"`
Not pretty, but it works.

---

4 The `vi` editor, `cat` and `more`

Sometimes you will have to edit text files like for example configuration files. On many small Linux-based boxes there is only the really essential software installed and fluffy colourful editors with many bells and whistles are not considered essential. However, there is one editor which is almost universally available on all Linux-/Unix-type boxes nowadays, the `vi`. `vi` stands for "visual" but you should not take this term too serious.[3]

The `vi` editor is rather powerful, but you have to get used to it, since its way of operation (let alone the key bindings) differs very much from everything you might have worked with so far under Windows. There are plenty of `vi` tutorials available on the web. You do not need to know much about `vi`, but the following exercise asks you to figure out basic operations.

[3] However, `vi` is much more "visual" than the editor that is really guaranteed to be available on absolutely every Unix system, the `ed`. See also `http://www.gnu.org/fun/jokes/ed.msg.html`.

Problem 4.1 (Review problem)**.**

- How to start and leave `vi`?
- What are the two modes of `vi` and how do you switch between them?
- How to move with the cursor?
- How to save a file?
- How to do regex-based search for text, and how to do regex-based search-and-replace?
- How to delete the character under cursor? How to delete a line? How to delete 377 lines? How can you repeat commands a given number of times?

Solution 4.1.

- Start: `vi filename.txt` on the command line, to leave `vi`, press `esc` and `:q` for quitting. When the file has been changed, `vi` won't let you quit. Then press either either `:wq` for writing the file and quitting, or `:q!` for quitting without writing.
- Insertion mode and command mode. Pressing `esc` key always brings you in command mode. When you are in command mode you can press `i` (for insert) or `a` (for append) to start entering text.
- Enter command mode, use the cursor keys
- See above, or: enter command mode, press `:w`
- Search: enter command mode, press `/` followed by regexp, then enter. Replace: enter command mode, press `:%s/original/replacement/` followed by enter to replace the original text `original` by `replacement`. If you add a `g` at the end, you can do global replacement
- Character under cursor: switch to command mode, enter `x`. Current line: switch to command mode, enter `dd`.

Sometimes you do not want to edit a file but just view it. A first suitable tool is the `cat` command, which is used as follows:

```
cat filename.txt
```

This command prints the entire contents of `filename.txt` to its standard output, which, when called from the shell, is the screen itself. When the contents of the file is longer than the number of lines available on the screen, you can use the `more` command, invoked as follows:

```
more filename.txt
```

which displays the first screenful of lines on the screen[4], waits until the user presses a key (perhaps the space key), then displays the next screenful and so on. Additionally, by entering the / key, immediately followed by a search string and enter, you can do a regex-based search within more.

Problem 4.2 (Review problem)**.**

- Read the man page of more, especially how to use the search facility.

Solution 4.2.

4.1 The Swiss-Army Knife of Stream Processing: AWK

The awk tool is actually a fully-fledged programming language which, according to unconfirmed legends, some ancient heroes even have used to write compilers.

In this section I have drawn on the online awk tutorial found under http://www.grymoire.com/Unix/Awk.html prepared by Bruce Barnett. We will not need much of the awk language.

The real strength of the awk tool is to process ASCII files line by line.[5] It can apply regex-based filters and substitutions, it can update variables, it can print intermediate results and even do calculations on the contents of a file (after interpreting and parsing tokens from the file as numbers). Basically, an awk script is a sequence of rules, one rule has the form

```
pattern { action }
```

where the curly braces have to be typed in as shown here. The pattern specifies a condition that a line must satisfy to perform the action on it. Typical choices for pattern are the blank or null pattern (which matches *every* line), regular expressions and certain special patterns like BEGIN (executed before the first line of input has been read) and END (executed after the last line of input has been read).

Instead of discussing too much of awk's syntax here (you find all relevant information in the tutorial referenced above) I will discuss two example programs. Open an editor,[6] enter the following program and save it as example1.awk:

[4] Note that more outputs directly to the screen, not to standard output.

[5] This can also be done with Perl or Python, but only awk can be assumed to be present on any Linux/Unix box, even the smallest ones that serve as routers somewhere.

[6] You can use vi, but in the virtual machines you will be working with later in this course, the other editor is also installed, the emacs editor. emacs is much more powerful than vi but has again its own very unique ideas about keybindings.

```
#!/usr/bin/awk -f
BEGIN {cnt=0}
/hello/ {cnt = cnt+1; print}
END {print "found", cnt, "lines with hello."}
```

(the first line of this file is not related to `awk` but is a so-called she-bang pattern, followed by the path to the `awk` executable. Such a she-bang pattern instructs the shell to run the indicated program and hand over the remaining contents of the script to it – this explanation is not entirely correct but serves the purpose. The `#` character starts a comment in `awk`). Once you have saved the file, run the command

```
chmod +x example1.awk
```

to make the file executable. To actually run the script you have to use the command

```
./example1.awk filename
```

The `BEGIN` pattern of the first example initializes the variable `cnt` to zero. The second pattern is applied to all lines which match the regular expression `hello` (i.e. which contain the sub-string `hello` at least once). If such a line is found, first the variable `cnt` is incremented, and then the entire line is printed – the call to `print` without further parameters prints the entire line. Statements have to be separated by semicola. In the final `END` pattern the final value of the `cnt` variable is printed.

The second example is much simpler:

```
#!/usr/bin/awk -f
{print $3}
```

`awk` views a line as being made up of different tokens or columns, which are separated by one or more subsequent whitespace characters (blanks, tabs). You can assess the tokens of a line through pseudo-variables, e.g. `$1` for the contents of the first column, `$2` for the contents of the second column and so forth. You can even do something like

```
#!/usr/bin/awk -f
BEGIN {x=4}
{print $x}
```

i.e. you can use a variable to control which column is printed.

Now create a testfile with the following data, save it as `test.data`.

```
type-a     10    20
type-b     xx    yy     zzz
type-c
type-a     10    30
type-a     10    40
type-a     10
type-a     10    45.33
```

Problem 4.3 (Review problem)**.**

- Write an `awk` script that computes the average value of the third column of all lines starting with `type-a`.
- Write an `awk` script which assumes its input to consist of lines with only one column, which all are numbers, and which computes the average value of these numbers.
- Use the solution to the second question as a component in a command-line invocation involving pipes (see below), `grep`, etc., that fullfils the same purpose as the solution to the first question. You can attempt this problem after you have worked through Section 5.

Solution 4.3. The following listing addresses only the second problem, its adaptation to the first is easy. If this script is called `average.awk` (and has been made executable) then a suitable invocation could be `grep "type-a" test.data | awk '{print $3}' | ./average.awk`

```
1 #!/usr/bin/awk -f
2 BEGIN {cnt = 0; sum = 0}
3 {cnt = cnt+1 ; sum = sum + $1;}
4 END {if (cnt > 0) {print sum / cnt}}
```

A final comment: in the examples above we have stored the scripts in a file (together with the she-bang line) and invoked that file directly from the command line. Another approach is to invoke the `awk` command with the `-e` option, so that the script can be given directly on the command line. The `man` page for `awk` has to say more on this topic.

5 Chaining Commands together through Pipes

The individual tools like `grep`, `sed` etc. usually have a rather limited functionality, but Unix / Linux allows to glue them together on the command line using **piping**. One pre-requisite of piping is that most of the Unix commands are able to receive the input file in two different ways:

- The first way is to supply the command with a filename on the command line, like e.g. in the incantation

  ```
  grep hello file.data
  ```

  The shell passes the filename `file.data` to the `grep` command, which then opens the file and processes it line by line.

- The second way is via **standard input** (abbreviated: `stdin`), which is a kind of implicit file that has already been opened by the shell and which the command then can read from. As a convention, when no filename is supplied to a command, the command reads its input from `stdin`. It is important to note that this convention is also true for our `awk` scripts given above.

Most of the Unix commands write all their output to the **standard output** channel (abbreviated: `stdout`), which for simple command invocations (i.e. without piping) is terminated by the shell itself, which collects all data from standard output and prints it.

The key to glueing commands in unix is the `|` symbol, known as the **pipe** symbol. One example of its usage is as follows:

```
cmd1 test.data | cmd2
```

Here, the command `cmd1` is called with a filename. `cmd1` reads the filename, opens the file and processes it. Most importantly, `cmd1` writes all its output to its `stdout` channel. The pipe operator `|` now connects the `stdout` of `cmd1` to `stdin` of `cmd2`. The command `cmd2` reads all its input from its `stdin` (and thus it receives everything that `cmd1` has written to its `stdout`), processes it and prints it to its own instance of `stdout`. Finally, the shell receives the data written to `cmd2`'s `stdout` and prints it.

Problem 5.1 (Review problem)**.**

- Read the man page for the `wc` command.
- Implement the same functionality as the first `awk` script above (`example1.awk`) without using `awk`.
- Independent of this, consult also the man pages for the `head` and `tail` commands

---

Solution 5.1. I only give the solution for the second problem. This is simply the invocation `grep 'hello' | wc` when the goal is just to print the number of lines containing "hello". If exactly the same output as the above `awk` script should be achieved, then the invocation is more involved (and uses facilities that we have not discussed yet), it becomes `echo "found ‘grep 'hello' testfile | wc | awk '{print $1}'‘ lines with hello"`. Please note the difference between normal quote characters (') and the backquote (‘). What is happening here?
