CSCI 2132: Software Development

Lab 2: Unix Utilities and Emacs

Synopsis

In this lab, you will:

- Review remote login to `bluenose` using `ssh`
- Review basic use of Subversion (SVN)
- Learn to work with autocompletion in the shell
- Improve your `emacs` skills
- Learn about using some Unix utilities
- Learn to use Unix pipes

Contents

Overview

Many Unix shells offer an autocompletion feature. `bash` is no different. You will learn to use this feature to reduce the amount of typing required to work with the shell. You are also encouraged to learn about advanced autocompletion features available in `bash` and other shells, such as `zsh`, not covered in this lab or in class. You will learn more about using `emacs`. In particular, you will learn about some hotkeys for basic editing operations. Finally, you will learn about using three Unix filters, `uniq`, `sort`, and `cut` and combine them using Unix pipes to perform some simple file transformations.

Step 1: Log in to `bluenose`

Review your notes from Lab 1 and follow the same steps as in Lab 1 to log into `bluenose`, either using `PuTTY` on Windows or using `ssh` on Linux or macOS.

After logging in, change to your SVN directory prepared in Lab 1, `~/csci2132/svn/CSID`, where `CSID` is your CSID.

Check that you are indeed in the right directory:

```
$ pwd
/users/cs/CSID/csci2132/svn/CSID
```

Step 2: Prepare a directory `lab2` for the current lab

Create the `lab2` directory:

```
$ mkdir lab2
```

Add the `lab2` directory to SVN and submit it. Review your notes from Lab 1 on how to to do this using the `svn add` and `svn commit` commands. Recall that `svn commit` requires a log message. You may use something like

```
$ svn commit -m"Directory lab2 created"
```

Step 3: Verify that you successfully submitted your changes

It is a good idea to check that you submitted your files successfully in the previous step. The simplest way to do this is by running

```
$ svn status
```

You should see no output if all files in your working copy have been added to SVN and have no changes in them that have not been committed yet.

To see the list of all files that SVN knows about and their status, run

```
$ svn status -v
             11806    11805 CSID    .
             11806    11805 CSID    lab1
             11806    11805 CSID    lab1/HelloWorld.java
             11806    11805 CSID    lab1/HiWorld.java
             11806    11805 CSID    lab2
```

where `CSID` is your CSID. The numbers may be different. What you want to see to be sure that all your work has been submitted successfully is:

- There should be only spaces before the two leading numbers (there are no changes that were not submitted).
- Every file that you wanted to submit should have two numbers in front of it. (These are revision numbers associated with these files. Every file that is known to SVN has these associated revision numbers. Missing revision numbers (and a ? at the beginning of the line) means you have not added the file to SVN yet.)

Another way to check what has been committed is to run

```
$ svn log -v | less
```

You should see an output similar to the following:

```
------------------------------------------------------------------------
r11805 | CSID | 2018-09-12 16:44:00 -0300 (Wed, 12 Sep 2018) | 1 line
Changed paths:
    A /CSID/lab1
    A /CSID/lab1/HelloWorld.java
    A /CSID/lab1/HiWorld.java
------------------------------------------------------------------------
```

There may be multiple entries separated by dashed lines. The important information you get is:

- The revision number of the commit (`r11805` in this example).

- The user who committed the changes in this revision.
- The date when this revision was created (the changes were submitted).
- The list of files that were changed. (The `A` means that the file was added. `M` and `D` denote modifications and deletions.)

Step 4: Autocompletion

Autocompletion, available in virtually every modern shell, is a handy feature that can reduce the amount of typing required by completing commands and file names you have begun typing. To try this, follow the following instructions to copy `~/csci2132/svn/CSID/lab1/HelloWorld.java` to the `lab2` directory:

Start by typing

```
$ cp ../lab1/Hel_
```

The underscore (_) indicates the position of the cursor. Now press `Tab`. If `HelloWorld.java` is the only file in the `lab1` directory whose name starts with `Hel`, the shell completes the filename on the command line:

```
$ cp ../lab1/HelloWorld.java_
```

Complete the command by entering `.` and then pressing `Enter`. The full command is

```
$ cp ../lab1/HelloWorld.java .
```

If there is another file in the `lab1` directory whose name starts with `Hel`, say `HelloWorld.class`, then the shell only completes the filename up to the first character where the two filenames differ. That is, you would see something like

```
$ cp ../lab1/HelloWorld._
```

Enter the next character needed to distinguish which file you mean:

```
$ cp ../lab1/HelloWorld.j_
```

and press `Tab` again to complete the filename

```
$ cp ../lab1/HelloWorld.java_
```

As above, complete the command to

```
$ cp ../lab1/HelloWorld.java .
```

and press `Enter`.

Clearly, this is a very handy feature. Autocompletion works for any filename, including directories you want to navigate to, and for command names.

Autocompletion on steroids

To pique your curiosity, here is something that my `zsh` setup can do:

```
$ cd D/R/P/T/C/R
```

I press `Tab` and it expands to

```
$ cd Documents/Research/Papers/TreeChildHybridization/Code/Rust
```

based on the directories in my home directory. Neither `bash` nor `zsh` supports this out of the box, but the right configuration options make `zsh` behave this way and I would not be surprised if `bash` can be convinced to do the same. If you think this is cool, figure out how to set up `bash` to support much more powerful autocompletion than it supports out of the box. (Google is your friend.)

Step 5: Moving around in emacs

One of the first things you want to become proficient at is moving around a file you opened in `emacs`. The commands discussed here are available in some form in simpler editors, too, but already here, `emacs` proves to be more efficient once you learn these hotkeys. The reason is that moving your hand away from its natural typing position to access the cursor keys or the `Home` or `End` keys is much more disruptive than pressing `Control` and a regular letter key.

To move the cursor left, right, up or down in `emacs`, use `C-b` (back), `C-f` (forward), `C-p` (previous line) or `C-n` (next line). (Cursor keys work, too, but I encourage you to practice these hotkeys.)

`M-b` and `M-f` move one word backward or forward instead of one character backward or forward.

Moving one screen forward or backward can be achieved using `C-v` or `M-v`.

`C-a` or `C-e` moves to the beginning or end of the current line.

Practice using these hotkeys by opening `HelloWorld.java`. (Do so by typing `em` `Tab` and then `H` `Tab` on the command line.) Once you are done editing the file, save it using `C-x` `C-s` and exit `emacs` using `C-x` `C-c`.

Step 6: Cut and paste whole lines in `emacs`

The next skill you will acquire is using cut-and-paste in `emacs`.

Use `emacs` to edit a file named "`names`" in the directory `lab2`. Type the following line in `emacs` and press `Enter`:

```
Alan Turing
_
```

The underscore (_) shows the position of the cursor. Now move up one line, press `C-k` `C-k` and then `C-y` `C-y` twice. Your file should now look like this:

```
Alan Turing
Alan Turing
_
```

You should have noticed that `C-k` deletes the characters from the current cursor position to the end of the line. Pressing `C-k` again deletes the end-of-line character. Thus, if you press `C-k` `C-k` at the beginning of a line, you delete the line completely. Everything removed by a sequence of `C-k` presses is appended to the current slot in `emacs`'s *kill ring* (similar to the clipboard on Windows or macOS). Now the character combination `C-k` should make some sense: "kill a line". To paste what you killed at your current cursor position (after moving the cursor where you want to paste it), press `C-y` (which stands for "yank"). Pressing `C-y` `C-y` as above yanks the most recently killed content twice, so we end up with two lines containing "Alan Turing".

Step 7: Choose what to yank

emacs's kill ring is more powerful than your standard clipboard in that it remembers the history of what you killed. Let's try it out. First add another line to your file:

```
Alan Turing
Alan Turing
Ken Thompson
_
```

Move the cursor one line up and press `C-k` `C-k` `C-y` to kill the line and then put it back. Move to the beginning of the file and yank the line there. Then move back to the end of the file. Your file should now look like this:

```
Ken Thompson
Alan Turing
Alan Turing
Ken Thompson
_
```

Now here comes the fun part: Press `C-y` `M-y` `C-y`. Your file should now look like this:

```
Ken Thompson
Alan Turing
Alan Turing
Ken Thompson
Alan Turing
Alan Turing
_
```

`M-y` moves to the previous slot in your kill ring (your kill history) and pastes its content instead of the most recently killed content, "Alan Turing" in this case. You can press `M-y` more than once to move to even earlier entries in the kill ring. It is called a kill *ring* because

- The history is circular. Pressing `M-y` repeatedly eventually loops around to your most recently killed entry.
- The order of the entries is fixed and, after yanking a previous entry using `C-y` `M-y`, this entry becomes the current entry that will be inserted if we press `C-y` again. This is why we did not have to press `C-y` `M-y` `C-y` `M-y` above.

The three hot keys to remember so far are:

`C-k`	Kill to end of line
`C-y`	Yank
`M-y`	Yank previous entry in kill ring

Step 8: Cut and paste a marked region

Another way to cut and paste in emacs is to mark a region and then cut it using C-w. For example, we can produce the following file content

```
Ken Thompson
Alan Turing
Alan Byron
Ken Byron
Alan Turing
Alan Turing
_
```

from the file we have produced so far using the following sequence of operations:

- Move to the end of the third line of the file.
- Delete "Turing" using Backspace or C-h and then type "Byron".
- Your cursor should now be at the end of the third line.
- Set emacs's *mark* using C-Space or C-@.
- Move to the beginning of the word "Byron" and press C-w C-y. C-w kills the current *region* which is the text between the current cursor position and the mark, no matter whether the mark is before or after the current cursor position. C-y then puts the killed text back.
- Now go to the end of the fourth line, delete "Thompson" and yank "Byron" in its place again by pressing C-y.

For good measure, save your file (C-x C-s). emacs makes backup copies of your current editing session at regular intervals, but it is still a good idea to save your progress frequently in case of a power outage or similar interruption.

You have learned two more hotkeys:

C-Space or C-@	Set mark
C-w	Kill to mark

Step 9: Copy and paste in emacs

Always killing text only to yank it right back is tedious. When killing whole lines, this is the only thing you can do as there is no "copy line" hotkey. You *can*, however, copy a region to the kill ring without killing it. The key to do so is `M-w`.[1] Practice this by transforming the `names` file so it looks as follows:

```
Ken Thompson
Alan Turing
Alan Byron
Ken Byron
Alan Turing
Alan Turing
Ada Byron
_
```

- Move to the beginning of the fourth line.
- Set the mark (`C-Space` or `C-@`).
- Move to the end of the fourth line (`C-e`).
- Copy the marked region (`M-w`).
- Go to the end of the file (`M->`).
- Yank the copied text (`C-y`).

The file should now look like this:

```
Ken Thompson
Alan Turing
Alan Byron
Ken Byron
Alan Turing
Alan Turing
Ken Byron_
```

- Move to the beginning of the last line (`C-a`).
- Delete "Ken" (`C-d` `C-d` `C-d`).
- Type "Ada".
- Move to the end of the last line (`C-e`).
- Press `Enter`.

[1] You may begin to see a pattern here. In emacs, related commands often have the same letter key. For one command, the letter is combined with `Control`; for the other, with `Meta`. So `C-w` kills the region while `M-w` copies the region. `C-v` moves one page forward; `M-v` moves one page backward.

You should be finished:

```
Ken Thompson
Alan Turing
Alan Byron
Ken Byron
Alan Turing
Alan Turing
Ada Byron
_
```

Well, aren't I a sneaky fellow. I slipped in two additional hotkeys and will just add a third one for completeness:

`M-<` Jump to the beginning of the file
`M->` Jump to the end of the file
`C-d` Delete character forward (as opposed to delete character backward with `C-h` or `Backspace`)
`M-v` Copy region

Save the current file contents.

Step 10: Undo and redo in emacs

As in any good text editor, you can undo and redo your changes in emacs. The key to do so is C-_ or C-x u. As everything in emacs, this comes with a twist. You may notice that there is no separate redo key; C-_ or C-x u both serve as undo *and* redo keys. Let's see how this works. First open a new file "numbers" by pressing C-x C-f (find file). This presents a prompt at the bottom of the emacs window for a file to open. Enter numbers and press Enter. Now populate this file with a bunch of numbers:

```
1
2
3
4
5
_
```

At this point, your editing history looks something like this: "Enter 1, Enter 2, Enter 3, Enter 4, Enter 5". You decide you don't like these edits. Press C-_ repeatedly until the file looks like this:

```
1_
```

Now the magic starts. Perform a normal editing operation, say by pressing the key x:

```
1x_
```

At this point, emacs appends all the changes made by pressing C-_ to the editing history as if they were normal edits, not undo operations. Your editing history therefore looks something like this: "Enter 1, Enter 2, Enter 3, Enter 4, Enter 5, Delete 5, Delete 4, Delete 3, Delete 2, Enter x". If you now press C-_ repeatedly again to undo the undo operations themselves, you recreate the original file contents:

```
1
2
3
4
5
_
```

This behaviour must look strange in the extreme to most of you. It certainly did to me when I first learned about it. However, it comes in handy quite often. Consider your standard text editor with C-z to undo and C-y or S-C-z to redo. Repeatedly pressing C-z to undo some changes works fine. If you don't perform any edits, you can decide to redo the undone changes using C-y or S-C-z. However, if you perform a normal edit operation after undoing some changes, most editors completely forget the changes that were undone; there is no way to bring them back. It has happened to me more often than I care to remember that I undid some edits, typed some text, and then decided that I actually wanted to bring at least some of the undone edits back. emacs's notion of undo/redo allows you to do this. In pretty much any other editor, you need to recreate the lost edits by hand.

emacs hotkeys

Before moving on to the second part of this lab, here is a list of emacs hotkeys we discussed in this lab (including ones learned in Lab 1 and some additional ones for completeness):

Movement commands:

`C-p`	previous line
`C-n`	next line
`C-b`	previous character
`C-f`	next character
`M-b`	previous word
`M-f`	next word
`C-a`	beginning of line
`C-e`	end of line
`M-<`	beginning of file
`M->`	end of file

Delete, copy, kill, and yank:

`Backspace` or `C-h`	Delete character backward
`M-Backspace`	Delete word backward
`C-d`	Delete character forward
`M-d`	Delete word forward
`C-Space` or `C-@`	Set mark
`C-w`	Kill region
`M-w`	Copy region
`C-k`	Kill to end of line
`C-y`	Yank
`M-y`	Cycle through kill ring

Undo:

`C-_` or `C-x` `u`	Undo

Find and replace:

`C-s`	Incremental find (press repeatedly to find successive matches)
`M-%`	Find and replace

Open, save, and exit:

`C-x` `C-f`	Open (find) file
`C-x` `C-s`	Save file
`C-x` `C-w`	Save file as (Write file)
`C-x` `C-c`	Exit emacs

Step 11: `uniq`

A *filter* is a program that gets most of its data from `stdin` and writes its results to `stdout`. The concept of a filter is part of an important software architectural pattern called "pipe-filter". By connecting filters into a pipeline, we can create more complex filters.

The first filter we explore is `uniq`. It can be used to display a file with identical **adjacent** lines replaced by a single occurrence of this repeated line.

We start with the `names` file we created earlier.

Use `wc` to verify that this file has exactly 7 lines.

You can read about the `wc` command using `man wc`. The command `wc` stands for "word count". Without command line arguments, it prints the number of characters, words, and lines in a file. You can choose to print only one of these numbers using the command line options `-c`, `-w`, and `-l`.

The syntax of `uniq` is `uniq [inputfile [outputfile]]`. The square brackets mean that the parameters are optional. If these parameters are absent, `uniq` reads its input from `stdin` and writes its output to `stdout`. The nesting of brackets means that, if you supply only one name, it is used as the input.

Perform the following tasks:

(11a) Run `uniq names`. The output should contain 6 lines. Only the sixth line is not printed because it is identical to the fifth line.

(11b) Run `uniq -c names`. This precedes each output line with the number of adjacent occurences of this line in the input.

(11c) Run `uniq -f 1 names`. The output should be 5 lines long. Use `man uniq` to find out what `uniq -f 1 names` does. This command is equivalent to `uniq -1 names`, which is described on page 88 of the textbook.

Use standard output redirection to save the outputs of the three steps above into files `step11a.out`–`step11c.out`:

```
$ uniq names > step11a.out
$ uniq -c names > step11b.out
$ uniq -f 1 names > steb11c.out
```

Make sure that you understand what all of the above commands do before moving on to the next part of the lab.

Step 12: `sort`

The next tool we explore is `sort`, which allows you to sort the lines of a file according to a number of different criteria.

The basic usage is `sort [filename]`, which sorts the lines in the file `filename` in lexicographic order (the order in which they would appear in a dictionary).

Try this out to sort the `names` file you created before:

```
$ sort names
```

If given a filename as argument, `sort` sorts the lines in the given file and prints the result to `stdout`. If given no argument, `sort` reads its input from `stdin`, sorts its lines, and writes the sorted result to `stdout`.

The second possible use of `sort` is to sort the lines in the input file by certain fields contained in different lines. By default, fields are separated by one or more whitespaces. To sort by the nth key, you use `sort -k` n. Try

```
$ sort -k 2 names
```

to sort the entries in this file by last name or

```
$ sort -k 2 -k 1 names
```

to sort primarily by last name and secondarily by first name.

To sort by a range of fields $m,\ldots,n$, use `sort -k` m`,`n. The form `sort -k` m`,` can be used to use keys $m, m+1, \ldots$ up to the last field as sort keys.

Note that the textbook uses a different syntax for sorting by a particular field. This syntax is obsolete even tough it still works on `bluenose`.

To sort in reverse order, `sort` provides the command line option `-r`.

To practice different combinations of sorting, run

(12a) `sort names`
(12b) `sort -k 1 names`
(12c) `sort -k 2 names`
(12d) `sort -k 2 -k 1 names`
(12e) `sort -k 1,2 names`
(12f) `sort -k 2,2 -k 1,1 names`
(12g) `sort -r -k 2 names`

and use output redirection to save the results into files `step12a.out`–`step12g.out`.

Step 13: Sort CSV (comma-separated values)

Fields in a file can be separated by characters other than spaces. For example, Excel allows spreadsheets to be exported in plain text as comma-separated values, that is, each row of the spreadsheet corresponds to a row in the produced text file and the cells in each row are separated by commas. `sort` supports working with such inputs as well.

Create the following file:

names.csv

```
Alan,Turing
Alan,Turing
Ken,Thompson
Alan,Turing
Alan,Byron
Ken,Byron
Ada,Byron
```

Verify that this file has 7 lines. (You learned about a tool that can do this. Use it.)

To make the `sort` command work with comma-separated values, the option `-t` can be used to specify a field separator different from whitespaces.

Try the command

```
$ sort -t, -k 2 -k 1 names.csv
```

What does this command do?

Save the output of the command into the file `step13.out`, again using output redirection. Then open the file using `emacs`, add an empty line to the end of the file, followed by a brief description of the contents of the file, that is, by your answer to the previous question.

Step 14: Sorting numerical values using `sort`

Recall that, by default, `sort` sorts its input lexicographically. This is true whether we treat each line as the key or whether we use only part of each line as the key by which to sort the lines. In each case, pairs of keys are treated as strings and are compared lexicographically.

Using the options `-n` and `-M`, we can modify this behaviour. Create the following file:

holidays

```
Dec 26 Boxing Day
Dec 25 Christmas Day
Jan 1 New Year
```

Verify that the file you created has 3 lines. Now let's sort these lines by date:

```
$ sort -k 1M -k 2n holidays
```

This instructs `sort` to treat the first field as an abbreviated month name and the second field as a number.

Place the output of the above command into a file `step14.out`.

`sort` has numerous additional options. You are encouraged to learn about them using `man sort`.

Step 15: cut

The cut command can be used to extract only a subset of fields from each line in a file. The syntax is `cut -d separator -f fields [filename]`. The separator is again the character used to separate fields. (It is a bit unfortunate that `sort` and `cut` use different command line options for this.) The fields in each line are numbered left to right, starting from 1. The `-f` option accepts a comma-separated list of fields as well as field ranges separated by dashes. You can learn more about this using `man cut`.

Perform the following command

```
$ cut -d " " -f 2,4 holidays
```

and record its output in a file `step15.out`. You do not need to provide a description of the output, but you should be able to explain to yourself or to your classmates what this command does.

Step 16: A mini-project using pipes and various command-line tools

For this mini-project, you will use the `names` file we created earlier as input. Your goal is to count how many unique last names there are in this file. You can achieve this by combining `sort`, `uniq`, `cut`, and `wc` into a pipeline, as discussed in class. Build up your pipeline by adding one command at a time, inspecting the output after each addition. Once you are done, the output should be `3`.

Save the command you constructed (not its output!) in a file `cmd16`. You can test this code by running `source cmd16`. This should again print `3` to your terminal.

Step 17: Submit your work

Do not forget this step!

Add the following files to your SVN working copy

- `HelloWorld.java` from Step 4.
- `names` obtained at the end of Step 9.
- `step11a.out`–`step11c.out` from Step 11.
- `step12a.out`–`step12g.out` from Step 12.
- `names.csv` and `step13.out` from Step 13.
- `holidays` and `step14.out` from Step 14.
- `step15.out` from Step 15.
- `cmd16` from Step 16.

Submit these files using `svn commit` and use `svn status` or `svn log` to verify that the submission was successful.