Operating Systems Lab - Week 5: exercise
- with answers

In this lab, you will work with the UNIX shell and `bash` programming. In the first part, you will practice with simple functionalities of the UNIX shell, e.g. tree navigation commands and file inspection tools. In the second section, you will write simple `sh` programs to process a given text file.

1 The UNIX shell

In this section, you will get familiar with the most common commands of the UNIX shell. You will also learn how to extract specific lines of a text file with command line tools.

1.1 Basic commands

If you do not know commands such as `cd`, `ls`,`mkdir`, `cp`, `mv`, we suggest you read the corresponding pages of the *command-line manual*. The command-line manual is available directly from the shell, and you can open the page of a given `unix_command`, e.g. `ls`, by typing

```
man unix_command
```

in the terminal. For more info about the shell and its features, have a look at this week's slide or this online tutorial. We suggest you implement some examples proposed in the slides.

1.2 Background processes

In the first week of the term, you have seen how to *open and modify* a file with a command-line editor, e.g. emacs, nano, or vim. So far, you have been closing the editor to return to the shell, e.g. to recompile your files. The *ampersand* operator '`&`' allows you to keep the editor *open in the background* while you enter other command in the terminal, e.g. if you need to perform other *unrelated* tasks. Use your favorite editor, `editor_name`, to create a file, `students.txt` in your directory, copy the content of students.txt into it, save, and exit. Reopen `students.txt` in the background by entering the command

```
editor_name students.txt &
```

Now the shell outputs the "job number" and PID (process id) of the process running the editor but does not open the usual editor window. Try the following commands:

- `ps`, to see all running processes and corresponding PID,
- `fg process_name`, e.g. `fg vim`, to open the editor window,
- `ctrl-z` in the editor window, to return to the terminal,
- `kill PID`, to terminate the process from the terminal. [1]
- `ps`, to list the active processes in the shell,[2] and
- `killall process_name`, to kill all matching processes.

Use tab-completion not to type out entire directory names: after typing the first few characters of a directory or file, hit `tab` key to let the shell complete the name. If there's more than one match, you can press `tab` twice to see a list of matches.

[1] If, after trying to kill a process through `kill PID`, you are still seeing it, try to run `kill -SIGKILL PID` and then `editor_name students.txt &` or`kill -9 PID` to send a SIGKILL signal to the OS.

[2] You can use `ps` to obtain the PID of a given process and terminate it using `kill <PID>`.

1.3 Inspecting files

To see the content of `students.txt` without opening a text editor, you can use

- `cat`
- `more`
- `head` or `tail`

Check the command-line manual of all these commands and answer the following questions:

1. How do you stop scrolling the file and return to the shell when you use `more`?

   Answer: With the *quit* command `q`

2. What is the difference between

   ```
   more students.txt
   ```

   and

   ```
   cat students.txt
   ```

   Answer: `cat` shows the whole file without having to scroll.

3. What does the `-n` flag does in `cat`?

   Answer: `cat -n` shows the line numbers.

4. What is the output of the following command?

   ```
   cat students.txt students.txt
   ```

   Answer: The content of the file is shown twice.

5. How many lines of `students.txt` are shown if you type

   ```
   head students.txt
   ```

   and if you add the flag `-10`?

   Answer: 10 lines

6. What is the difference in the output of the two following command

   ```
   head —40 students.txt
   tail —40 students.txt
   ```

   Answer: The first shows the first 40 lines of the file and the second the last 40 lines.

For extracting *global info* from a file without inspecting its content directly you can use `wc` with various options. Check the manual to see how to use `wc` to print the *line count* of `students.txt`. Add an empty line at the end of the file. Does `wc` count it?

Answer: Enter the command

```
wc —l students.txt
```

which prints

```
203 students.txt
```

The last empty line is counted.

1.4 Filtering

You can sort the line of an input text file according to a specified criterion with `sort`. Try the following commands.

```
sort students.txt
sort —r students.txt
sort —t/ —k 2 students.txt.
```

What are the corresponding criteria for sorting the entries? How does the option `-t` work? And what is the difference between adding `-t" "` and `-t/`?

Answer: `-r` reverse the numerical order and `-t` allows you to specify a separator that you can use to focus on specific *fields* of the lines. In this case, you need to specify the priority field by adding `-k n` (for prioritizing the nth field). Another filtering command is `cut`, which also allows you to specify customized sorting strategies. Check the manual to see how you can use it to filter the information printed out from `students.txt`. Can you figure out how to show only the student names?

Answer:

```
cut —d / —f 2 students.txt
```

1.5 IO Redirection

Normally, command-line programs print to *standard output*, which is connected to the terminal by default. The IO redirection commands,

$$>, \quad <, \quad >>, \quad |$$

allow you to *read and write* data to disk or to communicate between different commands, i.e. processes, by connecting their standard input and standard output streams.

- $x>y$ redirects the output of x to file y,
- $x>>y$ redirects the output of x on file y without overwriting the file,
- $x<y$ uses the content of file y as input of the command x, and
- $x|y$ connects the standard output of command x to the standard input of command y.

Try to understand how the redirection operators work in practice by combining two or three of the UNIX commands mentioned in the previous sections as suggested below.

1. What is printed in the file `lsOut.txt` after running `ls > lsOut.txt`?

   Answer: The content of the current directory.

2. What happens if you run `ls -l >> lsOut.txt` three times?

   Answer: The file contains three times the content of the current directory.

3. What is the difference between running `wc students.txt` and `wc < students.txt`?

   Answer: The second command outputs only the stats about the file, without the file name.

4. Try to predict the output of the following command

   ```
   tail —10 students.txt | head —5
   ```

before running it.

Answer: The first five lines of the last 10 lines of the file.

5. Complete the second command below so that it produces the same output as the first one

```
sort students.txt|
head —5
sort ... students.txt| ... —5 | sort ...
```

Answer:

```
sort —r students.txt  tail -5 sort
```

Optional. Combine `cut`, `sort`, and the I/O redirection commands to print on a new file, `names.txt`, the student names (only their names) sorted alphabetically by first name.

Answer:

```
cut —d / —f 2 students.txt  sort
```

1.6 `grep`

To quickly inspect and filter text files you can also use `grep`, which allows you to print all lines that match a pattern. In particular, `grep` is a powerful tool when its argument is a *regular expression*. See wild cards list for a list of the wild cards you can use to build regular expressions in UNIX and the manual page of `grep` for further details about its syntax. Note that some regular expressions you can use with `grep` differ from the classical ones. Then answer the following questions:

1. What is the difference between the output of the two following commands

```
grep Candice students.txt
grep Ca[np] students.txt
```

Answer: The output of the second includes

```
1098/Caprice Cerrato/CS1801/CS1820/CS1830/CS1840/CS1860
```

2. How can you combine `grep` and `wc` to find the number of students taking CS1860?

Answer:

```
grep CS1860 students.txt  wc -l
```

3. How can you print the profile, i.e. the whole line, of the students who are *not* taking CS1890?

Answer:

```
grep —v CS1890 students.txt  wc -l
```

2 `sh` scripts

In this section, you combine UNIX command-line instructions into basic shell scripts. Before starting, have a look at Example 5.1, Example 5.2, and Example 5.3 for an explicit example of how to use a `for`-loop or a `while`-loop and write a program that performs a simple text filtering task.

2.1 Variables and inputs

Copy the following script into a new file, `myGrep.sh`,

```
#!/bin/sh
#myGrep.sh
IN=$2
OUT="out.txt"
PATTERN=$1
if test —f "$IN" ; then
        grep $PATTERN $IN > $OUT
        head —10 $OUT
fi
echo "file '$OUT' written"
```

and use `ls -l` to check its permissions. If you do not have the right to execute change the permission with

```
chmod u+x myGrep.sh
```

Then you can run it by typing

```
./myGrep.sh pattern file_name
```

where `file_name` should be `students.txt` and `pattern` is a standard `grep` search pattern, e.g.`100[13579]`. Can you write a single-line combination of the UNIX commands in `myGrep.sh` that produces the same output on the terminal, *except the last line* and without creating a *temporary* file `out.txt`?

Answer:

```
grep pattern fileIn.txt  head -10
```

2.2 ID filter

In a new file, `select.sh`, write a more refined version of `myGrep.sh` that accepts two integer parameters, `startID` and `endID` such that `startID` $\leq$ `endID`, and prints on the terminal the lines of `students.txt` corresponding to all students whose student ID is included in the range [`startID`, `endID`], i.e. all lines starting with an `ID` such that `startID` $\leq$ `ID` $\leq$ `endID`. The input file can be fixed and does not need to be passed as a parameter, i.e. you can write

```
IN="student.txt"
```

instead of `IN=$1` as in `myGrep.sh`. Start by completing the following `bash`-script

```
#!/bin/sh
# select.sh
IN="students.txt"
START=...
END=...
if [ "$#" —ne 2 ]; then
    echo "Usage: ... [startID] [endID]"
else
        LOOP=$START
        while [ $LOOP —le ... ]
        do
                grep ... $IN
                LOOP=`expr ...`
        done
fi
```

Answer:

```
#!/bin/sh
# select.sh
IN="students.txt"
START=$1
END=$2
FIRST=`grep $1 $IN  head -1`LAST=`grep 2IN  tail —1`
if [ "$#" —ne 2 ]; then
    echo "Usage: $0 [startID] [endID]"
else
        LOOP=$START
        while [ $LOOP —le $END ]
        do
                grep $LOOP $IN
                LOOP=`expr $LOOP + 1`
        done
fi
```

When you run it, your program should produce an output analog to

```
./select.sh 1181 1185
1181/Kiera Croslin/CS1801/CS1820/CS1890
1182/Kenny Mcclelland/CS1801/CS1820/CS1830
1183/Ilse Wheat/CS1801/CS1820/CS1830
1184/Gregorio Melia/CS1801/CS1820/CS1830
1185/Londa Stacker/CS1801/CS1820/CS1830
```

or

```
./select.sh 1181
Usage: ./select.sh [startID] [endID]
```

2.3 Print the student names

Make your program print the names of the first and last students in the range. For example, your new version should produce the following outputs

```
 ./filter.sh 1181 1185
first student=Kiera Croslin
last student=Londa Stacker
1181/Kiera Croslin/CS1801/CS1820/CS1890
1182/Kenny Mcclelland/CS1801/CS1820/CS1830
1183/Ilse Wheat/CS1801/CS1820/CS1830
1184/Gregorio Melia/CS1801/CS1820/CS1830
1185/Londa Stacker/CS1801/CS1820/CS1830
```

or

```
./select.sh 1181
Usage: ./select.sh [startID] [endID]
```

Start by inserting the following 2 lines at the right position in `select.sh` to print the first name

```
FIRST=`grep $1 $IN | head —1`
echo "first student=`echo "$FIRST" |cut —d / —f 2`"
```

Check that your program prints the first student's name as in the first line of the output above *only if* the program is launched with the expected arguments. Write two similar statements that print the name of the last student.

Answer: A possible script that works as requested is

```
#!/bin/sh
# select.sh
IN="students.txt"
START=$1
END=$2
if [ "$#" —ne 2 ]; then
    echo "Usage: $0 [startID] [endID]"
else
        FIRST=`grep $1 $IN | head —1`
        echo "first student=`echo "$FIRST" |cut —d / —f 2`"
        LAST=`grep $2 $IN | tail —1`
        echo "last student=`echo "$LAST" | cut —d / —f 2`"
        LOOP=$START
        while [ $LOOP —le $END ]
        do
                grep $LOOP $IN
                LOOP=`expr $LOOP + 1`
        done
fi
```