

COMP 311 Linux Operating System Lab

Lab 6: *Shell Usage and Configuration – Part 2*

Ahmed Tamrawi



atamrawi



atamrawi.github.io



ahmedtamrawi@gmail.com



BIRZEIT UNIVERSITY

Computer Science Department

Linux OS Laboratory Manual (V 1.2)

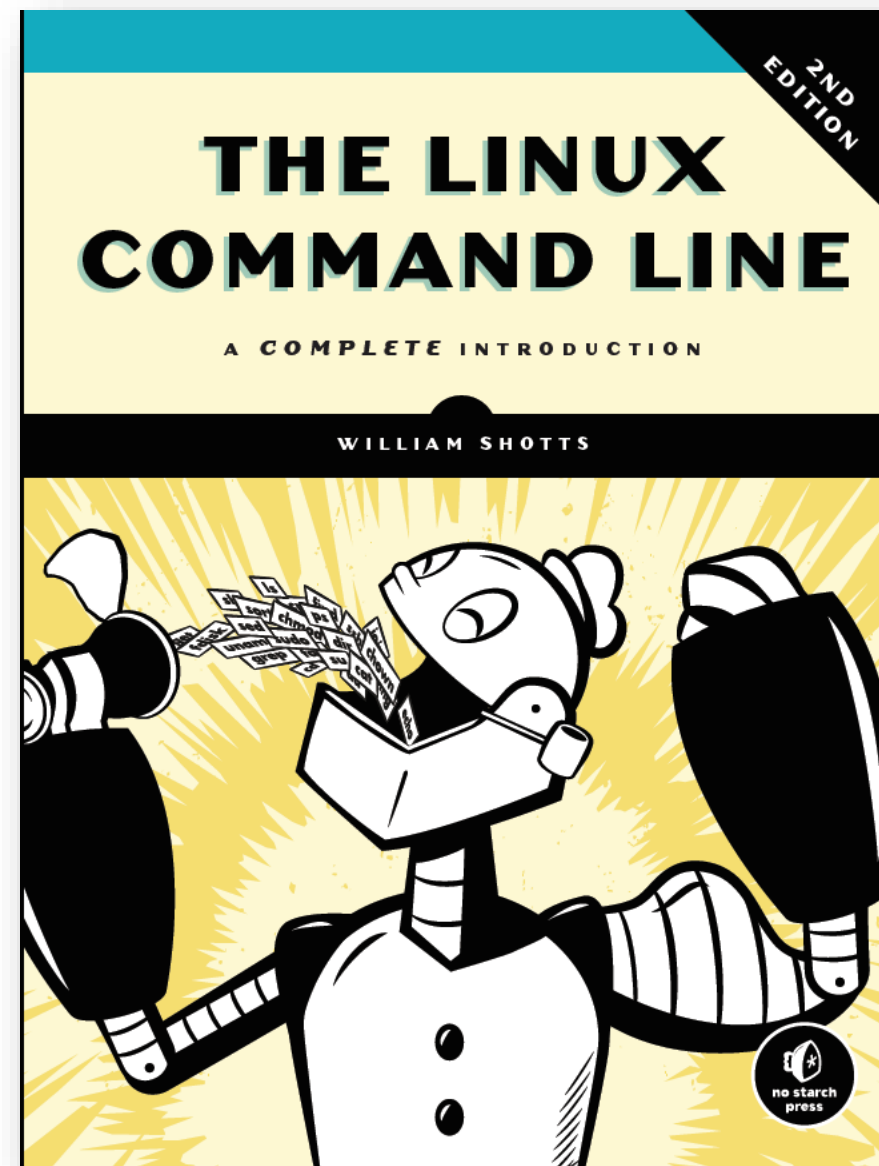
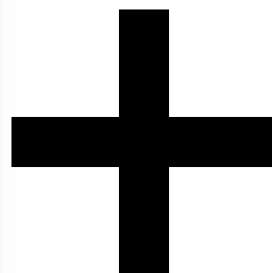
COMP311

Nael I. Qaraeen

Approved By:

Computer Science department

May 2015



I/O Redirection

- In this lab, we are going to unleash what may be the coolest feature of the command line. It's called **I/O redirection**.
- The “I/O” stands for input/output, and with this facility you can redirect the input and output of commands to and from files, as well as connect multiple commands together into powerful command pipelines.

Standard Input, Output, and Error

- Many of the programs that we have used so far produce output of some kind. This output often consists of two types.
 - **The program's results**; that is, the data the program is designed to produce.
 - **Status and error messages** that tell us how the program is getting along.
- If we look at a command like `ls`, we can see that it displays its results and its error messages on the screen.


Standard Input, Output, and Error

- Keeping with the Unix theme of “*everything is a file*,” programs such
- as `ls` send their results to a **special file** called **standard output** (often expressed as `stdout`) and their status messages to another file called **standard error** (`stderr`).
- By default, **both standard output and standard error** are *linked to the screen and not saved into a disk file*.
- In addition, many **programs take input from a facility** called **standard input** (`stdin`), which is, by default, attached to the **keyboard**.

I/O Redirection

- I/O redirection allows us to change where output goes and where input comes from.
- Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection, we can change that.

Redirecting Standard Output

- I/O redirection allows us **to redefine where standard output goes**.
- To redirect standard output to another file instead of the screen, we use the  redirection operator *followed by the name of the file*.
- Why would we want to do this? It's often useful to store the output of a command in a file.
- For example, we could tell the shell to send the output of the `ls` command to the file `ls-output.txt` instead of the screen.

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

```
[me@linuxbox ~]$ less ls-output.txt
```

Redirecting Standard Output

- Now, let's repeat our redirection test, but this time with a twist. We'll change **the name of the directory to one that does not exist**.

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt  
ls: cannot access /bin/usr: No such file or directory
```

- We received an **error message**. This makes sense since we specified the nonexistent directory `/bin/usr`.
- But why was the error message displayed on the screen rather than being redirected to the file `ls-output.txt`?

Redirecting Standard Output

- The answer is that the `ls` program does not send its error messages to standard output.
- Like most well-written Unix programs, it sends its error messages to **standard error**. *Because we redirected only standard output and not standard error*, the error message was still sent to the screen.
- We'll see how to redirect standard error in just a minute, but first let's look at what happened to our output file.

```
[me@linuxbox ~]$ ls -l ls-output.txt  
-rw-rw-r-- 1 me    me    0 2018-02-01 15:08 ls-output.txt
```

Redirecting Standard Output

```
[me@linuxbox ~]$ ls -l ls-output.txt  
-rw-rw-r-- 1 me    me    0 2018-02-01 15:08 ls-output.txt
```

- The file now has **zero length**! This is because when we redirect output
- with the **>** redirection operator, the destination file is always **rewritten from the beginning**.
- Because our `ls` command generated no results and only an error message, the redirection operation started to rewrite the file and then stopped because of the error, resulting in its truncation.

Redirecting Standard Output

- If we ever need to actually truncate a file (or create a new, empty file), we can use a trick like this:

```
[me@linuxbox ~]$ > ls-output.txt
```

- Simply using the **redirection operator with no command** preceding it will *truncate an existing file or create a new, empty file*.

Redirecting Standard Output

- How can we **append redirected output** to a file instead of overwriting the file from the beginning?

- For that, we use the **>>** redirection operator, like so:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

- Using the **>>** operator will result in the output being appended to the file. If the file does not already exist, it is created just as though the **>** operator had been used. Let's put it to the test.

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me  me  503634 2018-02-01 15:45 ls-output.txt
```

Redirecting Standard Error

- Redirecting standard error **lacks the ease of a dedicated redirection operator**. To redirect standard error, we must **refer to its file descriptor**.
- A program can produce output on any of *several numbered file streams*.
- We have referred to the first three of these file streams as **standard input, standard output, and standard error**,
- The shell references them internally as **file descriptors 0, 1, and 2**, respectively.

Redirecting Standard Error

- The shell provides a **notation for redirecting files using the file descriptor number**.
- Because standard error is the same as file descriptor number 2, we can redirect standard error with this notation:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

- The file descriptor **2** is placed immediately before the redirection operator to perform the **redirection of standard error** to the file `ls-error.txt`.

Redirecting Standard Output and Standard Error to One File

- There are cases in which we may want to **capture all of the output of a command to a single file**.
- To do this, we must **redirect both standard output and standard error** at the same time.
- We use the single notation **&>** to redirect both standard output and standard error to the file `ls-output.txt`.

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

- You may also append the standard output and standard error streams to a single file like so:

```
[me@linuxbox ~]$ ls -l /bin/usr &>> ls-output.txt
```

Disposing of Unwanted Output

- Sometimes “silence is golden” and we don’t want output from a command; we just want to throw it away.
- This applies particularly to error and status messages.
- The system provides a way to do this by redirecting output to a special file called `/dev/null`. This file is a system device often referred to as a bit bucket, which accepts input and does nothing with it.
- To **suppress error messages** from a command, we do this:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

Redirecting Standard Input

- The `cat` command reads one or more files and copies them to standard output like so:

```
[me@linuxbox ~]$ cat ls-output.txt
```

- What happens if we enter `cat` with no arguments?

```
[me@linuxbox ~]$ cat
```

- **Nothing happens**; it just sits there like it's hung. It might seem that way, but it's really doing exactly what it's supposed to do.
- If `cat` is not given any arguments, it reads from **standard input**, and since standard input is, by default, attached to the keyboard, it's waiting for us to type something!

Redirecting Standard Input

- Try adding the following text and pressing ENTER:

```
[me@linuxbox ~]$ cat  
The quick brown fox jumped over the lazy dog.
```

```
[me@linuxbox ~]$ cat  
The quick brown fox jumped over the lazy dog.  
The quick brown fox jumped over the lazy dog.
```

- In the absence of filename arguments, `cat` copies standard input to standard output, so we see our line of text repeated. We can use this behavior to create short text files.

Redirecting Standard Input

- Let's say we wanted to create a file called `lazy_dog.txt` containing the text in our example. We would do this:


```
[me@linuxbox ~]$ cat > lazy_dog.txt  
The quick brown fox jumped over the lazy dog.
```

```
[me@linuxbox ~]$ cat lazy_dog.txt  
The quick brown fox jumped over the lazy dog.
```

Redirecting Standard Input

- Now that we know how `cat` accepts standard input, in addition to filename arguments, let's try **redirecting standard input**.

```
[me@linuxbox ~]$ cat < lazy_dog.txt  
The quick brown fox jumped over the lazy dog.
```

- Using the  redirection operator, we change the source of standard input from the keyboard to the file `lazy_dog.txt`. We see that the result is the same as passing a single filename argument.

Redirecting Standard Input

- Another example is the `tr` (translate) command.
- This is a useful command used to **change input characters** and may be used to **encrypt characters**.
- For example, we can do the following to capitalize letters of the input sentence. When done entering your input press CTRL+D (EOF) to tell `tr` command that you have concluded your input.

```
ahmed@Ubuntu-Machine:~$ tr [a-z] [A-Z]  
Hey, This is Linux Operating System Lab (COMP311)!  
HEY, THIS IS LINUX OPERATING SYSTEM LAB (COMP311)!
```

Redirecting Standard Input

- We can ask the `tr` command to **append multiple inputs** using the **append input redirection operator** `<<` and the use of exclamation marks `!` to mark **the beginning and end of our input**.

```
ahmed@Ubuntu-Machine:~$ tr [a-z] [A-Z] <<!
This
is
Linux
Operating
System
Lab
(Comp 311)
!
THIS
IS
LINUX
OPERATING
SYSTEM
LAB
(COMP 311)
```

Pipelines

- The capability of commands to read data from standard input and send to standard output is utilized by a shell feature called **pipelines**.
- Using the **pipe operator** **|**, the *standard output of one command can be piped into the standard input of another*.

```
command1 | command2
```

Pipelines

- To fully demonstrate this, we are going to need some commands.
- Remember how we said there was one we already knew that accepts standard input? It's `less`.
- We can use `less` to display, page by page, the output of any command that sends its results to standard output.

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

- This is extremely handy! Using this technique, we can conveniently examine the output of any command that produces standard output.

Filters

- Pipelines are often used to **perform complex operations on data**. It is possible to put several commands together into a pipeline.
- Frequently, the commands used this way are referred to as **filters**. Filters take input, change it somehow, and then output it.
- The first one we will try is `sort`. Imagine we wanted to make a combined list of all the executable programs in `/bin` and `/usr/bin`, put them in sorted order, and view the resulting list.

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

uniq: Report or Omit Repeated Lines

- The `uniq` command is often used in conjunction with `sort`. `uniq` accepts a sorted list of data from either standard input or a single filename argument and, by default, **removes any duplicates from the list**.
- In this example, we use `uniq` to **remove** any duplicates from the output of the `sort` command.

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less
```

- If we want to **see the list of duplicates** instead, we add the `-d` option to `uniq` like so:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

wc: Print Line, Word, and Byte Counts

- The `wc` (word count) command is used to **display the number of lines, words, and bytes** contained in files.

```
[me@linuxbox ~]$ wc ls-output.txt  
7902  64566 503634 ls-output.txt
```

- To see the number of items we have in our sorted list, we can do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l  
2728
```

grep: Print Lines Matching a Pattern

- grep is a powerful program used to find text patterns within files.

```
grep pattern filename
```

- When grep encounters a “pattern” in the file, it prints out the lines containing it. The patterns that grep can match can be very complex, but for now we will concentrate on **simple text matches**.
- We’ll cover the advanced patterns, called **regular expressions** in later lab.

grep: Print Lines Matching a Pattern

- Suppose we wanted to find all the files in our list of programs that had the word **zip** embedded in the name. We would do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
```

- There are a couple of handy options for grep.
 - -i, which causes grep to ignore case when performing the search (normally searches are case sensitive).
 - -v, which tells grep to print only those lines that do not match the pattern.

head/tail: Print First/Last Part of Files

- Sometimes you don't want **all the output from a command**. You might want only the first few lines or the last few lines.
- The head command prints the **first 10 lines of a file**, and the tail command prints **the last 10 lines**.

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
total 343496
-rwxr-xr-x 1 root root      31316 2017-12-05 08:58 [
-rwxr-xr-x 1 root root       8240 2017-12-09 13:39 411toppm
-rwxr-xr-x 1 root root    111276 2017-11-26 14:27 a2p
-rwxr-xr-x 1 root root     25368 2016-10-06 20:16 a52dec
[me@linuxbox ~]$ tail -n 5 ls-output.txt
-rwxr-xr-x 1 root root      5234 2017-06-27 10:56 znew
-rwxr-xr-x 1 root root       691 2015-09-10 04:21 zonetab2pot.py
-rw-r--r-- 1 root root       930 2017-11-01 12:23 zonetab2pot.pyc
-rw-r--r-- 1 root root       930 2017-11-01 12:23 zonetab2pot.pyo
lrwxrwxrwx 1 root root         6 2016-01-31 05:22 zsoelim -> soelim
```

head/tail: Print First/Last Part of Files

- `tail` has an option that allows you to **view files in real time**.
- This is useful for watching the progress of log files as they are being written. Superuser privileges are required to do this on some Linux distributions because the `/var/log/messages` file might contain security information.

```
[me@linuxbox ~]$ tail -f /var/log/messages
Feb  8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 13:40:05 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1652 seconds.
Feb  8 13:55:32 twin4 mouted[3953]: /var/NFSv4/musicbox exported to both 192.168.1.0/24 and
twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:07:37 twin4 dhclient: DHCPREQUEST on eth0 to 192.168.1.1 port 67
Feb  8 14:07:37 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 14:07:37 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1771 seconds.
Feb  8 14:09:56 twin4 smartd[3468]: Device: /dev/hda, SMART Prefailure Attribute: 8 Seek_Time_
Performance changed from 237 to 236
Feb  8 14:10:37 twin4 mouted[3953]: /var/NFSv4/musicbox exported to both 192.168.1.0/24 and
twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:25:07 twin4 sshd(pam_unix)[29234]: session opened for user me by (uid=0)
Feb  8 14:25:36 twin4 su(pam_unix)[29279]: session opened for user root by me(uid=500)
```

tee: Read from Stdin and Output to Stdout and Files

- The tee program **reads standard input and copies it to both standard output** (allowing the data to continue down the pipeline) and **to one or more files**.
- This is useful for capturing a pipeline's contents at an intermediate stage of processing.

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

Pipelines Example 1

```
ahmed@Ubuntu-Machine:~$ cat /etc/passwd | grep ahmed
ahmed:x:1000:1000:Ahmed Tamrawi,,,:/home/ahmed:/bin/bash
ahmed@Ubuntu-Machine:~$ cat /etc/passwd | grep ahmed | cut -d: -f5
Ahmed Tamrawi,,
ahmed@Ubuntu-Machine:~$ cat /etc/passwd | grep ahmed | cut -d: -f5 | cut -d, -f1
Ahmed Tamrawi
ahmed@Ubuntu-Machine:~$ cat /etc/passwd | grep ahmed | cut -d: -f5 | cut -d, -f1 | cut -d' ' -f1
Ahmed
```

Pipelines Example 2

```
ahmed@Ubuntu-Machine:~$ who
ahmed      :0                2020-10-20 08:50 (:0)
ahmed@Ubuntu-Machine:~$ who | tr -s " "
ahmed :0 2020-10-20 08:50 (:0)
ahmed@Ubuntu-Machine:~$ who | tr -s " " | cut -d' ' -f3,4
2020-10-20 08:50
ahmed@Ubuntu-Machine:~$
```

Important Notes for Lab Solution Submission

- **What do you need to submit for each lab?**
 - Reply to the lab message on Ritaj with the following two items:
 - **[OPTIONAL]** *Your lab solution on a separate document.*
 - **[MUST]** *Script log file*; the `script` log file will prove that you have literally solved the lab tasks yourself as it contains timestamps as well as your username and machine name.
 - If you do not send your `script` log file, then you will be rewarded with **zero**.

Important Notes for Lab Solution Submission

- **How to generate the script log file?**
 - While solving lab tasks, perform the following per each task part:
 - `script -a UniversityID_FirstnameLastName.log`
 - solve the lab task using proper command, scripts, etc.
 - `exit`
 - At the beginning of your log file, you need to execute the following commands – not doing so will result on **zero** grade.

```
uname -a
cd; pwd; whoami; finger
tail -2 /etc/passwd
echo FullName
echo UniveristyId
```