SWitCH – Computing systems and networks (SCOMRED) 2018/2019

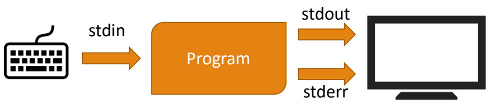
Laboratory Script – Week 03

IMPERATIVE: After each practical activity (in yellow), students are supposed to reflect on results, and wait for the teacher's acknowledgment before progressing to the next practical activity in this script.

1. Input/Output Redirection

In Linux, whenever a new process is started to execute a program, this new process sees three files that it can used for input/output: **stdin**, **stdout** and **stderror**:

- stdin (0): standard input. By default, the keyboard;
- stdout (1): standard output. By default, the terminal;
- stderr (2): standard error. By default the terminal.



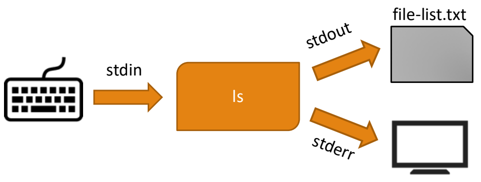
All open files are represented internally by what are called file descriptors and these are represented by numbers starting at zero: **stdin** is file descriptor 0 (zero), **stdout** is file descriptor 1 (one), and **stderr** is file descriptor 2 (two).

We can redirect the three standard file descriptors to, for example, get input from either a file or another command instead of from our keyboard. We can also, for example, write output and errors to files, or send them as input to other subsequent commands. This is a very useful and common feature of Linux.

We perform I/O redirection using '**>**' and '**<**'. Let us see a few examples.

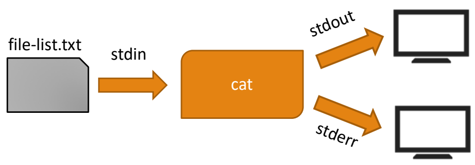
```
ls -1 > file-list.txt
```

In this example, we redirect the **`stdout`** of the command **`ls`** to a file called **`file-list.txt`**. The command **`ls -1`** lists each file in a single line, with no further info[1] and this output will be written to **`file-list.txt`** instead of printed to the terminal as usual.



```
cat < file-list.txt
```

Here, we redirect the **`stdin`** of the command **`cat`** to **`file-list.txt`**. Because **`cat`** (with no arguments) waits to receive input from **`stdin`** and prints it to **`stdout`**, the result is that **`cat`** will print the contents of the file[2].



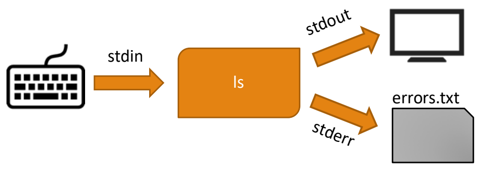
[1] You can execute **ls -1** and see the output for yourself before redirecting it.

[2] This is similar to the result of issuing the command **cat file-list.txt**. The difference is that we achieved it with I/O redirection instead of passing the file as argument to the command.

We can specify the file descriptor number of the file we want to redirect: '**1>**' for **stdout** and '**2>**' for **stderr**. Usually, we do not use '**1>**' as this is the default redirection. For example:

```
ls afile.txt 2> errors.txt
```

will result in printing the name of the file (the output of **ls** afile.txt) to the terminal, but any errors encountered by **ls** , for example, if afile.txt does not exist (which should be printed to **stderr**) will be output to the file **errors.txt** instead of the terminal.



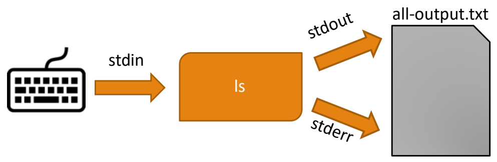
We could also redirect the output of **ls afile.txt** to a file (**file-list.txt)** as before[3]:

```
ls afile.txt > filelist.txt 2> errors.txt
```

To specify both **stdout** and **stderr** use '**&**'. For example:

```
ls afile.txt &> all-output.txt
```

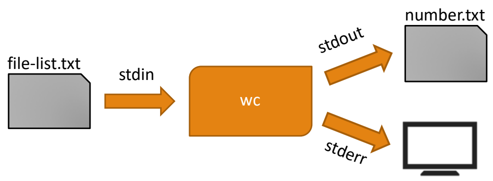
Will redirect both **stdout** and **stderr** to the file **all-output.txt**.



We can combine redirections (**wc -l** counts the number of lines in a file):

```
wc -l < file-list.txt > number.txt
```

We redirect the **stdin** of **wc** to the file **file-list.txt**. Then, **wc -l** will output the number of lines of the file, which is redirected to the file **number.txt**.



Important: Using '**>**' for redirection will clear the destination file. Use '**>>**' to append to the contents of the file:

```
ls >> file-list.txt
```

The above command will add to the contents of the file **file-list.txt**.

PRACTICE:

Use **cat**[4] and I/O redirection to:

a) Create a file named **test1**, type "Line1" and press **Ctrl-D**.

b) Create a file named **test2**, type "Line2" and save it by pressing **Ctrl-D**.

c) Create a file named **test3**, type "Line3" and save it by pressing **Ctrl-D**.

d) Concatenate the files **test1** and **test2** into a file named **newtest**.

[3] This is equivalent to `ls afile.txt 1> file-list.txt 2> errors.txt`. We usually omit the '1>'.

[4] **cat**: a command-line utility that waits for input entered and outputs it to **stdout** (**Ctrl+D** exits; to be more precise, it sends a special character called **EOF** – **E**nd **O**f **F**ile – which tells **cat** that the input ended, thus it can terminate)

e) View the contents of **newtest**.

f) Append the file test3 at the end **newtest**.

g) View the contents of the **newtest** file.

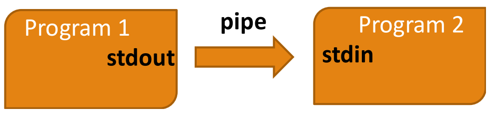
h) Append the following text to **newtest**: "Line4".

i) View the contents of the **newtest** file.

Pipes

The philosophy of Linux (originating from UNIX) is to write many simple programs that can be put together to produce more complex behaviour. This is often achieved with the help of **pipes**.

Pipes allow to redirect the output of one program to the input of another and, in the shell, it is represented by the vertical bar '**|**'. Suppose you have 3 different programs (**program1**, **program2**, **program3**), each reads from **stdin** and produces a result to **stdout**. You could "pipe" these programs together:



program1 | program2 | program3

In the above, the output of **program1** is sent to **program2** which produces an output that is sent to **program3**.

Let us see a more concrete example. We will use **ls -1** to produce a list of files (one per line: '**-1**' option), feed this list to **wc -l** that will count the number of lines in the list of files (received via **stdin**) and write this to a file called **number.txt**:

```
ls -1 | wc -l > number.txt
```

Let us look at a few more useful examples. Get only the first three files listed by ls:

```
ls | head -n3
```

Get only the third file listed by ls:

```
ls | head -n3 | tail -n1
```

Get the largest file (**ls -S -1** lists files line by line, sorted by size):

```
ls -S -1 | head -n1
```

Sort some input and save results to a file (**sort** is a utility that sorts lines of text received in **stdin**, printing them to **stdout**):

```
echo -e ”third \nfirst \nsecond" | sort > sorted-lines.txt
```

PRACTICE:

a) Try the command **dmesg**[5] . Try to use a pipe and the **less** (**less** is a command that receives input from **stdin** and prints this output page-by-page) to see the output of **dmesg** page-by-page.

[5] **dmesg** is an utility that prints all startup messages. It is usually a very long output.

In the command line, write[6]: **`shuf -i 1-100 -r -n 200 > numbers.txt`**

b) What will the above command do (see footnote[5])?

c) Using the command **`uniq`** (which receives input from **`stdin`** and outputs to **`stdout`** only the unique values) write only the numbers that are not repeated in the file **`numbers.txt`** to a new file called **`unique_numbers.txt`**.

d) Use the command **`sort -g`** (**-g** tells sort to treat the input as numbers) and the command **`head`**[7], to get the smallest number in **`numbers.txt`**.

2. The Global Regular Expression Print - `grep`

The grep is one of the most useful Linux programs. It is used to search the input for a **pattern** and, when this pattern is found, it prints the lines matching it. The patterns matched by grep are defined using **regular expressions**, which we will discuss later. Let us start with some simple **grep** examples.

Find lines with 'dorian' (caps matter for grep; this will not match "Dorian") in the file **`pg174.txt`**:

```
grep 'dorian' pg174.txt
```

Find lines with 'dorian' in several files:

```
grep 'dorian' file1.txt file2.txt file3.txt
```

Feed **`grep`** input from **`stdin`**:

```
cat pg174.txt | grep 'dorian'
```

Feed **`grep`** input from **`stdin`**, in this case produced by **`ls`** (**`grep`** will try to match lines with "afile"):

```
ls | grep 'afile’
```

PRACTICE:

a) Using **`grep`**, find the details of a user with "switch" in its username (in **`/etc/passwd`**).

b) Find if the word "error" was output during system startup (use **`dmesg`** to output system startup messages)[8].

[6] The command **`shuf -i 1-100 -r -n 200`** will generate 200 random numbers between 1-100.

[7] You should know what **`head`** does by now; if not: **`man head`**

[8] Use the grep options **`-B`*n*** and **`-A`*n*** to print some context of the line where the pattern was found: ***n*** lines before (**`-B`*n***) and ***n*** lines after (**`-A`*n***). Replace ***n*** with an integer. E.g.:
`grep -B5 -A5 “word”`
will print 5 lines before and after the line where "word" was found.

Regular Expressions

A regular expression (sometimes called *regex* or *regexp*) is defined with a sequence of characters, where some characters have particular meanings. These expressions can be used to search, edit and manipulate text. They can be very powerful but difficult to understand, particularly for newcomers.

Regular expressions are used by many tools and text editors. Most programming languages (e.g. Java, Perl, Python, etc) have some support for regular expressions. Unfortunately, there are small syntax differences in the several implementations of regular expressions, so some regular expressions might work slightly differently in different tools/applications. In this case, we are using **`grep`**, and the regular expressions here are to be used with this tool.

Let us see some simple regular expression examples.

Regular expression	**Meaning**	**Example matches**
`”a string”`	Matches any line with “a string” in it	“**a string**”, “this is **a string**”, “a sentence with **a string**”, …
`” a\s+string”`	Matches ”**a**”, followed by one or more whitespace characters (‘**\s+**’) followed by the word ”string"	“**a string**”, “**a string**”, “**a string**”, “any sentence with **a string**”, …
`”^a string$”`	Matches the entire input line with exactly “**a string**” (’^’ means start; ’$’ means end).	”**a string**” (only)

As a first example, use the following to match all lines that start with “CHAPTER” in the file **`pg174.txt`**:

```
grep “^CHAPTER” pg174.txt
```

Now that you have the general idea of what regular expressions do, have a look at this quick reference:

\	Does not interpret the next character (to ignore special characters: ‘**.**’, ‘**+**’, ‘*****’,...)
^	Matches the beginning of a line
$	Matches the end of the line
.	Matches any character
\s	Matches whitespace
\S	Matches any non-whitespace character
*****	Repeats previous character zero or more times
?	Repeats previous character zero or one time (requires “**–E**” option)
+	Repeats previous chracter one or more times (requires “**–E**” option)
{3}	Matches previous character 3 times (requires “**–E**” option)
[aeiou]	Matches a single character in the listed set
[^XYZ]	Matches a single character not in the listed set
[a-z0-9]	The set of characters can include a range
(...)	Capture everything enclosed (requires “**–E**” option)
(a|b)	Match either a or b (requires “**–E**” option)

Note that some options are what **grep** defines as extended functionality and require the **-E** option.

Using these, let us see a few more examples.

Regular expression	**Meaning**	**Example matches**
"b[aeiou]t"	Match any string with a "**b**" followed by any of the characters "**a**", "**e**", "**i**", "**o**", "**u**", followed by "**t**".	"**bat**", "**bet**", "**bit**", "**bot**", "**but**", "cricket **bat**", "**bit**ter lemon"
"^[0-9]+(\.[0-9]+)?"	Matches a string started ('**^**') by one or more digits ('**[0-9]+**'), followed by zero or more occurrence ('**?**') of the character "**.**" ('**\.**'), followed by one or more digits ('**[0-9]+**').	"**5**", "**1.5**", "**2.21**", …
" ^[a-z0-9]{5,15}$"	Matches a string with only letters and numbers and a length between 5 and 15 characters.	"**astring**", "**12345**", "**aaaaa**", "**abcdefgh123456**", …

PRACTICE:

Using **grep**, find sentences in the file **pg174.txt**:

a) Find sentences attributed to Lord Fermor (sentences with "said Lord Fermor").
b) Find sentences starting with the word "He".
c) Find sentences that are completely within quotes (start and end with ").
d) Find sentences with "asked you" or "asked me".
e) Find the last 5 lines of each chapter.
f) Find the first 5 lines of Chapter 1 and Chapter 10 to Chapter 19.

3. The **find** command

The **find** command is another very useful common utility in Linux. It can be used to search files and directories based on many criteria, such as: name, permissions, file type, user, group, date, or size.

The basic find command arguments are as follows:

```
find start_directory test options criteria_to_match action_to_perform_on_results
```

Let us see some examples. The most common usage of find is to find a file by name:

```
find . -name "pg174.txt"
```

The above command will search for files named **pg174.txt** (**-name pg174.txt**) in the current directory (**.**). Note that find will search inside all directories inside the current working directory. You can specify any starting directory. For example, **/home**:

```
find /home -name "pg174.txt"
```

You can also use wildcards[9] for the filename (`*` matches any number of characters; `?` matches one character). The following finds all files with **`.txt`** at the end of the filename:

```
find /home -name "*.txt"
```

The following finds all files with a filename composed of 4 characters followed by **`.txt`**:

```
find /home -name "????.txt"
```

You can also find files that do not match the criteria (**`-not`**) option (find files not named **`pg174.txt`**):

```
find /home -not -name "pg174.txt"
```

You can find files by filetype (`d`=directories, `f`=regular file):

```
find . -type d -name "somedirectory"
```

Find files by modification time (`-mtime`). When you use the `-mtime` option, you have to specify +/- the number of days, where "-" "means less than" and "+" means "more than". So, to find files modified less than 1 day ago:

```
find . -mtime -1
```

To find files modified more than 1 day ago:

```
find . -mtime +1
```

Find allows to execute commands on each file found with the option **`-exec`**. So, you can, for example, find all **`.txt`** and search for a certain word ("dorian" in the example) using grep:

```
find . -name "*.txt" -exec grep "dorian" {} \;
```

In the example above, the commands after **`-exec`** will be executed for each file that matches the criteria. The **`{}`** is used in place of the filename and \; is used to indicate the end of the input to the **`-exec`** option.

Another example where we run **`ls -l`** for each file found (**`{}`** is replaced by the filename):

```
find . -name "*.txt" -exec ls -l {} \;
```

[9] A wildcard is a character that can be used as a substitute for a range of characters. This of wildcards as a simple version of regular expressions.

PRACTICE:

Inside you home folder, two files named **`test1.txt`** and **`test2.txt`**, and also two directories called **`test1`** and **`test2`**[10].

a) Use **`find`** to search for files named "test*" inside you home folder.

b) Use **`find`** to search for directories named "test*" inside you home folder.

c) Use **`find`** to remove all directories named "test*" inside you home folder.

d) Use **`find`** to remove write permissions all **`.txt`** files inside you home folder.

[10] To create the file you can do: **`echo "this is a test" >`** ***`filename`***; To create a directory, do: **`mkdir`** ***`directory_name`***

4. Getting help about commands (manual pages)

Nobody is supposed to know the syntax of every command, what a BASH user and script developer is required to know is about the commands that exist and may be useful to him. Once the command name is known, the **man** command may be used to get the exact syntax (arguments usage).

Manual pages are organized in sections (check **man man**), some sections are about commands, others for instance about C functions, some pages exist in different sections because names are the same.

For instance **printf**, it's an external command, the manual page is in section 1, however there's also a printf library C function available at section 3. By default the man command starts by searching the manual page in section 1 (external commands).

One other issue regarding commands manual pages is about internal and external commands, the manual pages are about external commands, to get information about internal commands the manual page of the shell must be used, for instance for BASH **man bash**.

For the sake of efficiency many external commands have internal commands that replace them, usually they will have the same features and syntax.

External commands can be directly called by specifying the full path, the full path of an external command can be known by using the **which** command (as far as the executable file is on the PATH).

PRACTICE:

```
which printf
```

This returns **/usr/bin/printf**, that's the external command, now try:

```
man printf
```

This is the manual page of the printf external command, extensive information can be also attained by using the command's --help option:

```
/usr/bin/printf --help
```

This is the internal BASH version:

```
printf --help
```

It doesn't support the **--help** option. To get help you should use the BASH manual page (man bash), though they should be similar.

5. BASH scripts

We already know a BASH script is an executable text file (with the execute permission) that starts with **#!/bin/bash** line. The operating system will then recognise this file as an interpreted program to be executed by **/bin/bash**. The file should contain a sequence of lines with commands the BASH is able to execute, they will all get executed one after the other, line by line, starting from the first line.

To create and edit text files a text editor is required, for beginners, either **nano** or the mc's text editor are two good alternatives. The **nano** text editor is already available as part of the packages that are installed by default in your Linux distribution.

To use mc (GNU Midnight Commander), it must be installed first (optional):

sudo apt install mc

a) Create the following BASH script:

```
#!/bin/bash
echo "I've a number between 0 and 100, it may even be 0 or 100"
echo "Try to guess it!"
NUM=$(shuf -i 0-100 -n 1)
TRIES=0
GUESS=101
while [ $NUM != $GUESS ]; do
 read -p "Enter or guess please: " GUESS
 TRIES=$(($TRIES+1))
 if [ $GUESS -gt $NUM ]; then echo "Sorry, too large!"; fi
 if [ $GUESS -lt $NUM ]; then echo "Sorry, too short!"; fi
done
echo "Very well, you have guessed it in $TRIES tries"
```

Create this content in a new file named **guessNumberGame** in your current working directory, for instance with **nano** type:

```
nano guessNumberGame
```

Warning: don't copy & past the above content, you must type it by hand, otherwise some characters won't be correctly copied.

Give it the execute permission:

```
chmod +x guessNumberGame
```

Test the script by playing the game a couple of rounds.

```
./guessNumberGame
```

b) Add the following functionality to this script

For each user the best score he has ever achieved should be stored persistently. When each round ends, he must be informed if he has or not beaten his previous best score, and a congratulations message should be presented if so.

Suggestions:

- To be persistent, the best score must be stored in a file, to be personal, it should be stored in the user's home folder, for instance **~/.guestNumberGame.topScore**
- To get the file's content into a variable use VARNAME=$(cat ~/.guestNumberGame.topScore), remember on the first round the file will not exist.
- To place a new value in the file just echo the variable's value redirecting the output to the file.

Test the game again with the new functionality you have implemented.

How could you cheat the game as if you have achieved a previous best score of 2 tries only?

c) The following BASH script lists every valid user's login name and home folder:

```
#!/bin/bash
USERSLIST=$(getent passwd|cut -d ":" -f 1,6)
for USR in $USERSLIST; do
 echo "${USR%:*} - ${USR#*:}"
done
```

To list valid users and valid groups, the **getent** command should be used instead of listing the contents of files /etc/passwd and /etc/group. Valid users and groups in Linux may not be limited to those files contents. The output format of getent is the same as on those files.

The output of the getent command is sent (through a pipe) to the cut command to extract required data, the fields separator is the colon (-d ":") and we want fields number 1 (username) and number 6 (home folder). Each element of the list will have the format USERNAME:HOMEFOLDER.

Afterwards, by using the variable expansion with suffix and prefix pattern removal, the two elements are isolated.

PRACTICE:

Create the script file with the above content, for instance named **usershomes** and test it.

Add a new functionality such as the script checks if the home folder exists and it's a directory, if otherwise a conforming warning message should be presented.

Suggestion: check the **test** command, **man test**. Even though this manual page is about the external command, the internal **test** command should work the same way. You may also check it on the BASH manual page (**man bash**).

d) Create a BASH script that for each valid group (getent group) counts how many valid users (getent passwd) have that group as primary group.

Before proceeding let's add two groups and some users having those groups as their primary group:

```
sudo groupadd scomredgrp1
sudo groupadd scomredgrp2
sudo useradd -m -g scomredgrp1 scomreduser1
sudo useradd -m -g scomredgrp1 scomreduser2
sudo useradd -m -g scomredgrp2 scomreduser3
```

Users named scomreduser1 and scomreduser2 have primary group scomredgrp1, and user scomreduser3 has primary group scomredgrp2.

Suggestions:

- The primary group of each user is a user account's attribute (forth field), but is stored there as a GID (Group Identifier). To list every user's primary group GID (forth field) you may use:

  getent passwd | cut -d ":" -f 4

- The group identifier (GID) of each group is an attribute of the group account (third field), to get both the group name and GID (first and third fields) for every group you may use:

  getent group | cut -d ":" -f 1,3

  To go through the list of groups, use a **for** loop. Each element of this list will have the form GROUP-NAME:GID

  If one of these elements is stored in variable GNAMEGID, then you can extract the group name by expanding it as ${GNAMEGID%:*}, and you can extract the GID by expanding ${GNAMEGID#*:}. In the first case it's a prefix pattern removal and in the second case a suffix pattern removal.

- Once you have the GID of each group, then you can use the grep command to filter user accounts with that GID and pass the output (pipe) to the **wc** command to count the number of lines (**man wc** for details).

Before adding commands to the script, test those commands on the command line to see if they are working as expected.

The desired output for this script is a sequence of lines in the following format:

GROUPNAME - NUM users

Where NUM is the number of users having GROUPNAME as their primary group.