Lab report #3

ADS - System Administration

Alexis Martins, Pablo Saez, Thomas Germano

March 24, 2024

Contents

Task 1: Exercises on redirection

Run the following commands and tell where stdout and stderr are redirected to

```
./out > file
```

Redirect the output of `stdout` to `file` and write the output of `stderr` to the console.

```
./out 2> file
```

Redirect the output of `stderr` to `file` and write the output of `stdout` to the console.

```
./out > file 2>&1
```

`> file` will redirect `stdout` to `file`. Then `2>&1` means that `stderr` should be redirected to `stdout`. `stdout` being redirected to `file`, `stderr` will also be redirected to `file`.

```
./out 2>&1 > file
```

First we redirect the output of `stderr` to `stdout`. `stdout` is not yet redirected to `file`, so `stderr` will write in the console. Then `stdout` will be redirected to `file`.

```
./out &> file
```

This is a short way to write `./out > file 2>&1`. It will have the same behavior which means both `stdout` and `stderr` will be redirected to `file`.

What do the following commands do

```
cat /usr/share/doc/cron/README | grep -i edit
```

`cat /usr/share/doc/cron/README` will write the content of the `README` to `stdout`, but the content is piped which means that it won't print to `stdout` but it will be used as input in `grep -i edit` which will look for all the lines having the word `edit` in case insensitive and display them.

```
$ ./out 2>&1 | grep -i eeeee
grep: eeeee: No such file or directory
```

The output of `stderr` is redirect to `stdout`, but like the previous question it won't exacty print it's content to `stdout` but it will be used as input for the next command (because of the pipe). This input will be the string "OEOEOEOEOE". Then it will search for a string containing five consecutive 'e', but there the given string doesn't have that sequence. The program doesn't produce this sequence, so the `grep` command will generate an error.

```
$ ./out 2>&1 >/dev/null | grep -i eeeee
EEEEE
```

The first behavior of redirecting `stderr` to `stdout` is similar to the previous command. Then the `stdout`is redirected to `/dev/null`

However, because the order of redirections matters, the `stderr` (which is now mixed with the standard output) is not redirected to '/dev/null' and instead goes to 'grep'. It will process those messages and output 'EEEEE'.

Write commands to perform the following tasks

Produce a recursive listing, using ls , of files and directories in your home directory, including hidden files, in the file /tmp/homefileslist

```
ls -aR ~ > /tmp/homefilelist
```

Produce a (non-recursive) listing of all files in your home directory whose names end in .txt , .md or .pdf , in the file /tmp/homedocumentslist . The command must not display an error message if there are no corresponding files

```
find ~/ -maxdepth 1 -iname '*.md' -or -iname '*.pdf' -or -iname '*.txt' > /
    tmp/homedocumentslist
```

Task 2: Log analysis

```
curl https://ads.iict.ch/ads_website.log > log
```

How many log entries are in the file

```
$ wc -l log
2781 log
```

How many accesses were successful (server sends back a status of 200) and how many had an error of "Not Found" (status 404)

```
$ cut -f10 log | grep '200\|404'| sort | uniq -c
   1610 200
     21 404
```

What are the URIs that generated a "Not Found" response? Be careful in specifying the correct search criteria: avoid selecting lines that happen to have the character sequence 404 in the URI.

```
$ cut -f9-10 log | grep 404 | cut -f1 | grep -v 404

"GET /heigvd-ads?website HTTP/1.1"
"GET /heigvd-ads?lifecycle HTTP/1.1"
"GET /heigvd-ads?cors HTTP/1.1"
"GET /heigvd-ads?policy HTTP/1.1"
"GET /heigvd-ads?website HTTP/1.1"
"GET /heigvd-ads?lifecycle HTTP/1.1"
"GET /heigvd-ads?policy HTTP/1.1"
"GET /heigvd-ads?cors HTTP/1.1"
"GET /heigvd-ads?cors HTTP/1.1"
```

```
"GET /heigvd-ads?policy HTTP/1.1"
"GET /heigvd-ads?website HTTP/1.1"
"GET /heigvd-ads?lifecycle HTTP/1.1"
"GET /heigvd-ads?lifecycle HTTP/1.1"
"GET /heigvd-ads?cors HTTP/1.1"
"GET /heigvd-ads?cors HTTP/1.1"
"GET /heigvd-ads?policy HTTP/1.1"
"GET /heigvd-ads?lifecycle HTTP/1.1"
"GET /heigvd-ads?policy HTTP/1.1"
"GET /heigvd-ads?policy HTTP/1.1"
"GET /heigvd-ads?policy HTTP/1.1"
"GET /heigvd-ads?cors HTTP/1.1"
```

How many different days are there in the log file on which requests were made

```
$ cut -f3 log | cut -c2-12 | uniq | wc -l
21
```

How many accesses were there on 4th March 2021

```
$ cat log | grep -c "04/Mar/2021"
423
```

Which are the three days with the most accesses?

```
$ cut -f3 log | cut -c2-12 | uniq -c | sort -nr | head -3
898 13/Mar/2021
580 06/Mar/2021
423 04/Mar/2021
```

Which is the user agent string with the most accesses

```
$ cut -f17 log | sort | uniq -c | sort -nr | head -1
423 "Mozilla/5.0 (Windows NT 6.3; WOW64; rv:27.0) Gecko/20100101 Firefox
    /27.0"
```

If a web site is very popular and accessed by many people the user agent strings appearing in the server's log can be used to estimate the relative market share of the users' computers and operating systems. How many accesses were done from browsers that declare that they are running on Windows, Linux and Mac OS X (use three commands)

```
$ cut -f17 log | grep -ci 'Windows'
1751

$ cut -f17 log | grep -ci 'Linux'
180

$ cut -f17 log | grep -ci 'Mac OS X'
```

```
693
```

Read the documentation for the tee command. Repeat the analysis of the previous question for browsers running on Windows and insert tee into the pipeline such that the user agent strings (including repeats) are written to a file for further analysis (the filename should be useragents.txt)

```
$ cut -f17 log | grep -i 'Windows' | tee useragents.txt | wc -l
1751
```

And the content of the file `useragents.txt`

```
"Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/32.0.1700.107 Safari/537.36"
"Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/32.0.1700.107 Safari/537.36"
"Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/32.0.1700.107 Safari/537.36"
...
```

Why is the file access.log difficult to analyse, consider for example the analysis of question 7, with the commands you have seen so far?

Because if we use spaces to separate fields, it will be hard to really isolate them. For example in the user agent, there are a lot of spaces which means that it wouldn't be possible to correctly isolate the whole user agent.

Task 3: Conversion to CSV

```
$ echo "Date,Accesses" > accesses.csv | cut -f3 log | cut -c2-12 |  uniq -c
    | awk '{print $2","$1}'  >> accesses.csv
```

More informations in the `files` folder.