Lab 03 - Pipelines

ADS

Anthony David, Felix Breval, Timothée Van Hove

24 mars 2024

Table des matières

Task 1 : Exercices on redirection

Compile the following C++ program, called out.cpp , into an executable. The program writes a series of O characters on the stdout stream and a series of E characters on the stderr stream:

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
    for (int i = 0; i < 5; ++i) {
        cout << "O";
        cerr << "E";
    }
    return EXIT_SUCCESS;
}
```

Run the executable with ./out 1. Run the following commands and tell where stdout and stderr are redirected to.

a. `./out > file`

What happens: `stdout` is redirected to the `file` file and `stderr` is not redirected.

Explanation: The `>` operator is used to redirect `stdout` to a file named `file`. This means all `O` characters are redirected to `file`. However, since `stderr` is not redirected, the `E` characters appear on the terminal.

b. `./out 2> file`

What happens: `stdout` is not redirected whereas `stderr` is redirected to the `file` file.

Explanation: `2>` redirects only `stderr` (which is file descriptor 2) to `file`. So, the `O` characters appear on the terminal since `stdout` is not redirected.

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

What happens: Both `stdout` and `stderr` are redirected to the `file` file.

Explanation: `> file` redirects `stdout` to `file` and `2>&1` redirects `stderr` to wherever `stdout` is currently going. Since `stdout` is going to `file`, `stderr` will also be redirected to `file`.

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

What happens: `stdout` is redirected to the `file` file, and `stderr` is not redirected.

Explanation: `2>&1` redirects `stderr` to wherever `stdout` is currently going, which is still the terminal at this point. Then, `> file` redirects `stdout` to `file`, but this does not affect `stderr`, which has already been redirected to `stdout`. As a result, `E` characters appear on the terminal, and `O` characters are saved in `file`.

e. `./out &> file`

What happens: Both `stdout` and `stderr` are redirected to the `file` file.

Explanation: `&>` is a shortcut for redirecting both `stdout` and `stderr` to the same location, in this case, `file`. This means both `O` and `E` characters are redirected in `file`, and nothing appear on the terminal.

What do the following commands do? a. `cat /usr/share/doc/cron/README | grep -i edit`

Output:

```
# * documentation (don't take credit for my work), mark your changes (don't
have to go edit a couple of files... So, here's the checklist:
        Edit config.h
        Edit Makefile
```

Explanation: This command combines `cat` and `grep`, using a pipe `|`.

1. `cat /usr/share/doc/cron/README`: The `cat` command reads the file `README` and outputs its content to `stdout`.
2. `|`: The pipe takes the `stdout` of the command on its left (here, the output of `cat`) and passes it as the `stdin` to the command on its right.
3. `grep -i edit`: Searches its input for lines containing a match to the given pattern. The `-i` option makes the search case-insensitive. This command filters the input it receives and only outputs lines that contain the word "edit" in any case.

b.`./out 2>&1 | grep -i eeeee`

No output is produced

Explanation:

1. `./out`: Outputs `O` to `stdout` and Es to `stderr`.
2. `2>&1`: Redirects `stderr` to `stdout`. This means both `O` and `E` characters are sent to `stdout`.
3. `| grep -i eeeee`: The pipe sends the output to `grep`, which searches for the pattern `eeeee` case-insensitively.

Given that the program outputs alternating `O` and `E` characters, this command does not find any match, therefore, it does not produce any output.

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

No output is produced

Explanation:

1. `./out`: Outputs `O` to `stdout` and Es to `stderr`.
2. `2>&1`: Redirects `stderr` to `stdout`.
3. `>/dev/null`: This redirects `stdout` to `/dev/null`, which is a special file that discards all data written to it.
4. `| grep -i eeeee`: Since the `stdout` from `./out` is redirected to `/dev/null`, `grep` receives no input. Consequently, this command produce no output.

3. Write commands to perform the following tasks:

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

```
ls -laR ~ > /tmp/homefileslist
```

Explanation:

1. `-l`: This option tells `ls` to use a long listing format
2. `-a`: Includes hidden entries (beggining by a `.`)
3. `-R`: Recursively lists subdirectories.
4. `~`: This is a shorthand for the user's home directory.
5. `> /tmp/homefileslist`: Redirects the output of the `ls` command to `/tmp/homefileslist`.

b. 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.

```
ls -a ~ | grep -E "\.(txt|md|pdf)$" > /tmp/homedocumentslist 2>/dev/null
```

Explanation:

1. `ls -a ~`: Lists all files and directories in the user's home directory, including hidden ones.
2. `| grep -E`: Uses `grep` to filter the list. The `-E` option allows the use of extended reg ex.
3. `"\.(txt|md|pdf)$"`: looks for lines that end with `.txt`, `.md`, or `.pdf`. The `\` before the `.` is used to escape the dot. `$` ensures the pattern matches at the end of the line.
4. `> /tmp/homedocumentslist`: Redirects the output (the filtered list of files) to the specified file.
5. `2>/dev/null`: Redirects any error messages to `/dev/null`, discarding them.

Task 2 : Log analysis

In this task you will use command line pipelines to analyse log data of a website. The website of a course at HEIG-VD is hosted on Amazon S3. When a user requests a page from the site or makes another type of access S3 writes a log entry to the log file. The log entry contains information about who made the access, what type of access it was, what page or resource was accessed, etc.

Log files are typically in text format. Each line of the file corresponds to a logentry. A line contains a sequence of fields with values that are separated by a separator character. All lines contain the same sequence of fields.

Download the log file (http://ads.iict.ch/ads_website.log) using curl. The file has been reformatted a bit for this lab (see below). The format of S3 log files is described in detail in the S3 Server Access Log Format.The following is a summary:

- Each time a client sends an HTTP request to an S3 server a line is written to the log. A request is made when students browse the web site, but also when the professor uploads material to the web site.
- Each line has 18 fields. The fields are separated by tabs. (The fields of the original log produced by S3 are separated by spaces, for this lab they have been reformatted to be separated by tabs.)
- The 3rd field contains the time of the access (date and time).
- The 4th field contains the IP address where the request came from.
- The 7th field contains the S3 operation. S3 operations are an extension of the HTTP methods GET, POST, PUT and DELETE.
- The 9th field contains the URI of the request.
- The 10th field contains the HTTP status code of the server's response.
- The 17th field contains the user agent string which identifies the browser used to make the request.

Verify that the fields are indeed separated by tabs by using the xxd command to look at the file (look up `xxd` in the manual).

Answer the following question by using the command line and building a pipeline of commands. You can use ??`cat` , `grep` , `cut` , `tr` , `wc` , `sort` , `uniq` , `head` and `tail`. For each question give the answer and the pipeline you used to arrive at the answer.

1. How many log entries are in the file?

Answer:

There is 2781 lineswc

Command used:

```
wc -l ads_website.log
```

Output:

```
2781 ads_website.log
```

Explanation:

- `wc`: Stands for "word count" but can count lines, words, and characters.
- `-l`: Option that tells `wc` to count only lines.
- `ads_website.log`: The name of the file you're counting lines in. Replace `filename` with the actual file name.

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

Answer:

There is 1615 successful access and 34 “Not Found” errors.

Commands used and output:

For status of 200:

```
grep "200" ads_website.log | wc -l
1615
```

For the status 404:

```
grep "404" ads_website.log | wc -l
34
```

Explanation:

— `grep`: Searches for text within files. It matches lines containing the specified pattern.
— `"xxx"`: The text pattern being searched for, in this case, HTTP status code 200 or 404..
— `ads_website.log`: The name of your log file.
— `|`: A pipe that passes the output of one command (the lines containing " 200 ") as input to another command.
— `wc -l`: Counts the number of lines received from `grep`, indicating the number of successful accesses.

3. 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.

Output and Answer:

```
"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"
"PUT /assets%2Fnavigation%2Fbg_logo.png HTTP/1.1"
"GET /heigvd-ads?cors HTTP/1.1"
"GET /heigvd-ads?policy HTTP/1.1"
"DELETE /heigvd-ads/output/assets/navigation/bg_section_presentation_banner2.png HTTP/1.1"
"GET /heigvd-ads?website HTTP/1.1"
"DELETE /heigvd-ads/output/assets/navigation/breadcrumbs-left-bg.png HTTP/1.1"
"GET /heigvd-ads?lifecycle HTTP/1.1"
"GET /favicon.ico HTTP/1.1"
"PUT /readings%2FIcon%0D HTTP/1.1"
"GET /labs/index.html HTTP/1.1"
"GET /labs%2Fprepare%20ssh.html?acl HTTP/1.1"
"GET /favicon.ico HTTP/1.1"
"GET /assets/navigation/style_navigation.css HTTP/1.1"
"GET /assets/navigation/breadcrumbs-left-bg.png HTTP/1.1"
"GET /labs/index.html HTTP/1.1"
"GET /assets/navigation/breadcrumbs-right-bg.png HTTP/1.1"
"GET /assets/navigation/style_navigation.css 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"
```

Commands used:

```
grep "404" ads_website.log | cut -f9
```

Explanation:

- Before the `|` : same than last question.
- `cut -f9`: This command is used to select fields from each line of input. The `-f9` option specifies that you want only the ninth field from each line. In the context of your log files, assuming they follow a standard format where fields are separated by spaces or tabs, the ninth field typically contains the URI of the request.

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

Answer:

There is 21 differents days.

Commands used and output:

```
cut -f3 filename | cut -d'[' -f2 | cut -d':' -f1 | sort | uniq | wc -l
21
```

Explanation:

- `cut -f3 filename`: This command selects the third field from each line in your log file, which typically contains the date and time of the access. Replace `filename` with the actual name of your log file.
- `|`: The pipe character passes the output from one command to the next.
- `cut -d'[' -f2`: This further cuts the extracted date and time field, using the '[' character as a delimiter. The `-f2` selects the part after the first '[' character, which should be the date and time.
- `cut -d':' -f1`: Since the date and time are typically formatted like `[date:time]`, this command uses ':' as the delimiter and selects the date part only.
- `sort`: This command sorts the dates. Sorting is necessary before using `uniq` because `uniq` requires adjacent matching lines to count them only once.
- `uniq`: This removes duplicate lines; in this context, it removes duplicate dates.
- `wc -l`: Finally, this counts the number of unique dates, which represents the number of different days for which there are log entries in the file.

5. How many accesses were there on 4th March 2021?

Answer:

There was 423 accesses on March 4, 2021.

Commands used and output:

```
grep "04/Mar/2021" ads_website.log | wc -l
423
```

Explanation:

- `grep "04/Mar/2021" filename`: This command searches through your log file (`filename`) for entries that contain the date "04/Mar/2021". Replace `filename` with the actual name of your log file.
- `|`: The pipe passes the output from the `grep` command, which in this case is the lines containing "04/Mar/2021", to the next command.
- `wc -l`: This part of the command counts the number of lines that the `grep` command found, which corresponds to the number of accesses on March 4, 2021.

6. Which are the three days with the most accesses? Hint: Create first a pipeline that produces a list of dates preceded by the count of log entries on that date.

Answer:

The three days with the most accesses are :

- March 13, 2021
- March 06, 2021

— March 04, 2021

Commands used:

```
cut -f3 ads_website.log | cut -d'[' -f2 | cut -d':' -f1 | sort | uniq -c | sort -nr | head -3
```

Output:

```
898 13/Mar/2021
580 06/Mar/2021
423 04/Mar/2021
```

Explanation:

- `cut -f3 ads_website.log`: This extracts the third field from each line in the log file, which typically contains the timestamp including the date.
- `|`: This pipe symbol passes the output from one command as input into the next command.
- `cut -d'[' -f2`: This cuts the extracted timestamp field further, using the '[' character as a delimiter. The `-f2` option selects the part of the timestamp that contains the date.
- `cut -d':' -f1`: This uses ':' as the delimiter to cut the date and time information, selecting just the date part.
- `sort`: This command sorts the dates, which is necessary before using `uniq` because `uniq` requires sorted input to effectively count all unique lines.
- `uniq -c`: This removes duplicate dates but also prefixes each unique date with the count of occurrences, effectively giving you the number of accesses per day.
- `sort -nr`: This sorts the results numerically in reverse order, putting the days with the most accesses at the top.
- `head -3`: Finally, this displays just the top three lines from the sorted list, which represent the three days with the most accesses.

7. Which is the user agent string with the most accesses?

Answer:

The user agent string with the most accesses is `Mozilla/5.0 (Windows NT 6.3; WOW64; rv:27.0) Gecko/20100101 Firefox/27.0`.

Commands used:

```
cut -f17 ads_website.log | sort | uniq -c | sort -nr | head -1
```

Output:

```
423 "Mozilla/5.0 (Windows NT 6.3; WOW64; rv:27.0) Gecko/20100101 Firefox/27.0"
```

Explanation:

- `cut -f17 ads_website.log`: This command selects the 17th field from each line in your log file, which is expected to contain the user agent string.
- `|`: The pipe passes the output from one command to the next.
- `sort`: This sorts the user agent strings. This step is necessary before using `uniq` because `uniq` only removes adjacent duplicate lines.
- `uniq -c`: This command filters out duplicate lines, but with the `-c` option, it also counts occurrences of each unique line (in this case, each unique user agent string).
- `sort -nr`: This sorts the results numerically and in reverse order, so the most common user agent strings appear at the top.
- `head -1`: Finally, this command takes the top line from the sorted list, which represents the user agent string with the most accesses

8. 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)?

Answer:

There is 1751 asks made from browsers running on Windows, 180 on Linux and 693 on Mac OS X.

Commands used and outpt:

For Windows :

```
grep -i "Windows" ads_website.log | wc -l
1751
```

For Linux :

```
grep -i "Linux" ads_website.log | wc -l
180
```

For Mac OS X:

```
grep -i "Windows" ads_website.log | wc -l
693
```

Explanation:

- `grep -i "xxxxx" ads_website.log`: This command looks for lines in the log file (`ads_website.log`) that contain the word "xxxxxx" replace by "Windows", "Linux" or "Mac OS X", ignoring case (`-i` for case-insensitive). This is used to identify accesses from Windows browsers.
- `wc -l`: This counts the number of lines returned by the `grep` command, which corresponds to the number of accesses from Windows.

9. 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).

Command used:

```
cut -f17 ads_website.log | tee -a useragents.txt
```

see useragents

Explanation:

- `cut -f17 ads_website.log`: This portion uses the `cut` command to select the 17th field from each line in the file `ads_website.log`. Fields are assumed to be separated by tabs (the default delimiter for `cut`), meaning it's extracting the 17th piece of data from each line, which could be something like a user agent string if this is structured log data.
- `|`: This is a pipe. It takes the output from the command on the left (`cut`) and passes it as input to the command on the right (`tee`).
- `tee -a useragents.txt`: The `tee` command reads from standard input and writes to standard output and to a file. The `-a` option means it appends the output to the end of `useragents.txt` rather than overwriting the file. This means that the 17th field from each line of `ads_website.log` (likely user agent strings if the log format is consistent) will be added to the `useragents.txt` file while also being shown in the terminal.

As mentioned previously, the log you are analysing in this task was reformatted so that the fields are separated by tabs. A normal web server log typically uses spaces. You can see an example of such a log in the file access.log access.log.

10. 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?

Analyzing `access.log` is difficult primarily because it uses spaces as separators. This complicates field separation since values like user agent strings also contain spaces. Without a unique delimiter like tabs, extracting specific fields accurately becomes more challenging, making automated analysis less straightforward and more error-prone.

Task 3 : Conversion to CSV

In this task you will use the command line to summarize the log entries and convert the summary into a CSV file that can be read by a spreadsheet program. A CSV (Comma-Separated Values) file is a file containing a table in text format. It starts with a line containing the column names. Each name is speparated by a comma ,. The remaining lines contain the rows of the table, one line per row. Each line contains the cells of the corresponding row, again separated by commas. Here is an example of a CSV file:

```
Element, Atomic Mass
H, 1.008
He, 4.002602
Li, 6.94
Be, 9.0121831
```

Note: Spreadsheet software typically tolerates spaces between the values and the commas and removes them. In a spreadsheet the order of columns is not important. Depending on the language of your computer, your spreadsheet may use another character than comma , to separate the columns, for example Excel in French expects semicolon ;.

Produce a CSV file named accesses.csv that contains for each day (given by its date)the number of accesses on that day. Transfer that file to your workstation and use spreadsheet software to import the CSV file. Plot the data in a graph and produce a file named accesses.pdf .

Notes :

- Make sure the spreadsheet software correctly interprets the date fields as dates and not as text.
- The dates in the file will not be continuous, i.e. there are days without any accesses which will not appear in the file. Choose a type of plot appropriate for this case.

Command used:

```
grep -o '\[[0-9]*/[A-Z][a-z]*/[0-9]*:[0-9]*:[0-9]*' ads_website.log | cut -d'/' -f1-2 | sort |
↪  uniq -c | tr -d  '[' | sed -e 's/^ *//;s/ /,/'> accesses.csv
```

Output: See accesses

Explanation:

- `grep -o` only print the part captured in the regex. The regex grabs what should be an IP address and a timestamp.
- `cut -d'/' -f1-2 | sort` This step isolates the IP address and the first directory section of the path, assuming the pattern matched an IP address. Sort will sort the data alphabetically.
- `uniq -c | tr -d '['`This step generates a list with each unique combination followed by its count, representing the number of times that specific IP accessed that directory section.
- `sed -e 's/^ //;s/ /,/'> accesses.csv`This final step formats the data into a comma-separated format (CSV) suitable for import into spreadsheets or further analysis. The leading spaces and spaces between IP and count are removed, and commas are inserted to create a CSV file.

To create the csv, I used python. it can be easily run by doing:

```
python3 script.py
```