Unix-Like Data Processing Utilities

Oleksandr Shturmov oleks@oleks.info
Morten Brøns-Pedersen f@ntast.dk
Troels Henriksen athas@sigkill.dk

August 18, 2016

Contents

Operating Systems

An *operating system* stands between the programs you are running, e.g. Chrome, Firefox, or Adobe Acrobat Reader, and the physical hardware packed in your computer. An operating system makes sure that the few hardware resources you have in store, are fairly shared among your programs. It also makes sure that programs don't unintentionally get in the way of one another, and in the end permits you to run multiple programs, simultaneously on one computer.

Operating systems first appeared in the 1960s in response to high demand for computational power. At the time, different users (students, researchers, and staff) wanted to use the big clunky machines filling their basements for many different tasks. Some tasks took longer than others. Some tasks demanded more memory, or disk space than others. Early operating systems made sure to share those resources fairly among the users of what was often, one computer.

As computers evolved and personal computers emerged, the focus shifted from supporting multiple users to supporting multiple programs on one computer. Introvert personal computers quickly proved unproductive and boring: The Internet emerged to connect these marvelous machines into a World-Wide Web of raw computational power, where your operating system now also mediates your communication with the dangerous world that's out there.

This enabled the rich desktop, laptop, and handheld devices that we have today.

Unix-Like Operating Systems

UNIX® is a trademark of The Open Group. They certify which operating systems conform to their Single UNIX Specification.

Colloquially however, "Unix" refers to a family of operating systems developed at Bell Labs in the 1970s, and their countless descendants. Modern descendants

are better called "Unix-like" operating systems. You might be familiar with such Unix-like operating systems as Linux, FreeBSD, OpenBSD, OS X, and iOS. Most notoriously, Microsoft Windows is *not* a Unix-like operating system.

A 1982 Bell Labs video, recently made available under the AT&T Archives, starring such pioneers as Ken Thompson, Dennis Ritchie, and Brian Kernighan, gives further insight into what the original UNIX systems were like, and the philosophy and history behind them.

An important aspect of the history of UNIX is that it has always been guided by the needs of its users. This goal, quite incidentally, results in a philosophy:

> Even though the UNIX system introduces a number of innovative programs and techniques, no single program or idea makes it work well. Instead, what makes it effective is an approach to programming, a philosophy of using the computer. Although that philosophy can't be written down in a single sentence, at its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves. Many UNIX programs do quite trivial tasks in isolation, but, combined with other programs, become general and useful tools.
>
> *Brian Kernigan and Rob Pike. "The UNIX Programming Environment".Prentice Hall, 1984*

The ultimate purpose of this document is to introduce you to this philosophy of using the computer. This philosophy is likely to be different from what you are accustomed to; and yet, it stands on the shoulders of many humble pioneers of modern operating systems.

The reason that this way of using the computer is not in wide-spread adoption is perhaps due to:

1. a general public disinterest in computer programming,
2. the general public fondness of graphical user interfaces , and
3. the resulting sectarianism of those who practice what we preach.

Last, but not leat, many aspects of a Unix-like operating system are ultimately about *freedom*: the freedom to chose how to set up your computer. That is "free" as in free will, and not as in free beer. The price you pay for this freedom is sometimes your patience and your time.

Logging In On Our Linux Box

For Windows users we recommend installing PuTTY. For OS X and Linux users, we recommend installing Mosh. Mosh is also available as a Chrome Extension, or an Android App.

Shell 101

Dear to the heart of a Unix-like operating system is a command-line interface with the operating system, often referred to as a "shell" or "terminal".

A Command-Line Interface (CLI) interprets textual commands, rather than e.g. mouse clicks, or gestures. A CLI presents you with a line where you can enter text, and "prompts" you to enter a command.

You can then type away at the keyboard, and hit the Enter key to "enter" the command. The CLI responds by trying to understand your command, and if the command makes sense, by executing it. This execution may, or may not have a directly observable effect. If the execution terminates, you will be presented with another prompt, prompting for the next command.

When first logging in on a headless Linux box, you are greeted with a welcome message and a prompt:

```
Welcome to syrakuse.
Happy hacking!
alis@syrakuse:~$
```

In place of `alis` however, you will see the username you logged in with.

Try pressing enter a couple times,

```
Welcome to syrakyse.
Happy hacking!
alis@syrakuse:~$
alis@syrakuse:~$
alis@syrakuse:~$
alis@syrakuse:~$
```

This is how you "enter" commands into your computer.

The empty command is a little silly, so let's try something (slightly) less silly:

```
alis@syrakuse:~$ 42
-bash: 42: command not found
alis@syrakuse:~$ hello
-bash: hello: command not found
alis@syrakuse:~$ cowsay hello
 _______
< hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

```
alis@syrakuse:~$
```

`bash` is the program that interprets your commands. `42` and `hello` are not comands that this Linux box understands, but it understands `cowsay`. `cowsay` is a classic, silly little game we've put on our Linux box.

The Prompt

The line

```
alis@syrakuse:~$
```

is called a "prompt". This prompt shows the username you logged in with (in this case, `alis`), the hostname of the machine you logged in on (in this case, `syrakuse`), and the working directory of your shell (in this case, `~`).

In the remainder of this document, we won't point out the user and hostname, as all our examples will use the use the same user and hostname. So our promopts will start out like this:

```
~$
```

Working Directory

All major operating systems (both Unix-like and Windows) are centered around the idea of the user working with directories and files: a *file system*. In practice, this has two aspects:

1. every user has a some sort of a *home directory*, and
2. every program is working from the point of view of a particular directory: its *working directory*.

When you login on our Linux box, you land in a `bash` program, whose working directory is the home directory of the user you logged in with. The user's home directory has the alias `~` on Unix-like systems.

The following sections show how you can figure out what your working directory is, how to navigate the file system, create new directories, and change the working directory of `bash`.

`pwd`, `mkdir` and `ls`

`pwd` prints the working directory:

```
~$ pwd
/home/alis
~$
```

In Unix-like operating systems, file system paths are separated by `/` (forward slash). (In Windows, they are separated by \ (backward slash).) Furthermore, in Unix-likes systems, unlike in Windows, all directories and files reside in one file system, starting at `/`. This is called the *root directory*.

In this case, we see that the root directory has one subdirectory `home`, which has a sub-subdirectory `alis`. This is `alis`'s home directory. So `~` (in `alis`'s case) is really an alias for `/home/alis`. Let's make a mess of our home directory:

`mkdir` can create new directories.

```
~$ mkdir Andromeda
~$ mkdir Corvus Crux Lynx
~$ mkdir "Ursa Major" "Ursa Minor"
```

`ls` can list directory contents

```
~$ ls
Andromeda  Corvus  Crux  Lynx  Ursa Major  Ursa Minor
```

This is a little cryptic. How many directories do we actually have? To see this, you can pass the argument `-lh` to `ls`. The `l` (small L) *flag* asks for a long, and the `h` flag asks for a human-readable listing format.

```
~$ ls -lh
total 24K
drwxr-xr-x 2 alis alis 4.0K Sep 21 13:37 Andromeda
drwxr-xr-x 2 alis alis 4.0K Sep 21 13:37 Corvus
drwxr-xr-x 2 alis alis 4.0K Sep 21 13:37 Crux
drwxr-xr-x 2 alis alis 4.0K Sep 21 13:37 Lynx
drwxr-xr-x 2 alis alis 4.0K Sep 21 13:37 Ursa Major
drwxr-xr-x 2 alis alis 4.0K Sep 21 13:37 Ursa Minor
```

`ls -lh` lists a whole bunch of details you probably don't even want to care about (for now). What's important is that it lists one directory, or file, per line. Counting the lines, we see that the above `mkdir` commands created 6 directories. We can now see how the `mkdir` command works with `bash`:

```
~$ mkdir Corvus Crux Lynx
```

Created 3 directories: `Corvus`, `Crux`, and `Lynx`: `bash` splits command arguments at one or more space characters.

If you want to put a space in your argument, e.g. to create a directory with spaces in its name, you must put the argument in quotes, as we did with the following `mkdir` command:

```
~$ mkdir "Ursa Major" "Ursa Minor"
```

This created the two directories `Ursa Major` and `Ursa Minor`.

Another way to avoid `bash` splitting up your argument is to *escape* the spaces. Escaping is done by putting a special \ (back slash) in front of the character

you want "escaped". Some other characters that we often escape are `"` (double quote), and `\` (backslash) itself. The following `mkdir` command is equivalent to the above:

```
~$ mkdir Ursa\ Major Ursa\ Minor
```

By default, `ls` lists the contents of the working directory. It can also be used to list the contents of other directories.

Try typing `ls A` and press TAB. `bash` will expand this to `ls Andromeda/` (the trailing `/` indicates that `Andromeda` is a directory):

```
~$ ls Andromeda/
~$
```

`bash` is good friends with the working directory, and your file system in general. When specifying a path, it can see the files and directories in your system, and help you type them out quickly and correctly.

Try typing `ls U` and press TAB. `bash` will expand this to `ls Ursa\ M` but stop there, as it doesn't whether you mean `Ursa\ Major` or `Ursa\ Minor`. Continue by tying `a` and press TAB again. `bash` will now complete the name for you to `ls Ursa\ Major/`.

NB! Unfortunately, `bash` is not good friends with your programs, and it won't help you remember cryptic arguments like `-lh`.

The `Andromeda` directories contains no files or directories. We can use `mkdir` to create some subdirectories in `Andromeda` (remember TAB!):

```
~$ mkdir Andromeda/HAT-P-6 Andromeda/WASP-1
~$ ls -lh Andromeda
total 8.0K
drwxr-xr-x 2 alis alis 4.0K Sep 21 13:37 HAT-P-6
drwxr-xr-x 2 alis alis 4.0K Sep 21 13:37 WASP-1
```

Exercises

1. List the contents of the `/` directory.
2. List the contents of the `/home/` directory.
3. Create the directory `~/1/1/2/3/5/8/13/21`. (Hint: press UP-arrow to retype last command in `bash`.)
4. Create the directory `"hello,\ bash"` in your home directory.

NB! `ls -lh` should show something like this:

```
~$ ls -lh
...
drwxr-xr-x 2 alis alis 4.0K Sep 21 13:37 "hello,\ bash"
...
```

cd

`cd` changes the current working directory.

```
~$ cd Andromeda
~/Andromeda# ls -lh
total 8.0K
drwxr-xr-x 2 alis alis 4.0K Sep 21 13:37 HAT-P-6
drwxr-xr-x 2 alis alis 4.0K Sep 21 13:37 WASP-1
~/Andromeda#
```

The `bash` prompt will comfortably let you know where you stand, so `pwd` is no long necessary, as long as you know what `~` and `/` mean.

`cd` with no arguments will lead you back home:

```
~/Andromeda# cd
~$
```

Of course, you can also use the path alias `~`:

```
~$ cd Andromeda
~/Andromeda# cd ~
~$
```

Let's go deeper:

```
~$ cd Andromeda/HAT-P-6/
~/Andromeda/HAT-P-6#
```

Every directory has a special directory `..`

If you `cd` to `..` you go "up" a directory in the file system hierarchy. We say that `..` refers to the *parent directory*. So here's the long way home:

```
~/Andromeda/HAT-P-6# cd ../../
~$
```

If every directory has a special directory `..` then how come we didn't see it in our directory listings above? This is because Unix-like convention has it that hidden files and directories start with a `.` (dot). Of course, there is nothing overly special about "hidden" files (on Unix-like systems or Windows). We can get `ls` to show *all* the directories in a directory using an additional `a` flag:

```
~$ ls -lah
total 40K
drwxr-xr-x 10 alis alis 4.0K Sep 21 13:37 .
drwxr-xr-x  3 alis alis 4.0K Sep 21 13:37 ..
drwxr-xr-x  3 alis alis 4.0K Sep 21 13:37 1
drwxr-xr-x  4 alis alis 4.0K Sep 21 13:37 Andromeda
drwxr-xr-x  2 alis alis 4.0K Sep 21 13:37 Corvus
drwxr-xr-x  2 alis alis 4.0K Sep 21 13:37 Crux
```

```
drwxr-xr-x  2 alis alis 4.0K Sep 21 13:37 "hello,\ bash"
drwxr-xr-x  2 alis alis 4.0K Sep 21 13:37 Lynx
drwxr-xr-x  2 alis alis 4.0K Sep 21 13:37 Ursa Major
drwxr-xr-x  2 alis alis 4.0K Sep 21 13:37 Ursa Minor
~$
```

`.` is also special directory, it refers to the *current directory*.

Exercises

1. `cd ~/../../../../../../` Where do you end up?
2. Go back home.
3. `cd ~/./../alis/Andromeda/./HAT-P-6/../../` Where do you end up?
4. Create a hidden directory in your home directory.

`echo`, `cat`, and `>`

`echo` is a program that can display a line of text. For instance:

```
~$ echo "Roses are red,"
Roses are red,
~$
```

Note, again how arguments coontaining spaces are surrounded by double quotes.

At first sight, this is a rather useless program. This is where the power of a Unix-like operating system comes into play. `bash` can *redirect* the output of a program to a file. To do this, follow the command with the character `>`, followed by a path to the file where you want the output.

```
~$ echo "Roses are red," > roses.txt
~$
```

We can use `ls` to check to see what happened:

```
~$ ls -lah
...
drwxr-xr-x  2 alis alis 4.0K Sep 21 13:37 "hello,\ bash"
-rw-r--r--  1 alis alis   15 Sep 21 13:37 roses.txt
drwxr-xr-x  2 alis alis 4.0K Sep 21 13:37 Lynx
...
~$
```

So now there is something called `roses.txt` in our home directory. Unlike the directories we created before, the left-most character printed by `ls` is a `-` (dash), rather than a `d`. This indicates that `roses.txt` is *not* a directory.

`cd` can help us verify this:

```
~$ cd roses.txt
bash: cd: roses.txt: Not a directory
```

> **TIP** If you want `ls -lah` to *just* list the file that you are looking for, specify the file to `ls -lah`:
>
> ```
> ~$ ls -lah roses.txt
> -rw-r--r--  1 alis alis   15 Sep 21 13:37 roses.txt
> ```

`cat` is a program that can print the contents of a file back to `bash`:

```
~$ cat roses.txt
Roses are red,
~$
```

Let's create another file:

```
~$ echo "Violets are blue," > violets.txt
~$
```

> **NB** `>` will overwrite the file if it already exists.

`cat` can also *concatenate* files, and print the contents back to to `bash`:

```
~$ cat roses.txt violets.txt
Roses are red,
Violets are blue,
~$
```

Of course, `bash` can redirect the output of any command, so we can store this, more complete poem in `poem.txt`.

```
~$ cat roses.txt violets.txt > poem.txt
~$ cat poem.txt
Roses are red,
Violets are blue,
~$
```

Exercises

1. Create a file `sugar.txt` with the line `Sugar is sweet,`
2. Create a file `you.txt` with the line `And so are you.`
3. Complete the poem in `poem.txt` by combining `roses.txt`, `violets.txt`, `sugar.txt`, and `you.txt` (in that order).

It should be possible to do this in the end:

```
~$ cat poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
```

```
And so are you.
~$ cat violets.txt you.txt
Violets are blue,
And so are you.
~$
```

`wc` and `|`

`wc` prints the line, word, and byte count of a file.

```
~$ wc poem.txt
 4 13 65 poem.txt
~$
```

So our poem is 4 lines, 13 words, or 65 bytes in length.

> **TIP** This explains the `65` in the output of `ls -lah` for `poem.txt`:
>
> ```
> ~$ ls -lah poem.txt
> -rw-r--r-- 1 alis alis 65 Sep 21 13:37 poem.txt
> ```

The name `wc` is easy to remember if you think that it stands for "word count", although the program can do fair a bit more than that. In fact, it isn't even that good at counting words. Any sequence of non-whitespace characters is counted as a word. For instance, numbers are also counted as words. Your high-school teacher would not be happy with such a word count.

> **TIP** If you *just* want the line count for a file, use the `-l` option.
>
> ```
> ~$ wc -l poem.txt
> 4 poem.txt
> ~$
> ```

What if we wanted a word count of the silly poem we had above?

```
~$ cat violets.txt you.txt
Violets are blue,
And so are you.
~$
```

We could use `bash` redirection to put the silly poem in a silly file, pass the filename to `wc`, and finally remove the silly file (more on this below); but this would be rather silly: Why create a file in the first place?

`bash` can *pipe* the output of one program as input to another. To do this, type the first command, a `|` (pipe) character, then the second command:

```
~$ cat violets.txt you.txt | wc
      2       7      34
```

`wc` with no arguments, counts the lines, words, and bytes passed to it from `bash`. Now `wc` does not print a file name: There is no filename to print!

The silly poem is just 2 lines, 7 words, or 34 bytes in length.

Let's verify that this "pipe"-thing works by doing this with `poem.txt`:

```
~$ cat poem.txt | wc
      4      13      65
```

Except for the silly whitespace (more on how to handle this later), and the missing filename, the output is the same as with `wc poem.txt` above.

Exercises

1. Count the number of files and directories directly under the `/` directory.

`rm`

By now we've made a great big mess of our home directory. It is time to clean up a little bit. The `rm` command can be used to quickly delete files and folders.

Use `rm` to delete files:

```
~$ rm poem.txt
~$
```

We can use `ls` to verify that `poem.txt` indeed is gone:

```
~$ ls poem.txt
ls: cannot access poem.txt: No such file or directory
```

If you want to delete directories, you will need to specify the argument `-r`, where the `r` flag stands for "recursive".

```
~$ rm -r 1/
```

If you want to delete all the files and directories in a directory, but the directory itself, you can use a wildcard `*` after the directory name:

```
~$ rm -rf Andromeda/*
~$ ls -lah Andromeda/
total 8.0K
drwxr-xr-x  2 alis alis 4.0K Sep 19 14:58 .
drwxr-xr-x 10 alis alis 4.0K Sep 19 14:38 ..
~$
```

> **NB** `rm` does not throw your files and directories into some easily-accessible "garbage bin". Although things are not always permanently and securely deleted with `rm`, recovering them is a lofty task. Take care of what your remove.

Exercises

1. Remove all the files and directories in your home directory which you don't want to keep for future reference. (Don't remove `poem.txt`, we will use it shortly.)

Text-Editors

Although we can read and write files using our various command-line utilities, in conjunction with clever shell tricks, this mode of operation can get a little cumbersome. Text-editors are dedicated utilities to this end.

There are two classical text-editors in a Unix-like environment: `vim` and `emacs`. The users of one are often viciously dispassionate about the users of the other. More humble users use whatever suits the task at hand. For instance, `vi` (a slimmer, older version of `vim`) is available on most systems out of the box, and so `vim` proliferates in server environments, while `emacs` has often been a tool you need to explcitly install, making it more suitable in a desktop environment.

Another text editor available on many systems out of the box is `nano`. To avoid duels over the choice of text-editor, and still teach you a somewhat ubiquitous tool, we decided to focus on `nano`. To start editting our `poem.txt` with `nano`:

```
~$ nano poem.txt
```

`nano` uses a so-called "text-based user interface" (TUI), which is quite reminiscent of a graphical user interface, except that a user interacts with it by issuing commands and/or using keyboard shortcuts instead of relying on mouse clicks and/or gestures.

At the bottom of the TUI, `nano` shows a short reference of useful shortcuts:

```
  ^G Get Help      ^O Write Out     ^W Where Is     ...
  ^X Exit          ^R Read File     ^\ Replace      ...
```

Here, `^` indicates the Ctrl character. For instance, type Ctrl+o to save (i.e., "write out") the file you are editing. `nano` will now prompt you:

```
  File Name to Write: poem.txt
```

Type Enter to confirm and overwrite `poem.txt`. To exit `nano`, type Ctrl+x.

`vim` also uses a TUI by default, while `emacs` can be started in this mode with the command-line argument `-nw` (i.e., no window system). So `vim` and `emacs` can be used to similarly edit `poem.txt`, but they are far less friendly to beginners.

A typical problem that beginning users have is how exit either `vim` or `emacs` once they open them. In `vim`, you can press Esc to enter a so-called "command mode", enter the command `:q` and press Enter. In `emacs`, you use the keyboard

sequence Ctrl+x, Ctrl+c. Figuring out how to edit and save files in either `vim` or `emacs` is left as an exercise for the bored reader. Else, continue with `nano`.

Exercises

1. Open `poem.txt` in `nano`.
2. Cut out the first line and paste it (uncut) at the bottom of the poem. Save the new file.
3. Determine the number of lines and characters in the poem using `nano`. How many characters are there in the longest line?
4. Copy the entire poem beneath itself without doing this line-by-line. Hint: use "Read File".

`mv` and `&&`

`mv` can be used to move/rename files. It takes at least two command-line arguments, the source file path and a target file path. If you supply more than 2 arguments, all files listed before the last argument will be moved to the folder designated by the last argument.

`&&` can be used to chain commands together such that the second is executed if and only if, the first one succeeds.

```
mv poem.txt poem.txt.long && mv poem.txt.long poem.txt
```

Exercises

1. Create the directories `Andromeda`, `Corvus`, `Crux`, and `Lynx` in your home directory. Create a directory `Constellations` and move all the aforementioned directories into this one.
2. Remove the folder `Constallations`.
3. Do exercises 3 and 4 above (relating to `nano`) using `wc`, `cat`, `>`, `&&`, and `mv`. OBS! You can't read and write the same file in the same command. For task 4, you will need to use a temporary file.

Shell Scripts 101

Composing small utilities to form complicated commands is fun, but it is also hard work. We can save our work by encoding a command into a so-called shell script — a file containing shell commands. In effect, we are creating a utility of our own.

Let's walk through creating a shell-script for doubling the contents of a file.

First, open a file `double.sh` in `nano`. The `.sh` extension follows the convention that shell scripts should have the filename extensions `.sh`, although this does not really make it a "shell script".

`double.sh`, as a command-line utility, will take a command line argument (`$1`), regard it as a path to an existing file, `cat` this file twice into a temporary file (`$1.double`), and if this succeeds, move `$1.double` to `$1`, replacing the original file. Here, `$1` is a shell variable referring to the first command-line argument. If no such argument is given, `$1` is an empty string.

Write the following to `double.sh` using `nano`:

```
cat $1 $1 > $1.double && mv $1.double $1
```

Now, to run this shell script, pass it as an argument to the program `bash`:

```
~$ wc -l poem.txt
       4 poem.txt
~$ bash double.sh poem.txt
~$ wc -l poem.txt
       8 poem.txt
```

`double.sh` is still far from a conventional command-line utility. It is a lot of work typing `bash double.sh` instead of just `double`. We will come back to how you can do this later.

More Data For Your Shell

The world would be pretty borring if all you could do was write poems and mess about with your files and directories. It is time to get on the Internet!

`curl`

`curl` can fetch a URL and write the fetched data to your shell.

If you are reading this, you have already made your way to our server, https://uldp16.onlineta.org.

This machine runs a web server that exposes its access log to the world. Of course, everyone on the Internet can make all sorts of HTTP requests to this web server.

Let's try to fetch the access log:

```
~$ curl https://uldp16.onlineta.org/access.log.csv.txt
...
...GET /phpMyAdmin/scripts/setup.php HTTP/1.1"-
```

```
...
...GET /admin/basic HTTP/1.1"libwww-perl/6.05
...
...Googlebot/2.1; +http://www.google.com/bot.html)
...
...GET /Ringing.at.your.dorbell!...x00_-gawa.sa.pilipinas.2015
...
~$
```

> **TIP** Use Shift + PgUp to scroll up, and Shift + PgDn to scroll down in your shell.

If you `curl` the access log one more time, at the bottom of the file you will see a line ending with something like

```
...GET /access.log.csv.txt HTTP/1.1"curl/7.38.0
```

That's you!

Remember the dangerous world that's out there? This is it! This is what happens when you expose your computer to the Internet. All sorts of adventerous crawlers try to creep in on you.

Exercises

1. Make a local, snapshot copy of the access log. Save the log as `access.log` in your home directory.

`head` and `tail`

When you have a pretty long file, such as a web server access log, you sometimes want to just get a glance of the start of the file, or the end of the file.

`head` can be used to output the first part of a file:

```
~$ head access.log
... (first 10 lines of access.log)
~$
```

`tail` can be used to output the last part of a file:

```
~$ tail access.log
... (last 10 lines of access.log)
~$
```

For both `head` and `tail` you can specify the number of first or last lines to print, using the `-n` option:

```
~$ head -n 20 access.log
... (first 20 lines of access.log)
~$ tail -n 20 access.log
... (last 20 lines of access.log)
~$
```

cut

`cut` is a utility that can cut up the lines of a file.

You may have noticed the silly `"` (double quotes) in the log file. They separate the 4 fields of the log file:

1. IP (version 4) address of the requesting party.
2. Timestamp of request.
3. A description of the request.
4. A so-called *browser string*: What the requesting party otherwise tells about itself: Typically some sort of a browser, program, or crawler name.

This type of file is typically called a CSV-file. CSV stands for comma-separated values. You are probably familiar with Microsoft Excel, Google Sheets, Apple Numbers, or LibreOffice Calc. A CSV file maps naturally to a "spreadsheet": each row is a line in the file, with the columns separated by a "comma".

The name CSV is unfortunate. Oftentimes, the separator is a semi-colon (`;`), and in our case a double quote (`"`). People use whatever separates the columns best (you will soon see how to do this). We use a double quote because double quotes can't occur in either a URL or a browser string.

To get just the IP addresses of parties that have tried to access the server, we can cut up each line at `"` and select the first field:

```
~$ cat access.log | cut -d\" -f1
...
213.211.253.38
213.211.253.38
130.225.98.193
...
~$
```

You can select multiple fields, but they are always selected by `cut` in increasing order:

```
~$ cat access.log | cut -d\" -f1,3
...
213.211.253.38"x00_-gawa.sa.pilipinas.2015
213.211.253.38"x00_-gawa.sa.pilipinas.2015
130.225.98.193"curl/7.44.0
...
```

```
~$
```

Exercises

1. An IP (version 4) address is composed of 4 fields, each a number between 0 and 255, typically separated by a `.` (dot). List the first field of every IP address in the access log.
2. The server log the time access in ISO 8601 format. List all the dates of the month on which the log was accessed.
3. The HTTP request itself is described in the third column of the log file. It consists of 3 things:
   a. The HTTP Method used. Typically `GET`.
   b. The resource that was requested. This is typically the tail of the URL that the crawler used, i.e. the URL after the leading `https://uldp16.onlineta.org` has been cut off.
   c. The HTTP version used. Should be `HTTP/1.1` or `HTTP/1.0`.

   List all the resources that were requested.

`sort` and `uniq`

`sort` is a utility that can sort the lines of a file and output the sorted result.

```
~$ cat access.log | cut -d\" -f4 | sort
...
curl/7.44.0
...
x00_-gawa.sa.pilipinas.2015
x00_-gawa.sa.pilipinas.2015
...
~$ cat access.log | cut -d\" -f1 | sort -n
...
130.225.98.193
213.211.253.38
213.211.253.38
...
~$
```

The `-n` option tells `sort` to sort in *alphanumeric* order rather than *lexicographic* order. For instance, in lexicographic order, `213.211.253.38` comes *before* `36.225.235.94`. In alphanumeric order, it comes *after*.

See also the man page for `sort` for other useful sorting options.

`uniq` is a utility that can remove the immediate duplicates of lines. If you have a sorted file, you can use `uniq` to output the unique lines of the original file.

```
~$ cat access.log | cut -d\" -f1 | sort -n | uniq
...
130.225.98.193
213.211.253.38
...
~$
```

`uniq` also has the rather useful option that it can count the number of duplicate occurrences before it deletes them:

```
~$ cat access.log | cut -d\" -f1 | sort -n | uniq -c
...
     11 130.225.98.193
      4 213.211.253.38
...
~$
```

Exercises

1. List the unique browser strings.
2. Count how many times each browser string occurs.
3. Count the total number of unique IP addresses that have tried to access the server.
4. What are the different HTTP methods that have been used?
5. Count the number of requests made in every day for which the access log has data.
6. Count the number of *unique* requests made in every day for which the access log has data.
7. List the number of requests in each hour of the day (use the same time zone as the server, so just cut out the hour of every entry in the log).

Advanced Data Manipulation

`grep`

Let's say we want to find all the requests that were made using (supposedly) the Google Chrome web browser.

`grep` is a utility that can print the lines matching a pattern.

> **NOTE** `grep` is a slightly old utility, and so we will always use the more modern, *extended* variant of `grep`, be specifying the `-E` option. This is equivalent to using the `egrep` utility instead.

```
~$ cat access.log | grep -E "Chrome"
...
130.225.98.193...Chrome/45.0.2454.85 Safari/537.36
...
~$
```

The "patterns" that you can specify to `grep` are called *regular expressions*. Regular expressions offer a powerful syntax for matching and replacing strings. They deserve a special role in theoretical Computer Science. We will only take a look at some simple, practical elements of regular expressions.

For instance, the Internet Explorer web browser is typically identified by the string `Trident` occurring in the browser string. Here's how you would find all those requests that were made using *either* Google Chrome or Internet Explorer:

```
~$ cat access.log | grep -E "Chrome|Trident"
...
130.225.98.193...Chrome/45.0.2454.85 Safari/537.36
...
104.148.44.191...Windows NT 6.1; Trident/5.0
...
~$
```

(If this doesn't work, i.e. we have no Internet Explorer users in the audience, try `Firefox` instead.)

The character `|` is a regular expression *metacharacter*, and serves to say that the line should contain the string either `Chrome` *or* `Trident`.

The metacharacter `|` can be used multiple times. For instance, the Firefox web browser typically identifies itself with the string `Gecko`. Here is how you would find all the requests made with either Google Chrome, Internet Explorer or Firefox.

```
~$ cat access.log | grep -E "Chrome|Trident|Gecko"
...
130.225.98.193...Chrome/45.0.2454.85 Safari/537.36
...
104.148.44.191...Windows NT 6.1; Trident/5.0
...
109.200.246.88...Gecko/20100101 Firefox/9.0.1
...
~$
```

You can also ask `grep` for the inverse match, i.e. those lines not matching the given pattern using the `-v` option:

```
~$ cat access.log | grep -Ev "Chrome|Trident|Gecko"
...
130.225.98.193...curl/7.44.0
...
213.211.253.38...x00_-gawa.sa.pilipinas.2015
...
122.154.24.254...-
...
~$
```

> **TIP** To get `grep` to match in a *case-insensitive* manner. e.g. to match strings like `chrome` even though the pattern says `Chrome`, use the `-i` option. (An example follows further below.)

A regular expression is a string of characters. Every character is either a metacharacter, or a literal character. For instance, in the regular expression `Chrome`, all characters are literal characters. Here is an overview of a couple useful metacharacters:

Metacharacter	Meaning
`|` (pipe)	Alternation (choice)
`^` (caret)	Start of string
`$` (dollar)	End of string
`.` (dot)	Any character

If you actually want to match what is otherwise a metacharacter, you will need to *escape* it. Similarly to strings in `bash`, this is done by prefixing the metacharacter with the metacharacter `\`. For instance, `\|`, `\^`, `\$`, and `\.`

Consider our little poem from before:

```
~$ cat poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.
~$
```

Here is how we would find all the lines ending with a `.` (dot):

```
~$ cat poem.txt | grep -E "\.$"
And so are you.
```

What if we wanted to find all the strings ending with a `.` (dot) *or* a `,` (comma)? To this end, we could use *character groups*. A character group is a list of characters acceptable at a given position in the string.

Character groups are perhaps best shown by example. All the following regular

expressions match a single digit, i.e. a string that is either `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`:

1. `^[01234569]$`
2. `^[0-9]$`
3. `^[0-56-9]$`

A computer represents characters as numbers. There exists a range of standard ways of encoding characters as numbers, with perhaps the most ubiquitous being the UTF-8 character encoding.

Character ranges are based on these numeric representations of the characters involved. For instance:

1. The range `[a-z]` will contain the lower-case English alphabetic characters.
2. The range `[a-zA-Z]` contains both lower-case and upper-case English alphabetic characters.
3. The range `[a-zæøå]` contains the lower-case Danish alphabetic characters.
4. The range `[a-zæøåA-ZÆØÅ]` contains the lower and upper-case Danish alphabetic characters.
5. The range `[z-a]` contains no characters because the character code for `z` is larger than the character code for `a`.

   TIP If you want to quickly find the character code of a particular character, right-click in your Internet browser, e.g. Google Chrome, and bring up the HTML inspector. (Typically called something like "Inspect this element".) In the inspector window, you should be able to find a JavaScript *console*. Enter the following command in the console:

   ```
   > "A".charCodeAt(0)
   65
   ```

   Use any character you'd like in place of `A`.

These regular expressions use a couple new metacharacters:

Metacharacter	Meaning
`[`	Start a character group
`-`	Character range[1]
`]`	End a character group

Getting back to our original problem, here is how we would find all the lines ending with a `.` (dot) *or* a `,` (comma) in our little poem:

[1] The `-` character *only* acts a metacharacter between `[` and `]`, and you don't have to escape it elsewhere.

```
~$ cat poem.txt | grep -E "[\.,]$"
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.
```

How about those lines beginning with a flower?

This requires another little regular expression construct called *capturing groups*. Why these groups are called "capturing" will become clear shortly. For now, let us see how they help us solve our problem: A capturing group allows us to put a regular expression inside a regular expression, and treat it like a character.

For instance, here is how we could find all those lines that begin with a flower (note the use of the case-insensitive option):

```
~$ cat poem.txt | grep -Ei "^(roses|violets)"
Roses are red,
Violets are blue,
~$
```

This leaves us with the following additional metacharacters:

Metacharacter	Meaning
`(`	Start a capturing group
`)`	End a capturing group

Last, but not least, the elements of a regular expression can be quantified, to indicate that some parts of the string repeat a number of times.

Suffix	Meaning
`*`	This repeats 0 or more times
`+`	This repeats 1 or more times
`{` n `}`	This repeats exactly n times
`{` n `,}`	This repeats n or more times
`{` n `,` m `}`	This repeats between n and m times

Here `*`, `+`, `{`, `,`[2], and `}` are all metacharacters.

For instance, to find all the entries in the access log, with the browser string indicating that the requesting party is using Google Chrome on Linux, we could do this:

```
~$ cat access.log | grep -E "Linux.*Chrome"
...
```

[2]The `,` character only acts as a metacharacter between `{` and `}`.

```
...(X11; Linux x86_64)...Chrome/45.0.2454.85 Safari/537.36
...
~$
```

Here we accept any character between the *substrings* `Linux` and `Chrome`, 0 or more times. The result is those strings that contain both `Linux` and `Chrome` (in that order).

> **TIP** The above quantifiers are *greedy*: They will consume all possible occurrences. Sometimes, what you want is to consume up until the first occurrence of the next string. Follow the quantifier with `?` to make it *non-greedy*.

Exercises

1. How many requests were made from Google Chrome?
2. How many requests were made from neither Google Chrome, Internet Explorer, or Firefox?
3. If the visitor does not supply a browser string, our web server writes the browser string `-` (dash) to the log. Find the IP addresses of those visitors that do not supply a browser string.
4. Complete the first 14 lessons on http://regexone.com/. The tool is very permissive, and there are multiple answers which will be deemed "correct". For every lesson, try to come up with as many ways as possible to get the tool mark your answer as "correct".

`sed`

`sed` is a tool for transforming streams of data. Transformation can take place using regular expressions.

> **NOTE** `sed`, like `grep`, is a pretty old tool, so we will also always use the more modern, *extended* variant of `sed`, by specifying the `-E` option.

We can use `sed` for a lot of things (e.g. to play Tetris). We will only use sed for replacing the substrings of every line in a file, matching a regular expression with something else. The general way to call `sed` when doing this is as follows:

`~$ sed -E "s/` regular expression `/` replace expression `/g"`

For instance, we can replace all occurrences of `"` in our log-file with `___`.

```
~$ cat access.log | sed -E "s/\"/___/g"
...
109.200.246.88___2015-09-20T18:37:08-04:00___...
```

```
...
~$
```

The second argument `sed` (e.g. `"s/\"/___/g"`) is a `sed`-command. We are only concerned with search-and-replace commands. These commands start with an `s`, and are followed by a regular expression and a *replace expression*, started, separated, and ended by a `/`. (So `/` is now also a metacharacter, and must otherwise be escaped.) The final `g` signifies that we want to replace *all* matching substrings on every line (mnemonic: `g` for *global*).

A more interesting use of `sed` is using capturing groups. Capturing groups are so-called because the substrings they capture become available in a replace expression. To insert a substring matched in a capturing group, type `\` followed by the number of capturing group in the regular expression, numbered from left to right, starting at 1.

For instance, instead of using `cut`, we could've used `sed` to cut up our file:

```
~$ cat access.log | sed -E "s/^(.*)\"(.*)\"(.*)\"(.*)$/\1/g"
...
93.174.93.218
140.117.68.161
...
~$ cat access.log | sed -E "s/^(.*)\"(.*)\"(.*)\"(.*)$/\2/g"
...
2015-09-20T13:08:52-04:00
2015-09-20T13:52:37-04:00
...
~$
```

Unlike with `cut`, with `sed` we can even reorder the fields!

```
~$ cat access.log | sed -E "s/^(.*)\"(.*)\"(.*)\"(.*)$/\2\"\1/g"
...
2015-09-20T13:52:37-04:00"104.148.44.191
2015-09-20T13:52:38-04:00"104.148.44.191
...
~$
```

Exercises

1. The timestamps in our access log are in ISO 8601 format. List all the unique access dates in the format date-month-year.
2. List all the unique access times in the format hour.minte.
3. Replace the ISO 8601 date and time by the above. Don't change any other fields in the log.

4. Strip all digits and punctuation from the browser string.
5. List the number of requests made from each unique IP. Use following format: IP (two spaces) count. So, strip the leading spaces and swap the columns that you get after `uniq -c`.
6. List the number of requests in each hour of the day (use the same time zone as the server, so just cut out the hour of every entry in the log). Use following format: hour (two spaces) count. So, strip the leading spaces and swap the columns that you get after `uniq -c`.

Shell Scripts

(Preliminary version - finished in next version of the notes?)

One useful aspect of the Unix environment is the ease of turning manual workflows into automatic scripts. If you find yourself typing some specific sequence of commands frequently, just put them in some file `myscript.sh`, and run

```
~$ sh myscript.sh
```

which will cause the contents of the script to be executed just as if you had typed it yourself. (Actually, there are some exotic differences - primary among these is that using `cd` in the script will not have an effect once the script ends.) The `.sh` extension is irrelevant to the operating system - you can call it whatever you want and it will still work.

If you want to make your script feel even more like an ordinary program, there are two things you must do. First, you must add a *shebang* as the first line of the file:

```
#!/bin/sh
```

This line tells the operating system what kind of code is contained in the file. In this case, we tell it to execute it using the shell `/bin/sh`.

You must also mark the file as *executable*, or else the operating system will refuse to start it. This is done with the `chmod` command:

```
~$ chmod +x myscript.sh
```

Now we can launch the script without explicitly invoking `sh`, just by passing the path to the script:

```
~$ ./myscript.sh
```

In this case, the path indicates the current directory, but if the script was somewhere else, we could run it as:

```
~$ ~/scripts/myscript.sh
```

If you want to be able to run the script just by typing `myscript.sh`, you must add its containing directory to the `$PATH` environment variable. This is outside the scope of this document, but you can try using a search engine to figure out how to do it yourself.

Basic Control Flow

Copying commands into a file is a good starting point, but shell scripting first becomes a truly powerful tool when you add *control flow* (conditionals and branches) to your scripts. While much less powerful than a conventional programming language, shell script is a convenient way to automate workflows that involve Unix programs.

First, it is important to understand the notion of an *exit code*. Every program and shell command finishes by returning a number in the interval 0 to 255. By default, this number is not printed, but it is stored in the *shell variable* `$?`. For example, we can attempt to `ls` a file that does not exist:

```
~$ ls /foo/bar
ls: cannot access /foo/bar: No such file or directory
```

To print the exit code of `ls`, we echo the `$?` variable:

```
~$ echo $?
2
```

By convention, an exit code of 0 means "success", and other exit codes are used to indicate errors. In this case, it seems that `ls` uses the exit code `2` to indicate that the specified file does not exist. It may use a different exit code if the file exists, but you do not have permission to view it. Note what happens if we check the `$?` variable again:

```
~$ echo $?
0
```

The exit code is now 0! This is because the variable `$?` is overwritten every time we run a command. In this case, it now contained the exit code of our first `echo` command, which completed succesfully. It is often a good idea to save away the value of `$?` in some other shell variable, as most commands will overwrite `$?` itself.

Typically we are not much concerned with the specific code being returned - we only care whether it is 0 (success) or anything else (failure). This is also how the shell `if-then-else` construct operates. For example:

```
~$ if ls /foo/bar ; then echo 'I could ls it!'; else echo 'I could not. :-('; fi
ls: cannot access /foo/bar: No such file or directory
I could not. :-(
+# if ls /dev/null ; then echo 'I could ls it!'; else echo 'I could not. :-('; fi
```

```
/dev/null
I could ls it!
```

The semicolons are necessary to indicate where the arguments to `echo` stop. We could also use a newline instead.

A Simple Testing Program

Let us try to write a simple practical shell script. We want to create a program, `utest.sh`, that is invoked as follows:

```
~$ sh utest.sh prog input_file output_file
```

`utest.sh` will execute the program `prog` and feed it input from `input_file`. The output from `prog` will be captured and compared to `output_file`, and if it differs, `utest` will complain. Essentially, we are writing a small testing tool that is used to test whether some program, given some input, produces some specific output. You could use it to create tests for the log file analysis scripts you wrote previously.

I will go through `utest.sh` line by line. Some new concepts will be introduced as necessary.

```
#!/bin/sh
```

First we have the shebang. While not strictly necessary, it is good style.

```
if test $# -lt 3; then
  echo "Usage: <program> <input> <output>"
  exit 1
fi
```

The *number* of arguments given to the script is stored in the variable `$#`. We use the `test` program to check whether it is less than (`-lt`) 3. If so, we complain to the user and exit with a code indicating error.

```
program=$1
input=$2
output=$3
```

The arguments to the script are stored in variables named `$n`, where `n` is a number. We create new variables with more descriptive names for holding the arguments.

```
if ! ($program < $input | diff -u $output /dev/stdin); then
    echo 'Failed; check diff.'
fi
```

A lot of things are happening here. Let's take them one by one.

```
$program < $input
```

This runs the program stored in the variable `$program` with input from the file named by the variable `$input`.

```
($program < $input | diff -u $output /dev/stdin)
```

We pipe the output of `$program` into the `diff` program. At its most basic operation, the `diff` program takes two files as arguments, and prints how they differ. In this case, the first file is `$output` (remember, the file containing *expected* output), as well as the special pseudo-file `/dev/stdin`, which corresponds to the input read from the pipe. We additionally use the `-u` option to `diff` to get slightly prettier output.

```
if ! ($program < $input | diff -u $output /dev/stdin); then
    echo 'Failed; check diff.'
fi
```

The entire pipeline is wrapped in parentheses and prefixed with `!`. The `!` simply inverts the exit code of a command - this is because `diff` returns 0 ("true") if the files are *identical*, while we want to enter the branch if they are *different*.

Exercises

1. Turn your command lines for the previous set of exercises into shell scripts.
2. Use `utest.sh` to test your shell scripts.
3. Write a master test script that automatically runs `utest.sh` on all your scripts and their corresponding input-output files.
4. Learn about shell script `while`-loops and the `shift` command, and write a version of `utest.sh` called `mtest.sh` that can operate on any number of test sets. The script should take as argument the program to test, followed by input-output file pairs, e.g. `./mtest.sh prog in.1 out.1 in.2 out.2 ...`.

`diff` and `patch`

`diff` and `patch` are fundamental to programming in a Unix-like environment. Tools like `git` build on top of `diff` and `patch` to offer a (somewhat) cleaner interface, but sometimes, `diff` and `patch` are useful in their own right.

1. Create a folder `handout`.
2. Rewrite the shell scripts with diff and process substitution.

`find`

1. Use `grep` to find all the files.
2. Find all the files ending with ".

Environment Variables (e.g., `PATH`)

`gnuplot`

`gnuplot` is a command-line driven graphing utility. `gnuplot` is installed on our server, but you can also go ahead and install it locally.

If you run `gnuplot` in the wild it will attempt to start up a graphical user interface once there is a plot to show. Since we have no graphical user interface when connected to our server, we want to suppress this default behaviour. Initial user-level settings for `gnuplot` can be specified in your `~/.gnuplot` file.

To ask `gnuplot` to plot graphs in ASCII, add this line to your `~/.gnuplot`:

```
set terminal dumb
```

We can now try to plot something in the terminal:

```
$ echo "plot sin(x)" | gnuplot
```

or equivalently,

```
$ gnuplot -e "plot sin(x)"
```

To plot, multiple things at once, we can separate them by comma's:

```
$ gnuplot -e "plot sin(x), cos(x)"
```

To limit the x-axis range of the plot, we can specify this range after the `plot` command:

```
$ gnuplot -e "plot [-5:5] sin(x), cos(x)"
```

Alternatively, we can issue a command to set the x-axis range before calling `plot`:

```
$ gnuplot -e "set xrange [-5:5]; plot sin(x), cos(x)"
```

Similarly, we can set the y-axis range:

```
$ gnuplot -e "plot [-5:5] [-5:5] sin(x), cos(x)"
```

`public_html`

A web server has been set up on the work server, such that if you create a folder `public_html` in your home directory, and put something you are willing to share with the world in that directory, you can reach it from your browser via the URL `<host>/~<username>`.

Further Reading

1. http://people.duke.edu/~hpgavin/gnuplot.html
2. http://www.ibm.com/developerworks/aix/library/au-gnuplot/
3. http://site.ebrary.com.ep.fjernadgang.kb.dk/lib/royallibrary/reader.action?docID=10537861 (via REX)

Small Tricks and Hints

Keeping your session alive

When you are working remotely on a Unix system, all your processes will die when you log out or otherwise lose the network connection - for example when you move from the canteen to an auditorium, or in the unlikely case that Eduroam should go down. It may also be useful to have a process running for a long time, for example for running simulations or an IRC client, even when you are not logged into the machine. For these purposes, people have written *terminal multiplexers*. As the name suggests, their primary purpose is to make one terminal act like several (roughly like tabs in a browser), but they are also commonly used for their ability to *detach* from a controlling terminal, and keep running even when their owning user is not logged in to the machine. One such multiplexer is `tmux`.

`tmux` has many features, but we will introduce only a few. Starting a new tmux session is easy:

```
~$ tmux
```

You will be returned to a shell prompt, but the status bar at the bottom shows that you have started a *new* shell that runs inside `tmux`. You can *detach* `tmux` by the keyboard shortcut `CTRL-b d` (that is, hold `CTRL` while clicking `b`, then release `CTRL` and `b` and click `d`). All `tmux` commands use `CTRL-b` as their prefix, although this is configurable if you are unhappy with the default. After detaching `tmux`, you will be returned to the shell you were in when you entered `tmux`. You can re-attach a detached `tmux` instance by typing:

```
~$ tmux attach
```

If you have several detached `tmux` sessions, `tmux attach` will enter the first one. You can use flags to specify another one, but the details are out of scope for this guide. Read the documentation for details. Learning just a few `tmux` features can dramatically improve you remote Unix experience.

Uploading/Downloading Your Work

Similar to the `cp` command, there is an `scp` which can be used to copy files between machines over SSH. You can use this utility to download your work from a remote server, and/or upload your work to a remote server.

> **OBS** The following guide is only for Linux and OS X users. Windows users are referred to PSCP.

To copy a file located at `<absolute-path>` on `<host>` to `<local-path>` on your machine, while using SSH to login as `<username>`, use the following:

```
$ scp <username>@<host>:<absolute-path> <local-path>
```

To copy a local file located at `<local-path>` to `<absolute-path>` on the remote `<host>` machine, while using SSH to login as `<username>`, use the following:

```
$ scp <local-path> <username>@<host>:<absolute-path>
```

To copy a directory, pass in the option `-r`, as with `cp`. In general, the API of `scp` is very similar to `cp`. For more advanced system administration (e.g., backup), we recommend using `rsync`. See the `man`-page for `rsync` for more details.

Conclusion

If you found Unix-like operating systems interesting, we recommend that you try to play around with either Antergos or Ubuntu in your spare time. These are Linux-based operating systems well-suited for beginners coming from other operating systems. If you're afraid to mess up your computer, you can try to install them in a VirtualBox first, and try out Linux in a virtual machine.