

Console Toolkit Exercises

1 Introduction

Welcome to this exercise session! This is a practical exercise on the Linux command line. If you have any question about the exercises, feel free to ask the helpers for assistance.

Difficulty

The exercises are designed in such a way that you will have to find solutions by reading the manuals and using search engines. You are expected to come up with appropriate solutions on your own, there will be no step-by-step guided exercises!

There will be quite simple exercises for complete beginners at the start, but there are also some much more difficult ones further into the exercise. Feel free to skip ahead!

Liability

By taking part in this exercise, you acknowledge that you alone are responsible for your computer. TheAlternative and its parent organisation, the Student Sustainability Commission, will not be held liable for any damages or loss of data.

2 Basics

To solve some of the exercises, you will need to download some files. Whenever you see an orange box like below, this means you have a task to solve.

Task 2.1: Getting the repository

Open a terminal and enter the following (all on one line):

```
curl https://files.project21.ch/LinuxDays-Public/HS18-exercise.zip  
-o exercises.zip
```

Then extract the zip archive like this:

```
unzip exercises.zip
```

2.1 Running a command

To run a command, you just type it into the console and hit enter. Most useful commands require some arguments. Arguments are given after the name of the command, separated by spaces. There are also so-called *flags*, which set some options to alter the behaviour of a command. They have the form `-x` (single letter form) or `--long-option` (long format).

An example of a more involved command:

```
ls -l --human-readable ~/Downloads
```

2.2 Getting help

`man` is a command you will need throughout the following exercises. It stands for “manual” and shows you what almost any given command can do and which options are available for it. Almost all commands provide such a “manpage”, which is typically written by the developers themselves.

Task 2.2: The manual’s manual

The `man` command itself has its own manpage. Type `man man` to access it, use the arrow keys to scroll, and press `q` to exit. Have a look around, especially at the different section numbers. What is the section number for system calls?

An alternative to using the man pages is to try and see if a command has a help option.

This is usually displayed by appending the option `--help` after the name of the command, which will often display much shorter and concise instructions on how to use a command.

Task 2.3: The help option

Look at the help output of `ls`. What is the flag to see hidden files?

2.3 Navigating directories

You can navigate your file system in the terminal, just like you could with a graphical program. Below are the most important commands.

Commands for navigating directories	
Command	Description
<code>pwd</code>	Display the current working directory.
<code>tree DIR</code>	Get a visualization of the directory tree under DIR.
<code>ls DIR</code>	List all files and directories in the current working directory.
<code>cd DIR</code>	Change the current working directory to the given directory.

Task 2.4: Making yourself at home

Have a look around your home directory and try out the mentioned commands. You will most likely recognise the layout from your graphical file browser. Remember you can always return to your home directory by running `cd` without any arguments.

2.4 Modifying directories

Just like in graphical programs, you can create and delete directories on the command line.

Commands for modifying directories	
Command	Description
<code>mkdir DIR</code>	Create a new directory with the given name.
<code>rmdir DIR</code>	Remove a directory. Will not work with non-empty directories.
<code>rm -r DIR</code>	Remove a directory and its contents. Attention! There is no trashcan on the command line! Files and directories will be deleted irrevocably.
<code>cp -r SOURCE TARGET</code>	Copy a source directory (recursively with all its contents) to a target.
<code>mv SOURCE TARGET</code>	Move a file or directory. Also used to rename files/directories.

Task 2.5: Creating your first directory

Create a directory for the following exercises (you can call it `console-toolkit`). Change into that directory and create a file called `notes`. Then move the exercise file directory you downloaded in task 2.1 into your newly created directory.

2.5 Viewing files

If you want to view the contents of a text file, you have the following options:

Commands for viewing files	
Command	Description
<code>cat FILE</code>	Output a file to the terminal.
<code>head FILE</code>	Output the first couple of lines to the terminal.
<code>tail FILE</code>	Output the last couple of lines to the terminal.
<code>less FILE</code>	Browse a file in a visual viewer.

Task 2.6: Log files

`tail` is often used to inspect errors log files. Use `tail` to find the last three lines of `dmesg.log` (in the exercise files).



`dmesg` is tool to view the log output of your kernel.

2.6 Console tips & tricks

There are many small things that can make your life when using the terminal little bit easier. Following is a table with the most important keyboard shortcuts you can use to speed up your workflow.

Terminal keyboard shortcuts	
<code>ctrl + w</code>	Delete one word backwards.
<code>ctrl + u</code>	Delete the entire line.
<code>ctrl + l</code>	Clear the terminal.
<code>ctrl + a</code>	Go to the beginning of the line.
<code>ctrl + e</code>	Go to the end of the line.
<code>ctrl + c</code>	Terminate the currently running process.
<code>ctrl + d</code>	Quit the shell.

3 Files & Permissions

This section introduces you to basic commands related to files. We will first look at a command that can be used for creating empty files.

The `touch` command is most frequently used for creating new, empty files. If you check the manual, you will see that it can also change certain file timestamps (access and modification times).

Task 3.1: The magic touch

Switch to your home directory. Execute `touch file`. A new file should have been created. Confirm that this is the case.

We already saw `cat` in the previous section. Another tool to look at files is `less`. This command is great if you want to look at large bodies of text because unlike `cat`, it has a built-in scroll function. Once you are done looking at the file, you can quit with `q`.

3.1 Permissions

We will now look at two commands to manage permissions.

`chown` stands for “change owner”. It is used to change the owner of a given file. It can also change the group ownership.

Task 3.2: Get off my lawn

Create a new, empty file (hint: `touch`) and use `chown` to change its ownership to the `root` user (you will probably need `sudo` for this). Try deleting it. Then change the ownership back to your own user.

A very similar command is `chmod`. `chmod` stands for “change file mode bits” and controls the following permissions on any given file:

- Read: Who can see the file data.
- Write: Who can modify the file.
- Execute: Who is allowed to run the file (like a program or a script)



Permissions can be viewed using `ls -l`.

Task 3.3: Read, Write, Execute

The read, write and execute permissions do different things, depending on whether they are set for directories or for files. Find out what they mean for files and directories by playing around with the `chmod` command, reading its manual or searching on the internet.

3.2 File tools

There are a number of other commands that are frequently used to get information on files:

Commands for getting information about files	
Command	Description
<code>wc \$file</code>	Display line, word, and character count of a file
<code>diff \$file1 \$file2</code>	Show differences between files
<code>ln \$target \$link_name</code>	Create a link
<code>du \$file</code>	Estimate file space usage
<code>df</code>	Report file system disk usage

In the next exercise we are going to read some text from the standard input (also called `stdin`). Reading from `stdin` means you can just type text into the console. When you are done, hit `Enter` to enter a newline and then hit `ctrl`+`d`. This is the universal end-of-file character in Unix-like systems. It signals to the command that the input has finished. You can even use this key shortcut to close your terminal.

Task 3.4: Cat tricks

Type the command `cat -`. This starts `cat` to read from `stdin`. Enter some text and observe what happens when you press `Enter`. `cat` will just keep reading until the end-of-file character, so you can enter multiple lines of text. When you are done, terminate the input as described above.

Task 3.5: Counting words

Sometimes you might want to know the length of some text. First, consult `wc`'s manual to find out how you can read from standard input. Copy a line of text of your choice from the `wc` manual, then use the `wc` syntax that you looked up to read from standard input. Paste the text and terminate the input as before.



Copying and pasting in the console usually works with `ctrl`+`⬆`+`c` and `ctrl`+`⬆`+`v`, this depends on the terminal though.

Hardlink A hardlink is a pointer to a specific, physical location on a hard disk. If the file name of the file that is linked to changes, the link still works. However, if the file is replaced by a different file with the same name, the hardlink will now *not* point to the new file! It will still point to the location of the old file.

Symlink A symlink (short for symbolic link) points to a file name, to an actual file path. If the data behind the file name is changed, the symbolic link will then point to this new file, as long as the name stays the same. This is a higher level concept than a hardlink, as it operates above the concept of file names. Most of the time, a symlink is preferable to a hardlink.

Task 3.6: Symlinks

Have a look at the `ln` manual to find out how to create a symbolic link. Then create a link in your home directory to a directory you use frequently.

Lastly, two commands that are loosely related to files are `df` and `du`. These commands are used to find out how much space files take up. `df` shows coarse-grained disk usage as reported by the file system. `du` counts the disk usage of individual files and directories.

Task 3.7: Reporting disk usage

When you run `df` and `du`, the sizes of files are reported in multiples of blocks. The size of a block depends on your file system, but typically is between 512 bytes and 1 kilobyte. This means that you would have to multiply by this size the true size, in bytes. Find out how to make the output of `df` and `du` more readable for humans by looking at their options in the manual (hint: the option is the same for both tools).

4 Leveraging the command line

4.1 Globbing (Wildcards)

Globbing is a useful tool to work with files that share a pattern. For example, to refer to all `jpg` images in a directory, you can use `*.jpg`. The `*` means ‘any number of any characters’, so this pattern will give you all files that end with `.jpg`. Similarly, `dir/*` will give you *all* files in `dir`, *except* hidden files.

There are a few more useful patterns, like `?` (question mark) which matches any *single* character, or `{*.jpg,*.png}`, which is a list that matches both `.jpg` and `.png` files. To learn more, you can have a look at `man 7 glob`.

Task 4.1: Printing text, again

Output all `.txt` files in `notes/` to the terminal in a single command. The output will tell you if you did it correctly.

Task 4.2: Managing cat pictures

Create a new directory called `cats`. Copy all cat pictures in `pictures/` (in the exercise files) to your new `cats` directory, with a single command.

4.2 Pipes

Pipes are useful to connect multiple commands together. One of the design principles of Unix was “do one thing, and do it well”. This means, tools should be simple and powerful in their specific domain, but there should also be a mechanism to combine multiple programs in an easy way. This mechanism is the *pipe*, the `|` symbol. It allows you to connect the output of one program to the input of another program.

For example, we can combine `du` and `sort` to work together. `du` analyses the usage of disk space, and `sort` just sorts lines it is given. This pipe analyzes your home directory and tells you which directory uses the most space:

```
du -sh * | sort -h
```

Here the `*` globbing pattern stands for “every file in the current directory”.

Task 4.3: Counting lines in multiple files

`wc` (word count) can be used to count the number of lines in a file (`-l` flag). Use `cat` to output all files in the `notes` directory (exercise files) at the same time and connect it to `wc` to count the lines.

4.3 Find

Looking for specific files on your system is a task that one has to do fairly regularly. Two very popular tool for finding files with certain properties are `find` and `grep`. In this section, we cover `find`. As a rule of thumb, `find` is best used when you know that your file has certain metadata: Maybe you know approximately when you have created the file, when you last looked at it, how large it is etc. In contrast, `grep` is usually used when you want to look at *the content* of a file.

`find` works by essentially filtering all files through a chain of conditions, given by flags. Working with `find` is fairly straight forward and almost like verbally explaining what you are looking for.

```
find -mtime -1 -size +100k
```

This searches for all files that have been modified (`-mtime`) less than one day ago (`-1`) and have size (`-size`) larger than 100 Kilobytes (`+100k`). `find` has a lot of conditions where it makes sense to specify “greater than” or “smaller than”. This is specified by prefixing the amount with a `+` or a `-`. This is why the condition “larger than 100 Kilobytes” is written as `-size +100k`.

If we additionally knew that our file was smaller than one gigabyte, we would type:

```
find -mtime -1 -size +100k -size -2G
```

It's necessary to use `-2G` here, since file sizes are rounded up. A file that has 800M will

be rounded to 1G. The semantics of `find` use strictly smaller (`<`), therefore we have to compare to 2G.

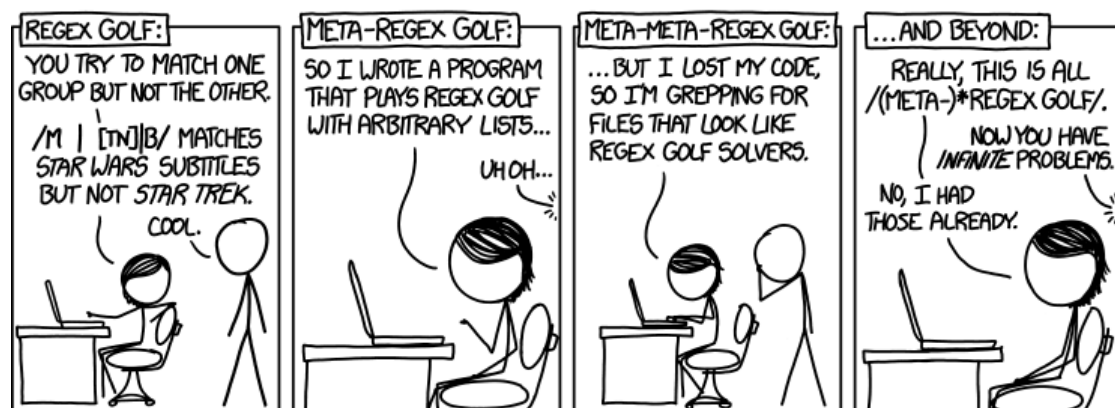
Here is an overview of options for `find` that are useful to know.

Useful find flags	
Flag	Description
<code>-name</code>	Matches the name of a file.
<code>-size</code>	Condition on the size of a file.
<code>-atime</code>	Condition on when file was last accessed.
<code>-mtime</code>	Condition on when file was last modified.
<code>-type</code>	Specifies type of file (directory, regular file ...).

Task 4.4: The largest cat

Use `find` to find all cat pictures (`cat*`) with a size over 200 kilobytes in the exercise directory. There should be more than one!

4.4 Grep and regular expressions



xkcd 1313, <https://xkcd.com/1313/>

While working with many files in different folders, you can find yourself losing track of what is in what file. To quickly find something in one of many files, you can use `grep`. `Grep` is a tool for searching a body of text for a specific pattern. Those patterns are specified with a syntax called “Regular Expressions” or `regexes`.

Regexes can easily become *very* complicated (there are entire books on this topic). We’ll have a look at the very basics here. Regexes are comprised of a string of characters, e.g. `foo.bar[0-9]`. Let’s look at it in more detail:

- `foo` Matches the literal word “foo”.
- `.` Matches any single character.
- `bar` Matches the literal word “bar”.
- `[0-9]` Matches a single digit between 0 and 9.

For example, this regex `foo.bar[0-9]` would match on all of these strings:

- `foo_bar1`
- `fooobar9`
- `foozbar5`
- `foo1bar2`

An important thing to note is that regular expressions don’t have to match the entire string. So the example regex would also match on `XXXXfoo1bar2XXXX`, because `foo1bar2` is part of that string.

So how do we try that on the command line? Simple, to test out `grep`, we can use `echo` to pipe a string into `grep` to match. Like this:

```
echo foo1bar2 | grep foo.bar[0-9]
```

This should output `foo1bar2`, because `grep` outputs all lines from its input that match the regex.

There are a few command line options that you should know about `grep`. These can also be combined, like so: `grep -rnoh PATTERN FILE`.

Useful grep flags	
Command	Description
<code>-r</code>	Recursively look at all files under the given directory.
<code>-o</code>	Only print the matching strings, not the entire line.
<code>-h</code>	Don’t print the filename, just the matching string/line.
<code>-n</code>	Prefix the output with the linenummer where the match occurred.

Regex building blocks		
Regex	Example	Description
Match Expressions		
<character literal>	f	Matches the character <character literal>. The example matches on f.
.	.	Matches any single character.
[<char1>-<char2>]	[a-z]	Matches a single character in the range from <char1> to <char2> (inclusive). The example matches on any lowercase character. The character range can be specified multiple times, like so: [a-zA-Z0-9]. This would match any lower- or uppercase character or digit.
[[[:alnum:]]]	[[[:alnum:]]]	Alphanumeric characters (letters and digits).
[[[:alpha:]]]	[[[:alpha:]]]	Letters.
[[[:digit:]]]	[[[:digit:]]]	Digits.
[[[:xdigit:]]]	[[[:xdigit:]]]	Hexadecimal digits.
[[[:blank:]]]	[[[:blank:]]]	Blank space (spaces and tabs).
[[[:word:]]]	[[[:word:]]]	Digits, letters and underscores.
Modifiers		
?	.?	The preceding expression appears zero times or exactly once. The example would match on an empty string or any string of length one.
*	.*	The preceding expression appears zero, one or many times. The example would match on a string of arbitrary length, including an empty one.
+	.+	The preceding expression appears one or many times. The example matches on a string of any length greater than zero.
{<num>}	{3}	Matches on a string if the preceding expression appears exactly <num> times.
{<num1>,<num2>}	{3,6}	Matches on a string if the preceding expression appears between <num1> and <num2> times (inclusive).

Note When using the characters |, +, ? with their regex meaning, they must be escaped with a backslash like this: \|, \+, \?. If you want to use any of the other special characters as regular characters, i.e. to match a literal . you can use \.

Task 4.5: Finding email addresses

Somewhere in the exercise files, there is an email address hidden in a text file. Create a regular expression that matches email addresses that end in .ch and find it!

5 Remote machines

This section will be about working with remote machines. You will connect to one and move files from and to it. Additionally, you will set up your SSH so you can log in without entering a password.

ssh (secure shell) is an extremely useful tool, as it allows you to manage servers, connecting to your home computer from a laptop, running computations on super computers, and even using graphical applications that are installed on a different computer. **scp** (secure copy) works just like **cp**, but allows you to copy files to and from remote machines. **scp** is based on the SSH protocol, so it works similarly and the public key authentication in exercise 5.2 will automatically also apply to **scp**.

Remote access tools	
<code>ssh user@hostname</code>	Open an SSH session with a specific user on a host.
<code>scp user@hostname:file.txt .</code>	Copy a file from remote machine to the current directory.
<code>scp file.txt user@hostname:file.txt</code>	Copy a file from current directory to remote host.



If you don't have an ETH account, use `hackingsession@pterodactyl.vsos.ethz.ch` with password `BoredHacker` for the following exercises instead.

Task 5.1: Establishing an SSH connection

Use SSH to connect to the Euler supercomputer at `nethz-login@euler.ethz.ch`, where `nethz-login` is your *nethz* account name. You can login using your regular credentials.

It is inconvenient that you have to enter your password every time you want to log in. You can set up SSH in a way where you *don't* have to provide your password! This is done with public key cryptography. Basically, you generate a public key and a private key. The private key you keep safe and only for yourself. The public key you can

distribute to anyone. Now you can authenticate yourself for anyone that has your public key: You can create a signature with your private key, that can be verified using the public key. However, only the owner of the private key can create the signature that is verifiable by the public key! This can be used by an SSH server to authenticate clients that want to open an SSH session.

Task 5.2: Generating your key

Generate your private/public key pair using `ssh-keygen`. Then copy your generated public key (`.ssh/id_rsa.pub`) to `.ssh/authorized_keys` on the server. Try logging in with SSH now, it should no longer require a password.

6 Jobs & Processes

Jobs are groups of processes that are running in the current shell. Often, a job only consists of one process but sometimes a job is a process *pipeline* consisting of multiple jobs, such as `echo "Hello" | cat -`. Consult the `JOB CONTROL` section of the `bash` manual if you want to study the details.

Knowing how to control jobs is essential if you want to be able to simultaneously run multiple processes from the same shell.

6.1 Job control

Jobs can be put into the background, started in the background, brought to the foreground and killed. We say a job is running in the background when it is currently running, but not blocking the shell. In other words, when a job is running in the background, you can still use your shell normally (although the job might produce output that will get displayed on your terminal which can be quite annoying).

Working with jobs	
<code>jobs</code>	List all jobs in the current shell
<code>cmd &</code>	Start <code>cmd</code> in the background
<code>fg</code>	Bring a job into the foreground
<code>bg</code>	Bring a job into the background

Controlling jobs	
<code>ctrl+z</code>	Suspend current job. Processes may ignore this.
<code>ctrl+c</code>	Kill current job. Processes may ignore this.
<code>kill %N</code>	Kill job, where N is its number as listed in <code>jobs</code> . Processes may <i>still</i> ignore this.
<code>kill -9 %N</code>	Kill job with number N for sure.

You can *usually* use `ctrl+c` to terminate a job in the foreground and `ctrl+z` to suspend a job in the foreground. Suspended jobs are paused until they are resumed with either `fg` or `bg`. Note that processes can choose to overwrite these shortcuts so they are not guaranteed to work. If they don't work, we can always use more drastic measures ...

6.2 Processes

Processes operate system-wide. Each process has a unique process ID, called the PID for short. You can think of processes as separate programs running independent of each other. For example, an `echo` command spawns a new process that is separated from the shell process that you are in (which is typically `bash`).

Managing	
<code>top</code>	Display info about all currently running processes. Quit with <code>q</code>
<code>htop</code>	A more modern equivalent of <code>top</code> . It is not installed by default on most distributions, but can be installed with your package manager.
<code>kill PID</code>	Send a termination signal to the process PID. This is the preferred way to terminate a process, as it allows it to exit normally and perform clean up.
<code>kill -9 PID</code>	Send a killing signal to the process PID. The process is killed by the operating system immediately. Useful for unresponsive processes.
<code>pkill PATTERN</code>	Like <code>kill</code> , but kills processes whose names match a pattern instead.
<code>pkill -9 PATTERN</code>	Send a kill signal instead.
<code>ps</code>	Utility to display information about processes. Most commonly used as <code>ps aux</code> .

`kill` and `pkill` actually send so-called signals to processes. There are many signals that all do different things. When you call `kill PID`, the signal 15 is sent to the process PID. This signal is called the `SIGTERM` signal and is used to *ask* a process to terminate.

When we write `kill -9 PID`, we are actually sending the signal 9 which is also known as `SIGKILL` and kills a process instantly. Read `man 7 signal` if you want to know more about signals.

Task 6.1: Killed in their sleep

Start a new process by typing `sleep 1000` in your shell. Open another shell. Use `top`, `htop` or `ps` to figure out the PID of the `sleep` process. Kill it.

Task 6.2: A job for kill

`kill` also works on jobs. Start `sleep 1000` in the background (or start it in the foreground, suspend it, then bring it to the background). Then type `kill %N` where `N` is the job's number (probably 1). Check the output of the `jobs` command and confirm that the job is not displayed anymore.

7 License

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. See also: <http://creativecommons.org/licenses/by-sa/4.0/>.

