# Export statement

When we are defining environmental variables we can use the export statement:

* Export X=1

This way the specified variable will be accessible in all the subprocesses (compiled programs) which we run after defining that variable.

Here are examples of subprocesses using environmental variable:

* Example 1. A bash subprocess:
  + export X=1
  + bash -c 'echo $X'
* Example 2. A Python subprocess:
  + export X=1
  + python3 -c 'import os; print(os.environ["X"])'

# Init system

That is a process which runs as the first one after kernel boots.

# Shell

Shell is a program which allows using commands in terminal and returns outputs. Those commands are from the shell’s language. Popular shells:

* sh (Bourne Shell)
* bash (Bourne Again Shell)

On Linux both those shells are saved in the /bin/sh and /bin/bash respectively.

# Processes

## Daemons

Daemons are background, long running processes.

## Commands

* ps aux – show all processes

# Useful bash commands and CLI Linux tools

## Commands outputs

There are two types of bash commands outputs:

* stdout (standard output)
* stderr (standard error)

The stdout is the normal output of a command – the result which it prints when it runs successfully.

The stderr is the error or warning message from a command.

## Exit codes

Every bash command returns an exit code. Exit code = 0 means success and any other value means some kind of error.

We can access an exit code of a command by using $? after that command, for example:



## Redirect an output of a command

We can redirect an output of a command:

* To a file:
  + Command > file – overwrite a file if it exist
  + Command >> file – append to a file if it exists
* To another command:
  + Command1 | command2 – output of the command1 becomes an input for the command2
* Redirect the standard output (stdout) to one location and standard error (stderr) to another location:
  + Command > filename1.txt > filename2.txt
* Redirect stdout and stderr to the same location:
  + Command > filename.txt 2>&1
* Hide the stdout and stderr (redirect it to the ‘black hole’):
  + Command > /dev/null 2>&1

## Get a command output

* X=$(command) – Assign output of a commnd to an environental variable X

## Grep

It is a tool for filtering text. For example we can:

* Filter a content of a file. Below command shows only those lines from the file.txt file which contains ‘error’ string:
  + Cat file.txt | grep “error”
* Filter output of some command. Below command shows only running processes related to nginx:
  + ps aux | grep nginx

## Editing files context

For editing files the best tools are:

* sed - More information in the ‘Sed’ section of this document.
* Cat used together with heredoc and command redirection - More information in the ‘Heredoc’ section of this document.

## Sed

Sed is used to replace all the strings matching the pattern with a new string in a specific file:

* sed -i ‘s/pattern/replacement/’ file\_path

Patterns which are being matched to strings are using Regular Expressions, more information is here: [link](https://www.gnu.org/software/sed/manual/html_node/Regular-Expressions.html).

## Heredoc

Heredoc is used in order to pass a multiline argument to a command:

* Command << EOF

Line1

Line2

EOF

For example it can pass a multiline text as an argument to the Cat command what will display all those lines.

If we additionally use the ‘>’ or ‘>>’ to redirect an output we can save multiline text to a file:

* Cat << EOF >> file\_path

Line1

Line2

EOF

## Tee

The tee command redirects an output of another command to both a terminal and a file. Example use cases:

* Command | tee filename - Save an output of the command in the given file and also write it in a terminal.
* Command | tee filename | another\_command – save an output of the command in the given file and also redirect that output to another command.

## Echo

The echo command prints a text. It can be used for example in order to save a text into a file.

## Ls

Ls command is used for listing files contained in a directory and checking their permissions. Syntax:

* Ls [options] folder\_path

Where folder \_path is a path to the folder from which we want to list the content (files and folders).

Popular options:

* -l – show permissions, owner, size etc.
* -a - show all file, including the hidden ones (which name starts with a dot ‘.’).
* -d – Get information about the directory itself (given by the folder\_path), not its content.

## Mkdir

Create a folder. Syntax:

* Mkdir [options] folder path

Popular options:

* -p – Create the entire path. For example if we run ‘mkdir -p folder1/folde2 and both folders doesn’t exist yet, then it will create both of them.

## Modifying files and folders permissions and ownerships

### Chmod

Chmod is changing permissions for a specific file or folder. It uses this syntax:

* Chmod xxx file\_path

Where xxx are numbers specifying the permissions.

### Chown

Chown is changing an owner of a file or folder. It uses this syntax:

* Chown [options] [username][:usergroup] file\_path

Where:

* Username – username of a new owner
* :usergroup (optional) a new group owner

Popular options:

* -R – change ownership of all the files and folders which are contained in the folder given by the file\_path parameter.

## Installing packages

### Apt-get

Apt-get is a popular package manager used for installing different tools.

Popular commands:

* Apt-get update - Updates the list of available packages
* apt-get install <package> - Installs a package
* apt-get remove <package> - Remove a package

### GPG key

The GPG keys are used to ensure that the packages which we install are secure. When we want to install packages from a new repository we might need to import ints GPG key to our system so apt-get trusts packages from it.

## Loops

### For loop

Iterate over a list:

A screenshot of a black screen

AI-generated content may be incorrect.

Iterate over a sequence of numbers:

A screenshot of a black screen

AI-generated content may be incorrect.

Iterate over a sequence of numbers with specified step:

A screenshot of a black screen

AI-generated content may be incorrect.

### While loop

Syntax for the while loop:

A black screen with white text

AI-generated content may be incorrect.

## If statements

Syntax:

A screen shot of a computer

AI-generated content may be incorrect.

The condition must return either:

* true or false
* zero or non zero value

For example:

* [X -eq 10] condition compares environmental variable X to 10 and returns true or false.
* [command] condition runs a command which returns an exit code. If exit code = 0 that means true, any non zero value means false.

## Sysctl

It is used to view and modify kernel parameters without rebooting the system.

Those parameters control how Linux kernel behaves, including things like:

* Networking settings
* Memory limits
* Process handling
* Security controls

Those parameters are stored in the /proc/sys/ directory.

If we change some parameters using the sysctl command, then those changes are not persistent across reboots.

If we want them to be consistent we need to create a .conf file in the /etc/sysctl.d/ folder and save those changes there.

For example this command:

* sysctl -w net.ipv4.ip\_forward=1

changes the net.ipv4.ip\_forward parameter which is stored in the /proc/sys/net/ipv4/ip\_forward file.

If we want to have this parameter set up like this across reboots then we need to create a file like this:

* echo “net.ipv4.ip\_forward=1” > /etc/sysctl.d/filename.conf

# Isolating processes

We can isolate processes on Linux, such that they can:

* Use only a specific amount of resources (CPU, RAM)
* See and access only a specific files, other processes

For managing which resources processes can use, we are using cgroups.

For managing what processes can see we are using namespaces.

## Cgroups

We can use cgroups (control groups) in Linux in order to control and limit the resource usage (CPU, RAM etc) of groups of processes.

Each cgroup get assigned a specific amount of resources.

We can use for example systemd or cgroupfs for managing cgroups.

## Namespaces

Namespaces isolate what a process can see and access – such as files, processes, users, network interfaces and more.

Namespace is a group of processes for which we apply restrictions.