Automation

Bash scripts

Scripting is a great way to automate simple tasks or chains of commands. Imagine you want to take a backup. Usually this means creating a `zip` archive of certain files and folders, giving this `zip` file the appropriate name and then move the `zip` file to a certain folder. Running these commands manually is prone to errors and time consuming. A better way would be to bundle these commands in a script, so we can instead run one command that will trigger running all of the steps described above.

In Ubuntu we often use *bash* scripts. *bash* refers to the default shell that we use as a user in Ubuntu. There are different kinds of *shells* but this is out of scope for this course. Most syntax seen in this chapter will work in most shell environments.

> Note that we will focus on the basic syntax of a shell script. You can always learn more about if statements, loops, custom options and arguments, tests, ... in scripts but this is out of scope for this course.

Hello world

To start of with our first bash script we'll create a new file called `helloworld.sh` and add some contents using `nano` .

```bash
student@linux-ess:~$ nano helloworld.sh
```

We give the file the following contents:

bash

```
#!/bin/bash
echo "hello world"
echo "this is our first bash script"
```

The first line in this script is named in the shebang and is a special line that will make sure the script will be ran in a `bash` shell. After that we can have different lines of all kinds of commands that will be run in sequential order.

> using `#` signs are interpreted as comments. Any code after those signs will not be executed.

In the example above we will run 2 `echo` commands.

To run this bash script, we will have to add *execute* rights:

bash

```
student@linux-ess:~$ ls -l helloworld.sh
-rw-rw-r-- 1 student student 371 Nov 11 15:55 helloworld.sh
student@linux-ess:~$ chmod u+x helloworld.sh
student@linux-ess:~$ ls -l helloworld.sh
-rwxrw-r-- 1 student student 371 Nov 11 15:55 helloworld.sh
```

After doing this we can execute the script:

bash

```
student@linux-ess:~$ ./helloworld.sh
hello world
this is our first bash script
```

> Although the script is in the working directory, we have to specify it with: ./helloworld.sh Another way to run it, is to specify the full path: /home/student/helloworld.sh

> Only scripts that are executable and saved in a directory which is specified in the $PATH variable can be executed without specifying the full path.

date with shell embedding

Lets extend our script with some nifty logic to use to output of a certain command in another command. We edit the script contents as follows:
nano showdate.sh

bash

```bash
#!/bin/bash
echo "hello world"
echo "this is our first bash script"
echo "the date of today is $(date)"
```

When we run this script, we get the following output:

bash

```bash
student@linux-ess:~$ chmod u+x showdate.sh
student@linux-ess:~$ ./showdate.sh
hello world
this is our first bash script
the date of today is Tue Jun 28 22:04:09 CEST 2022
```

The concept we've used here is called *shell embedding*. The `$(...)` syntax opens up a new (sub)shell and runs a command. The output of the `date` command is then directly used in the echo command.

Variables

We also can make use of variables to reuse data:
nano vars.sh

bash

```
#!/bin/bash
CUSTOMDIR=testdir
echo "customdir is set to $CUSTOMDIR"
mkdir ~/$CUSTOMDIR
touch ~/$CUSTOMDIR/testfile
ls -l ~/$CUSTOMDIR/*
```

We can run the script without setting the execute script (chmod u+x) first, but then we have to specify the interpreter (here bash) to define in which language the script is written

bash

```
student@linux-ess:~$ bash vars.sh
customdir is set to testdir
-rw-r----- 1 student student 0 Nov 11 15:52 /home/student/testdir/testfil
```

System variables

We also can make use of variables that are set on the system:
nano sysvars.sh

bash

```
#!/bin/bash
echo "hello $USER"
echo "your homefolder is $HOME"
```

chmod u+x sysvars.sh

bash

```
student@linux-ess:~$ ./sysvars.sh
hello student
your homefolder is /home/student
```

Lets combine some of the stuff we learned so far.

Lets create a new script called `countfiles.sh` :

nano countfiles.sh

bash

```
#!/bin/bash
echo "hello $USER"
echo "your homefolder has $(ls -A1 ~ | wc -l) files/folders."
```

When we execute it, we get the following result:

bash

```
student@linux-ess:~$ bash countfiles.sh
hello student
your homefolder has 12 files/folders.
```

Another example creates a file with the current date in the filename:

nano datefile.sh

bash

```
#!/bin/bash
# This will create a variable with the current date as value
createdate=$(date +"%Y-%m-%d")
touch ~/${createdate}-superfile && echo "File ${createdate}-superfile cre
```

When we execute it, we get the following result:

bash

```
student@linux-ess:~$ bash datefile.sh
File 2022-11-11-superfile created/touched in homedir.
```

Mind that if we ask for the value of a variable by putting a dollar sign in

front (eg. `echo $createdate`). But when we put text right after the variable without a space in between, we almost always have to put the variable within boundaries with the curly brackets (eg. `echo ${createdate}superfile`)

Reading user input

Sometimes it can be helpful to ask for user input. We place the answer of the user in a variable, so that we can reuse it later on in the script:

nano list5.sh

bash

```
#!/bin/bash
echo "Enter the absolute path of a folder you want to check:"
read folder
echo "The selected folder is $folder."
```

Let's extend this script to give it some more functionality and combine some of the concepts we've learned before:

bash

```
#!/bin/bash
echo "Enter the absolute path of a folder you want to check:"
read folder
echo "The selected folder is $folder. This folder contains $(ls -A1 $fold
echo "Showing the first 5:"
ls -A1 $folder | head -5
```

Let's see what it does:

bash

```
student@linux-ess:~$ ./list5.sh
Enter the absolute path of a folder you want to check:
/home/student
the selected folder is /home/student. This folder contains 23 files/folde
```

```
Showing the first 5:
2022-11-11-superfile
auth.log
.bash_history
.bash_logout
.bashrc
```

Using a parameter

We are going to keep it simple, so we will not use more than nine parameters. In our example we will only use two ($1 and $2), but know that method stays the same for all nine ($1-$9). As a sidenote I can tell you that $0 also exists and holds the command with which the script was run.

bash

```
student@linux-ess:~$ nano params.sh
student@linux-ess:~$ cat params.sh
echo Param one is: $1
echo Param two is: $2
student@linux-ess:~$ chmod u+x params.sh
student@linux-ess:~$ ./params.sh first 2nd
Param one was: first
Param two was: 2nd
```

The PATH variable

Linux has multiple places where binaries are stored. These are often bundled in the PATH variable.
If we run a command without specifying the path where the command is saved, there will be searched for within every path of the PATH variable.

bash

```
student@linux-ess:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/
```

Mind that every path is seperated with a semicolon

bash

```
student@linux-ess:~$ echo $PATH | tr ':' '\n'
/usr/local/sbin
/usr/local/bin
/usr/sbin
/usr/bin
/sbin
/bin
/usr/games
/usr/local/games
/snap/bin
```

The PATH variable gets set and altered in multiple scripts. It starts with a system wide setting in /etc/environment :

bash

```
student@linux-ess:~$ cat /etc/environment
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/g
student@linux-ess:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/
```

So we can alter this file if we want to change the PATH variable for every user on the system. The change will be visible for a user when he logs in.

But if we want to alter the PATH variable for one user, we can do this from within the file ~/.profile. The change will be visible for a user when he logs in:

bash

```
student@linux-ess:~$ grep -C1 "HOME/bin" .profile
# set PATH so it includes user's private bin if it exists
if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
fi
```

> Mind that this already exists, so it is best to save your scripts in a (new) folder named *bin* in your homefolder. Also mind that after creating the *bin* directory you have to login again so the *bin* folder gets added to the PATH variable.

As we have seen already you can use the command `which` to find out if the command gets found and where precisely.

bash

```
student@linux-ess:~$ which reboot
/usr/sbin/reboot
```

At

You can use the at command to run a script/command at a specific time.
For the at command we echo an output into the at command as shown by an example below:

bash

```
student@linux-ess:~$ echo ./datefile.sh | at 14:00
warning: commands will be executed using /bin/sh
job 1 at Sat Nov 12 14:00:00 2022
student@linux-ess:~$ echo 'echo "hello world" > /tmp/hello.txt' | at now
warning: commands will be executed using /bin/sh
job 2 at Mon Nov 14 10:36:00 2022
student@linux-ess:~$ at -l
2       Mon Nov 14 10:36:00 2022 a student
1       Sat Nov 12 14:00:00 2022 a student
```

you can check what is scheduled with the `atq` or `at -l` commands and remove when necesarry with the `atrm`, `at -d` or `at -r` commands:

bash

```
student@linux-ess:~$ at -l
```

```
2       Mon Nov 14 10:36:00 2022 a student
1       Sat Nov 12 14:00:00 2022 a student
student@linux-ess:~$ at -d 2
student@linux-ess:~$ at -l
1       Sat Nov 12 14:00:00 2022 a student
```

For more info check the manpage of `at`.

Crontab

With the cron command it is possible to plan to run your scripts/commands to run on a regular basis. Cron works different, it uses a file where we put the commands/scripts we want to run and specify when the runs have to happen. To open your crontab-file you give the command `crontab -e`.

In the example below we are going to echo some text to a file every minute.

bash

```
.---------------- minute (0 - 59)
|  .------------- hour (0 - 23)
|  |  .---------- day of month (1 - 31)
|  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
|  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,
|  |  |  |  |
*  *  *  *  *   echo Command run at $(date) >> /tmp/crontest
```

Simply remove the line from the crontab to stop this from happening. For more info check the manpage of `cron`.

> Mind that the first time we open cron we have to specify which editor we want to use. If you choose '1' you will have the editor `nano`.

bash

```
@linux-ess:~$ crontab -e
```

```
no crontab for student - using an empty one

Select an editor.  To change later, run 'select-editor'.
  1. /bin/nano        <---- easiest
  2. /usr/bin/vim.tiny
  3. /bin/ed

Choose 1-3 [1]: 1
crontab: installing new crontab
...
# m h  dom mon dow   command
* * * * * echo Command run at $(date) >> /tmp/crontest

@linux-ess:~$ crontab -l
...
# m h  dom mon dow   command
* * * * * echo Command run at $(date) >> /tmp/crontest

student@linux-ess:~$ cat /tmp/crontest
Command run at Sat Nov 12 11:02:01 AM UTC 2022
Command run at Sat Nov 12 11:03:01 AM UTC 2022
Command run at Sat Nov 12 11:04:01 AM UTC 2022
```

If we have to run a script as another user or with elevated privileges (as root), we can make use of the general *crontab* file in */etc*. In this file there is an additional column to specify the user under which the script has to run.
You will need elevated privileges to edit this file (sudo).
In the following example we will run a backup script every friday at 23:30 :

bash

```
student@linux-ess:~$ sudo nano /etc/crontab
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
```

```
# You can also override PATH, but by default, newer versions inherit it f
#PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,we
# |  |  |  |  |
# *  *  *  *  * user-name command to be executed
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts
30 23   5 * *   root    /scripts/backuphomefolders.sh
```

Notice that there are some folders (cron.daily, cron.weekly, cron.monthly) that can hold scripts that will be executed by cron regularly.

bash

```
student@linux-ess:~$ ls /etc/cron.daily/
apport  apt-compat  dpkg  logrotate  man-db
```

In Linux the system is managed with systemd. And systemd also has an implementation for repeating jobs, namely *timers*. But that's out of scope for this course.

Previous
9 Users & permissions

Next
Lab