EDUCATION OF THE REPUBLIC OF BELARUS

EDUCATIONAL INSTITUTION
«BREST STATE TECHNICAL UNIVERSITY»

DEPARTMENT OF INTELLIGENT INFORMATION TECHNOLOGY

GUIDELINES

FOR LABORATORY WORKS CARRYING OUTON DISCIPLINE

«System Software»

for full-time and extra-mural students of the undergraduate programme
1–53 01 02 Automated Data Processing Systems

PART I

«Linux Command Line Interface»

BREST 2015

UDC 004.45

The goal of these guidelines is to help students in the study of theoretical and practical principles of work in Linux operating system using Command Line Interface.

In the theoretical sections of the manual basic concepts and commands are presented. This theory covers laboratory works №№1–4 on the discipline «System Software», including file system manipulations, links and permissions, data streams redirection and piping, regular expressions. The tasks for the laboratory works are given in Appendixes A–D. Each task contains reference to the theoretical sections describing the commands to be used.

The guidelines are designed for the students of the undergraduate programme 1–53 01 02 Automated Data Processing Systems of full-time and extra-mural forms of education.

The guidelines are published in 2 parts. Part 1. Linux Command Line Interface.

Fig. 3, ref. 4.

Compilers: P.A. Kachurka, PhD., associate professor
V.A. Kachurka, teacher assistant

Educational institution
© «Brest State Technical University», 2015

CONTENTS

1 COMMANDS **5**
1.1 Command Line Interface. Terminal. 5
1.2 Manual Pages 6
2 FILE SYSTEM **7**
2.1 File System Navigation 7
2.2 Paths 8
2.3 Let's Move Around a Bit 10
2.4 Everything is a File 11
2.5 Filesystem Hierarchy Standard 13
3 DIRECTORIES AND FILES **16**
3.1 Making a Directory 16
3.2 Removing a Directory 17
3.3 Creating a Blank File 17
3.4 Copying a File or Directory 18
3.5 Moving a File or Directory 19
3.6 Renaming Files and Directories 19
3.7 Removing a File 20
3.8 Removing non empty Directories 20
4 LINKS **20**
4.1 Hard link 20
4.3 The Difference Between Soft and Hard Links 22
5 FILE PERMISSIONS **22**
5.1 Types of Permissions 22
5.2 Change Permissions 23
5.3 Setting Permissions Shorthand 24
5.4 Permissions for Directories 24
5.5 Basic Security 25
6 FILTERS **26**
6.1 What Are Filters For? 26
6.2 Print First Lines – head 27
6.3 Print Last Lines – tail 27
6.4 Sort Lines – sort 28
6.5 Count Lines – nl 28
6.6 Count Words – wc 29
6.7 Search and Replace – sed 29
7 REDIRECTION **30**
7.1 Data Streams 30
7.2 Redirecting to a File 31
7.3 Saving to an Existing File 31
7.4 Redirecting from a File 32
7.5 Redirecting STDERR 32
7.6 Piping 33

8 WILDCARDS **34**
8.1 Basics of Wildcards 34
8.2 Use of Wildcards 36
9 REGULAR EXPRESSIONS **36**
9.1 RE's and Grepping 36
9.2 Learning Regular Expressions 37
9.3 Regular Expression Syntax 38
APPENDIX A. LABORATORY WORK №1 .... **39**
APPENDIX B. LABORATORY WORK №2 .... **40**
APPENDIX C. LABORATORY WORK №3 .... **40**
APPENDIX D. LABORATORY WORK №4 .... **41**
APPENDIX E. SHORT HOW-TO'S **41**
APPENDIX F. THE REPORT **42**
REFERENCES **43**

1 COMMANDS

Linux has a graphical user interface and it works pretty much like the GUI's on other systems that you are familiar with such as Windows and OSX. This tutorial won't focus on these. This tutorial will focus instead on the command line (also known as a terminal) running Bash.

1.1 Command Line Interface. Terminal.

The command line is an interesting beast, and if you've not used one before, can be a bit daunting. Don't worry, with a bit of practice you'll soon come to see it as your friend. Don't think of it as leaving the GUI behind so much as adding to it. While you can leave the GUI alltogether, most people open up a command line interface just as another window on their desktop (in fact you can have as many open as you like). This is also to our advantage as we can have several command lines open and doing different tasks in each at the same time. We can also easily jump back to the GUI when it suits us. Experiment until you find the setup that suits you best. As an example I will typically have 3 terminals open: 1 in which I do my working, another to bring up ancilliary data and a final one for viewing Manual pages (more on these later).

A **command line**, or **terminal**, is a text based interface to the system. You are able to enter commands by typing them on the keyboard and feedback will be given to you similarly as text.

The command line typically presents you with a prompt. As you type, it will be displayed after the prompt. Most of the time you will be issuing commands. Here is an example:

```
user@ubuntu:~$ ls -l /home/user
total 3
drwxr-xr-x  2 user user 4096 Mar 23 13:34 bin
drwxr-xr-x 18 user user 4096 Feb 17 09:12 Documents
drwxr-xr-x  2 user user 4096 May 05 17:25 public_html
user@ubuntu:~$ cat textfile
user@ubuntu:~$ echo message
```

Let's break it down:

Line 1 presents us with a prompt *user@ubuntu:~$*. It contains the following parts: *user* – the name of current user, *ubuntu* – the name of your machine, ~ – current working directory (see Section 2.2).

After that we entered a **command** (*ls*). Typically a command is always the first thing you type. After that we have what are referred to as **command line arguments** (*-l /home/user*). Important to note, <u>these are separated by spaces</u> (there must be a space between the command and the first command line argument also). The first command line argument (*-l*) is also referred to as an **option.** Options are typically used to modify the behaviour of the command. Options are usually listed before other arguments and typically start with a dash (-).

Lines 2 - 5 are output from running the command. Most commands produce output and it will be listed straight under the issuing of the command. Other commands just perform their task and don't display any information unless there was an error.

Line 6 presents us with a prompt again. After the commannd has run and the terminal is ready for you to enter another command the prompt will be displayed. If no prompt is displayed then the command may still be running (you will learn later how to deal with this). Here we execute the command **cat** to view the contents of the file textfile.

Line 7 shows how to print message using **echo** command.

Within a terminal you have what is known as a **shell**. This is a part of the operating system that defines how the terminal will behave and looks after running (or executing) commands for you. There are various shells available but the most common one is called **bash** which stands for *Bourne again shell*. This tutorial will assume you are using bash as your shell.

The terminal may seem daunting but don't fret. Linux is full of **shortcuts** to help make your life easier. You'll be introduced to several of them throughout this tutorial. Take note of them as not only do they make your life easier, they often also save you from making silly mistakes such as typos.

Here's your first shortcut. When you enter commands, they are actually stored in a history. You can traverse this history using the up and down arrow keys. So don't bother re-typing out commands you have previously entered, you can usually just hit the up arrow a few times. You can also edit these commands using the left and right arrow keys to move the cursor where you want.

1.2 Manual Pages

The manual pages are a set of pages that explain every command available on your system including what they do, the specifics of how you run them and what command line arguments they accept. Some of them are a little hard to get your head around but they are fairly consistent in their structure so once you get the hang of it it's not too bad. You invoke the manual pages with the following command:

```
man <command to look up>
```

```
user@ubuntu:~$ man ls
Name
    ls - list directory contents

Synopsis
    ls [option] ... [file] ...

Description
    List information about the FILEs (the current directory by de-
fault). Sort entries alphabetically if none of -cftuvSUX nor --
sort is specified.

    Mandatory arguments to long options are mandatory for short
options too.

    -a, --all
        do not ignore entries starting with .

    -A, --almost-all
        do not list implied . and ..

...
```

Let's break it down:

Line 3 tells us the actual command followed by a simple one line description of it's function.

Lines 6 is what's called the synopsis. This is really just a quick overview of how the command should be run. Square brackets ([ ]) indicate that something is optional. (option on this line refers to the command line options listed below the description).

Line 9 presents us with a more detailed description of the command.

Line 11 onwards Below the description will always be a list of all the command line options that are avaialable for the command.

To exit the man pages press 'q' for quit.

It is possible to do a keyword search on the Manual pages. This can be helpful if you're not quite sure of what command you may want to use but you know what you want to achieve. To be effective with this approach, you may need a few goes. It is not uncommon to find that a particular word exists in many manual pages.

```
man -k <search term>
```

If you want to search within a manual page this is also possible. To do this, whilst you are in the particular manual page you would like to search press forward slash '/' followed by the term you would like to search for and hit 'enter' If the term appears multiple times you may cycle through them by pressing the 'n' button for next.

2 FILE SYSTEM

Let's learn the basics of moving around the system. Many tasks rely on being able to get to, or reference the correct location in the system. As such, this stuff really forms the foundation of being able to work effectively in Linux. Make sure you understand it well.

2.1 File System Navigation

The first command we are going to learn is **pwd** which stands for Present Working Directory. (You'll find that a lot of commands in linux are named as an abbreviation of a word or words describing them. This makes it easier to remember them.) The command does just that. It tells you what your current or present working directory is. Give it a try now.

```
user@ubuntu:~$ pwd
/home/user
```

A lot of commands on the terminal will rely on you being in the right location. As you're moving around, it can be easy to lose track of where you are at. Make use of this command often so as to remind yourself where you presently are.

It's one thing to know where we are. Next we'll want to know what is there. The command for this task is **ls**. It's short for list. Let's give it a go.

```
user@ubuntu:~$ ls
bin Documents public_html
```

Whereas pwd is just run by itself with no arguments, ls is a little more powerful. We have run it here with no arguments in which case it will just do a plain listing of our current location. We can do more with ls however. Below is an outline of it's usage:

```
ls [options] [location]
```

In the above example, the square brackets ([ ]) mean that those items are optional, we may run the command with or without them. In the terminal below I have run ls in a few different ways to demonstrate.

```
user@ubuntu:~$ ls
bin Documents public_html
user@ubuntu:~$ ls -l
total 3
drwxr-xr-x  2 user user 4096 Mar 23 13:34 bin
drwxr-xr-x 18 user user 4096 Feb 17 09:12 Documents
drwxr-xr-x  2 user user 4096 May 05 17:25 public_html
user@ubuntu:~$ ls /etc
a2ps.cfg aliases alsa.d cups fonts my.conf systemd
...
user@ubuntu:~$ ls -l /etc
total 3
-rwxr-xr-x  2 root root 123 Mar 23 13:34 a2ps.cfg
-rwxr-xr-x 18 root root 78 Feb 17 09:12 aliases
drwxr-xr-x  2 user user 4096 May 05 17:25 alsa.d
...
```

Let's break it down:

Line 1 - We ran ls in it's most basic form. It listed the contents of our current directory.

Line 4 - We ran ls with a single command line option (-l) which indicates we are going to do a long listing. A long listing has the following:

First character indicates whether it is a normal file (-) or directory (d)

Next 9 characters are permissions for the file or directory (we'll learn more about them in section 6).

The next file is the number of blocks (don't worry too much about this).

The next field is the owner of the file or directory (user in this case).

The next field is the group the file or directory belongs to (users in this case).

Following this is the file size.

Next up is the file modification time.

Finally we have the actual name of the file or directory.

Line 10 - We ran ls with a command line argument (/etc). When we do this it tells ls not to list our current directory but instead to list that directories contents.

Line 13 - We ran ls with both a command line option and argument. As such it did a long listing of the directory /etc.

Lines 12 and 18 just indicate that I have cut out some of the commands normal output for brevities sake. When you run the commands you will see a longer listing of files and directories.

2.2 Paths

In the previous commands we started touching on something called a path. I would like to go into more detail on them now as they are important in being proficient with Linux. Whenever we refer to either a file or directory on the command line, we are in fact referring to a path. ie. A path is a means to get to a particular file or directory on the system.

There are 2 types of paths we can use, **absolute** and **relative**. Whenever we refer to a file or directory we are using one of these paths. Whenever we refer to a file or directory, we can, in fact, use either type of path (either way, the system will still be directed to the same location).

To begin with, we have to understand that the file system under linux is a heirarchical structure. At the very top of the structure is what's called **the root directory**. It is denoted by a single slash (/). It has subdirectories, they have subdirectories and so on. Files may reside in any of these directories.

Absolute paths specify a location (file or directory) in relation to the root directory. You can identify them easily as they always begin with a /

Relative paths specify a location (file or directory) in relation to where we currently are in the system. They will not begin with a slash.

Here's an example to illustrate:

```
user@ubuntu:~$ pwd
/home/user
user@ubuntu:~$ ls Documents
file1.txt file2.txt file3.txt
...
user@ubuntu:~$ ls /home/user/Documents
file1.txt file2.txt file3.txt
...
```

Line 1 - We ran pwd just to verify where we currently are.

Line 4 - We ran ls providing it with a relative path. Documents is a directory in our current location. This command could produce different results depending on where we are. If we had another user on the system, bob, and we ran the command when in their home directory then we would list the contents of their Documents directory instead.

Line 7 - We ran ls providing it with an absolute path. This command will provide the same output regardless of our current location when we run it.

You'll find that a lot of stuff in Linux can be achieved in several different ways. Paths are no different. Here are some more building blocks you may use to help build your paths.

~ (tilde) - This is a shortcut for your home directory. eg, if your home directory is /home/user then you could refer to the directory Documents with the path /home/user/Documents or ~/Documents

. (dot) - This is a reference to your current directory. eg in the example above we referred to Documents on line 4 with a relative path. It could also be written as ./Documents (Normally this extra bit is not required but in later sections we will see where it comes in handy).

.. (dotdot)- This is a reference to the parent directory. You can use this several times in a path to keep going up the heirarchy. eg if you were in the path /home/user you could run the command ls ../../ and this would do a listing of the root directory.

So now you are probably starting to see that we can refer to a location in a variety of different ways. Some of you may be asking the question, which one should I use? The answer is that you can use any method you like to refer to a location. Whenever you refer to a file or directory on the command line you are actually referring to a path and your path can be constructed using any of these elements. The best approach is whichever is the most convenient for you. Here are some examples:

```
user@ubuntu:~$ pwd
/home/user
user@ubuntu:~$ ls ~/Documents
file1.txt file2.txt file3.txt
...
user@ubuntu:~$ ls ./Documents
file1.txt file2.txt file3.txt
...
user@ubuntu:~$ ls /home/user/Documents
file1.txt file2.txt file3.txt
...
```

```
user@ubuntu:~$ ls ../../
bin boot dev etc home lib var
...
user@ubuntu:~$ ls /
bin boot dev etc home lib var
...
```

After playing about with these on the command line yourself they will start to make a bit more sense. Make sure you understand how all of these elements of building a path work as you'll use all of them in future sections.

2.3 Let's Move Around a Bit

In order to move around in the system we use a command called **cd** which stands for change directory. It works as follows:

cd [location]

If you run the command cd without any arguments then it will always take you back to your home directory.

The command cd may be run without a location as we saw in the shortcut above but usually will be run with a single command line argument which is the location we would like to change into. The location is specified as a path and as such may be specified as either an absolute or relative path and using any of the path building blocks mentioned above. Here are some examples.

```
user@ubuntu:~$ pwd
/home/user
user@ubuntu:~$ cd Documents
user@ubuntu:~$ ls
file1.txt file2.txt file3.txt
...
user@ubuntu:~$ cd /
user@ubuntu:~$ pwd
/
user@ubuntu:~$ ls
bin boot dev etc home lib var
...
user@ubuntu:~$ cd ~/Documents
user@ubuntu:~$ pwd
/home/user/Documents
user@ubuntu:~$ cd ../../
user@ubuntu:~$ pwd
/home
user@ubuntu:~$ cd
user@ubuntu:~$ pwd
/home/user
```

Typing out these paths can become tedious. If you're like me, you're also prone to making typos. The command line has a nice little mechanism to help us in this respect. It's called **Tab Completion.**

When you start typing a path (anywhere on the command line, you're not just limited to certain commands) you may hit the *Tab key* on your keyboard at any time which will invoke an auto complete action. If nothing happens then that means there are several possibilities. If you hit Tab again it will show you those possibilities. You may then continue typing and hit Tab again and it will again try to auto complete for you.

It's kinda hard to demonstrate here so it's probably best if you try it yourself. If you start typing cd /h*Tab*/<beginning of your username>*Tab* you'll get a feel for how it works.

2.4 Everything is a File

Ok, the first thing we need to appreciate with linux is that under the hood, everything is actually a file. A text file is a file, a directory is a file, your keyboard is a file (one that the system reads from only), your monitor is a file (one that the system writes to only) etc. To begin with, this won't affect what we do too much but keep it in mind as it helps with understanding the behaviour of Linux as we manage files and directories.

Linux is an Extensionless System

This one can sometimes be hard to get your head around but as you work through the sections it will start to make more sense. A file extension is normally a set of 2 - 4 characters after a full stop at the end of a file, which denotes what type of file it is. The following are common extensions:

```
file.exe - an executable file, or program.
file.txt - a plain text file.
file.png, file.gif, file.jpg - an image.
```

In other systems such as Windows the extension is important and the system uses it to determine what type of file it is. Under Linux the system actually ignores the extension and looks inside the file to determine what type of file it is. So for instance I could have a file myself.png which is a picture of me. I could rename the file to myself.txt or just myself and Linux would still happily treat the file as an image file. As such it can sometimes be hard to know for certain what type of file a particular file is. Luckily there is a command called file which we can use to find this out.

```
file [path]
```

Now you may be wondering why I specified the command line argument above as path instead of file. If you remember from the previous section, whenever we specify a file or directory on the command line it is actually a path. Also because directories (as mentioned above) are actually just a special type of file, it would be more accurate to say that a path is a means to get to a particular location in the system and that location is a file.

Linux is Case Sensitive

This is very important and a common source of problems for people new to Linux. Other systems such as Windows are case insensitive when it comes to referring to files. Linux is not like this. As such it is possible to have two or more files and directories with the same name but letters of different case.

```
user@ubuntu:~$ ls Documents
FILE1.txt File1.txt file1.TXT
...
user@ubuntu:~$ file Documents/file1.txt
Documents/file1.txt: ERROR: cannot open 'file1.txt' (No such file
or directory)
```

Linux sees these all as distinct and separate files.

Also be aware of case sensitivity when dealing with command line options. For instance with the command ls there are two options s and S both of which do different things. A common mistake is to see an option which is upper case but enter it as lower case and wonder why the output doesn't match your expectation.

Spaces in names

Spaces in file and directory names are perfectly valid but we need to be a little careful with them. As you would remember, a space on the command line is how we seperate items. They are how we know what is the program name and can identify each command line argument. If we wanted to move into a directory called Holiday Photos for example the following would not work.

```
user@ubuntu:~$ ls Documents
FILE1.txt File1.txt file1.TXT Holiday Photos
...
user@ubuntu:~$ cd Holiday Photos
bash: cd: Holiday: No such file or directory
```

What happens is that Holiday Photos is seen as two command line arguments. cd moves into whichever directory is specified by the first command line argument only. To get around this we need to identify to the terminal that we wish Holiday Photos to be seen as a single command line argument. There are two ways to go about this, either way is just as valid.

Quotes

The first approach involves using quotes around the entire item. You may use either single or double quotes (later on we will see that there is a subtle difference between the two but for now that difference is not a problem). Anything inside quotes is considered a single item.

```
user@ubuntu:~$ cd 'Holiday Photos'
user@ubuntu:~$ pwd
/home/user/Documents/Holiday Photos
```

Escape Characters

Another method is to use what is called an escape character, which is a backslash (\ ). What the backslash does is escape (or nullify) the special meaning of the next character.

```
user@ubuntu:~$ cd Holiday\ Photos
user@ubuntu:~$ pwd
/home/user/Documents/Holiday Photos
```

In the above example the space between Holiday and Photos would normally have a special meaning which is to separate them as distinct command line arguments. Because we placed a backslash in front of it, that special meaning was removed.

In the previous section we learnt about something called Tab Completion. If you use that before encountering the space in the directory name then the terminal will automatically escape any spaces in the name for you.

Hidden Files and Directories

Linux actually has a very simple and elegant mechanism for specifying that a file or directory is hidden. If the file or directory's name begins with a . (full stop) then it is considered to be hidden. You don't even need a special command or action to make a file hidden. Files and di-

rectories may be hidden for a variety of reasons. Configuration files for a particular user (which are normally stored in their home directory) are hidden for instance so that they don't get in the way of the user doing their everyday tasks.

To make a file or directory hidden all you need to do is create the file or directory with it's name beginning with a . or rename it to be as such. Likewise you may rename a hidden file to remove the . and it will become unhidden. The command ls which we have seen in the previous section will not list hidden files and directories by default. We may modify it by including the command line option -a so that it does show hidden files and directories.

```
user@ubuntu:~$ ls Documents
FILE1.txt File1.txt file1.TXT
...
user@ubuntu:~$ ls -a Documents
. .. FILE1.txt File1.txt file1.TXT .hidden .file.txt
...
```

In the above example you will see that when we listed all items in our current directory the first two items were . and .. If you're unsure what these are then you may wish to have a read over our previous section on Paths.

2.5 Filesystem Hierarchy Standard

The **Filesystem Hierarchy Standard (FHS)** defines the directory structure and directory contents in Unix and Unix-like operating systems. It is maintained by the Linux Foundation. The latest version is 3.0, released on 3 June 2015. Currently it is only used by Linux distributions.

In the FHS all files and directories appear under the root directory /, even if they are stored on different physical or virtual devices. Note however that some of these directories may or may not be present on a Unix system depending on whether certain subsystems, such as the X Window System, are installed.

The majority of these directories exist in all UNIX operating systems and are generally used in much the same way; however, the descriptions here are those used specifically for the FHS, and are not considered authoritative for platforms other than Linux.

Main directories in FHS:

`/`

Primary hierarchy root and root directory of the entire file system hierarchy.

`/bin`

Essential command binaries that need to be available in single user mode; for all users, e.g., cat, ls, cp.

`/boot`

Boot loader files, e.g., kernels, initrd.

`/dev`

Essential devices, e.g., /dev/null.

`/etc`

Host-specific system-wide configuration files

There has been controversy over the meaning of the name itself. In early versions of the UNIX Implementation Document from Bell labs, /etc is referred to as the etcetera directory, as this directory historically held everything that did not belong elsewhere (however, the FHS restricts /etc to static configuration files and may not contain binaries). Since the publication of early documentation, the directory name has been re-explained in various ways. Recent interpretations include backronyms such as "Editable Text Configuration" or "Extended Tool Chest".

`/home`

Users' home directories, containing saved files, personal settings, etc.

`/lib`

Libraries essential for the binaries in /bin/ and /sbin/.

`/media`

Mount points for removable media such as CD-ROMs (appeared in FHS-2.3).

`/mnt`

Temporarily mounted filesystems.

`/opt`

Optional application software packages.

`/proc`

Virtual filesystem providing process and kernel information as files. In Linux, corresponds to a procfs mount.

`/root`

Home directory for the root user.

`/run`

Run-time variable data: Information about the running system since last boot, e.g., currently logged-in users and running daemons.

`/sbin`

Essential system binaries, e.g., fsck, init, route.

`/srv`

Site-specific data which are served by the system.

`/tmp`

Temporary files (see also /var/tmp). Often not preserved between system reboots, and may be severely size restricted.

`/usr`

Secondary hierarchy for read-only user data; contains the majority of (multi-)user utilities and applications.

`/usr/bin`

Non-essential command binaries (not needed in single user mode); for all users.

`/usr/include`

Standard include files.

`/usr/lib`

Libraries for the binaries in /usr/bin/ and /usr/sbin/.

`/usr/local`

Tertiary hierarchy for local data, specific to this host. Typically has further subdirectories, e.g., bin/, lib/, share/.

`/usr/sbin`

Non-essential system binaries, e.g., daemons for various network-services.

`/usr/share`

Architecture-independent (shared) data.

`/usr/src`

Source code, e.g., the kernel source code with its header files.

`/var`

Variable files—files whose content is expected to continually change during normal operation of the system—such as logs, spool files, and temporary e-mail files.

`/var/cache`

Application cache data. Such data are locally generated as a result of time-consuming I/O or calculation. The application must be able to regenerate or restore the data. The cached files can be deleted without loss of data.

`/var/log`

Log files. Various logs.

`/var/tmp`

Temporary files to be preserved between reboots.

3 DIRECTORIES AND FILES

In this section we shall learn the most often used commands. These commands create, move, copy and remove files and directories.

3.1 Making a Directory

Linux organises it's file system in a heirarchical way. Over time you'll tend to build up a fair amount of data (storage capacities are always increasing). It's important that we create a directory structure that will help us organise that data in a manageable way. I've seen way too many people just dump everything directly at the base of their home directory and waste a lot of their time trying to find what they are after amongst 100's (or even 1000's) of other files. Develop the habit of organising your stuff into an elegant file structure now and you will thank yourself for years to come.

Creating a directory is pretty easy. The command we are after is mkdir which is short for Make Directory.

```
mkdir [options] <Directory>
```

In it's most basic form we can run mkdir supplying only a directory and it will create it for us.

```
user@ubuntu:~$ pwd
/home/user
user@ubuntu:~$ ls
bin Documents public_html
user@ubuntu:~$ mkdir linuxtutorialwork
user@ubuntu:~$ ls
bin Documents linuxtutorialwork public_html
```

Let's break it down:

Line 1 Let's start off by making sure we are where we think we should be. (In the example above I am in my home directory)

Lines 2 We'll do a quick listing so we know what is already in our directory.

Line 7 Run the command mkdir and create a directory linuxtutorialwork (a nice place to put further work we do relating to this tutorial just to keep it separate from our other stuff).

Remember that when we supply a directory in the above command we are actually supplying a path. Is the path we specified relative or absolute? Here are a few more examples of how we can supply a directory to be created

```
user@ubuntu:~$ mkdir /home/user/foo
user@ubuntu:~$ mkdir ./blah
user@ubuntu:~$ mkdir ../dir1
user@ubuntu:~$ mkdir ~/linuxtutorialwork/dir2
```

If these don't make sense then review section Navigation

There are a few useful options available for mkdir. Can you remember where we may go to find out the command line options a particular command supports?

The first one is -p which tells mkdir to make parent directories as needed (demonstration of what that actually means below). The second one is -v which makes mkdir tell us what it is doing (as you saw in the example above, it normally does not).

```
user@ubuntu:~$ mkdir -p linuxtutorialwork/foo/bar
user@ubuntu:~$ cd linuxtutorialwork/foo/bar
user@ubuntu:~$ pwd
/home/user/linuxtutorialwork/foo/bar
```

And now the same command but with the -v option

```
user@ubuntu:~$ mkdir -pv linuxtutorialwork/foo/bar
mkdir: created directory 'linuxtutorialwork/foo'
mkdir: created directory 'linuxtutorialwork/foo/bar'
user@ubuntu:~$ cd linuxtutorialwork/foo/bar
user@ubuntu:~$ pwd
/home/user/linuxtutorialwork/foo/bar
```

3.2 Removing a Directory

Creating a directory is pretty easy. Removing or deleting a directory is easy too. One thing to note, however, is that there is no undo when it comes to the command line on Linux (Linux GUI desktop environments typically do provide an undo feature but the command line does not). Just be careful with what you do. The command to remove a directory is rmdir, short for remove directory.

rmdir [options] <Directory>

Two things to note. Firstly, rmdir supports the -v and -p options similar to mkdir. Secondly, a directory must be empty before it may be removed (later on we'll see a way to get around this).

```
user@ubuntu:~$ rmdir linuxtutorialwork/foo/bar
user@ubuntu:~$ ls linuxtutorialwork/foo
```

3.3 Creating a Blank File

A lot of commands that involve manipulating data within a file have the nice feature that they will create a file automatically if we refer to it and it does not exist. In fact we can make use of this very characteristic to create blank files using the command touch.

touch [options] <filename>

```
user@ubuntu:~$ pwd
/home/user/linuxtutorialwork
user@ubuntu:~$ ls
foo
user@ubuntu:~$ touch example1
user@ubuntu:~$ ls
example1 foo
```

touch is actually a command we may use to modify the access and modification times on a file (normally not needed but sometimes when you're testing a system that relies on file access or modification times it can be useful). I know some students have tried using this command to make it look like their assignment files were not modified after the due date (there are means to detect this so it never works but points for being creative). What we are taking advantage of here is that if we touch a file and it does not exist, the command will do us a favor and automatically create it for us.

Many things in Linux are not done directly but by knowing the behaviour of certain commands and aspects of the system and using them in creative ways to achieve the desired outcome.

Remember in the introduction we talked about the command line as providing you with a series of building blocks. You are free to use these building blocks in any way you like but you can really only do this effectively if you understand how they do their function as well as why.

At the moment the file is blank which is kinda boring but in coming sections we'll look at putting data into files and extracting data from them.

3.4 Copying a File or Directory

There are many reasons why we may want to make a duplicate of a file or directory. Often before changing something, we may wish to create a duplicate so that if something goes wrong we can easily revert back to the original. The command we use for this is cp which stands for copy.

```
cp [options] <source> <destination>
```

There are quite a few options available to cp. I'll intoduce one of them further below but it's worth checking out the man page for cp to see what else is available.

```
user@ubuntu:~$ ls
example1 foo
user@ubuntu:~$ cp example1 barney
user@ubuntu:~$ ls
barney example1 foo
```

Note that both the source and destination are paths. This means we may refer to them using both absolute and relative paths. Here are a few examples:

```
user@ubuntu:~$ cp /home/user/linuxtutorialwork/example2 example3
user@ubuntu:~$ cp example2 ../../backups
user@ubuntu:~$ cp ecample2 ../../backups/example4
user@ubuntu:~$ cp /home/user/linuxtutorialwork/example2 /otherdir/foo/example5
```

When we use cp the destination can be a path to either a file or directory. If it is to a file (such as examples 1, 3 and 4 above) then it will create a copy of the source but name the copy the filename specified in destination. If we provide a directory as the destination then it will copy the file into that directory and the copy will have the same name as the source.

In it's default behaviour cp will only copy a file (there is a way to copy several files in one go but we'll get to that in section 6. Wildcards). Using the -r option, which stands for recursive, we may copy directories. Recursive means that we want to look at a directory and all files and directories within it, and for subdirectories, go into them and do the same thing and keep doing this.

```
user@ubuntu:~$ ls
barney example1 foo
user@ubuntu:~$ cp foo foo2
cp: omitting directory 'foo'
user@ubuntu:~$ cp -r foo foo2
user@ubuntu:~$ ls
barney example1 foo foo2
```

In the above example any files and directories within the directory foo will also be copied to foo2.

3.5 Moving a File or Directory

To move a file we use the command mv which is short for move. It operates in a similar way to cp. One slight advantage is that we can move directories without having to provide the -r option.

```
mv [options] <source> <destination>
```

```
user@ubuntu:~$ ls
barney example1 foo foo2
user@ubuntu:~$ mkdir backups
user@ubuntu:~$ mv foo2 backups/foo3
user@ubuntu:~$ mv barney backups/
user@ubuntu:~$ ls
backups example1 foo
```

Let's break it down:

Line 3 We created a new directory called backups.

Line 4 We moved the directory foo2 into the directory backups and renamed it as foo3

Line 7 We moved the file barney into backups. As we did not provide a destination name, it kept the same name.

Note that again the source and destination are paths and may be referred to as either absolute or relative paths.

3.6 Renaming Files and Directories

Now just as above with the command touch, we can use the basic behaviour of the command **mv** in a creative way to achieve a slighly different outcome. Normally mv will be used to move a file or directory into a new directory. As we saw on line 4 above, we may provide a new name for the file or directory and as part of the move it will also rename it. Now if we specify the destination to be the same directory as the source, but with a different name, then we have effectively used mv to rename a file or directory.

```
user@ubuntu:~$ ls
backups example1 foo
user@ubuntu:~$ mv foo foo3
user@ubuntu:~$ ls
backups example1 foo3
user@ubuntu:~$ cd ..
user@ubuntu:~$ mkdir linuxtutorialwork/testdir
user@ubuntu:~$ mv linuxtutorialwork/testdir
/home/user/linuxtutorialwork/fred
user@ubuntu:~$ ls
backups example1 foo3 fred
```

Let's break it down:

Line 3 We renamed the file foo to be foo3 (both paths are relative).

Line 6 We moved into our parent directory. This was done only so in the next line we can illustrate that we may run commands on files and directories even if we are not currently in the directory they reside in.

Line 8 We renamed the directory testdir to fred (the source path was a relative path and the destination was an aboslute path).

3.7 Removing a File

As with rmdir, removing a file is an action that may not be undone so be careful. The command to remove or delete a file is rm which stands for remove.

```
rm [options] <file>
user@ubuntu:~$ ls
backups example1 foo3 fred
user@ubuntu:~$ rm example1
user@ubuntu:~$ ls
backups foo3 fred
```

3.8 Removing non empty Directories

Like several other commands introduced in this section, rm has several options that alter it's behaviour. I'll leave it up to you to look at the man page to see what they are but I will introduce one particularly useful option which is -r. Similar to cp it stands for recursive. When rm is run with the -r option it allows us to remove directories and all files and directories contained within.

```
user@ubuntu:~$ ls
backups foo3 fred
user@ubuntu:~$ rmdir backups
rmdir: failed to remove 'backups': Directory not empty
user@ubuntu:~$ rm backups
rm: cannot remove 'backups': Is a directory
user@ubuntu:~$ rm -r backups
user@ubuntu:~$ ls
foo3 fred
```

A good option to use in combination with r is i which stands for interactive. This option will prompt you before removing each file and directory and give you the option to cancel the command.

4 LINKS

A link provides a connection between files. This provides the ability to have a single file or directory referred to through different names. The nearest comparison with the Windows world is of a shortcut, but that is an unfair comparison as a link in Linux is far more powerful. A Windows shortcut is just a way of launching a file from a different place, whereas a link can make a file appear in multiple locations which is invisible to the applications.

There are two types of links that can be created. The first is a hard link and the other is a soft link (sometimes called symbolic link or symlink). The command to create these is the same - ln.

4.1 Hard link

A hard link creates a second file that refers to the same file on the physical disk. This is achieved by having two filenames that point directly at the same file. This is normally used where file entries (links) are on the same filesystem. When a hard link is created then all the names that link to that file are given the same status. Deleting one of the files will break the link, but the file can still exist under the other linked filenames. This works by maintaining a counter of the number of filenames that the file has. When the number of filenames reaches zero then the file is considered to be deleted and is removed.

The number of filenames for a file can be seen using the ls -l command. The number following the file permissions indicates the number of linked filenames. The following screenshot shows that filename1 and filename2 are to linked files with one of the file denoted by the 2 (in this case the same file, but they could be to completely different files), the file not_link is a single file denoted by the 1.

```
user@ubuntu:~$ ls -l
total 2432
-rw-r--r-- 2 stewart stewart 1241088 2009-01-23 15:26 filename1
-rw-r--r-- 2 stewart stewart 1241088 2009-01-23 15:26 filename2
-rw-r--r-- 1 stewart stewart       0 2009-01-23 15:26 not_link
```

Note that whilst the two files appear to occupy 1.2Mb each the actual space used is only 1.2Mb in total as the file only exists once.

```
user@ubuntu:~$ du -h
1.2M
```

The default for the ln command is a hard link. Assuming filename1 already exists filename2 is created using:

```
user@ubuntu:~$ ln filename1 filename2
```

4.2 Soft Link

A soft link is sometimes referred to as a **symbolic link** or **symlink**. A filename created as a soft link is a special file that has the pathname of the file to redirect to. Behind the scenes when you try and access a symlink it just goes to the filename referred to instead.

In the following example a file has been created called original_file with a soft link to that same file called softlink_tofile. As you can see the ls command makes it clear that this is a link through the l at the beginning of the file permissions and due to the reference notation after the filename.

```
user@ubuntu:~$ ls -l
total 440
-rw-r--r-- 1 stewart stewart 446464 2009-01-23 15:21 original_file
lrwxrwxrwx 1 stewart stewart 13 2009-01-23 15:20 sof-
tlink_tofile -> original_file
```

Note that with a soft link the permissions to the link file are set to full access as the user is contstrained by the permissions on the original file. Also note that if the filesize of the original file changes the link will remain the same (on this system 13 bytes).

If the link is deleted then this will have no impact on the original file, but if the original file is deleted this will result in a broken link. The example below shows how removing the original file results in a error "No such file or directory".

```
user@ubuntu:~$ rm original_file
user@ubuntu:~$ ls -l
total 0
lrwxrwxrwx 1 stewart stewart 13 2009-01-23 15:20 softlink_tofile -
> original_file
user@ubuntu:~$ cat softlink_tofile
cat: softlink_tofile: No such file or directory
```

The -s option is used on the ln command to create a softlink.

```
user@ubuntu:~$ ln -s original_file softlink_tofile
```

Note that if original_file does not exist then the soft link will be created anyway. An attempt to read it will give the error message we encountered earlier, but an attempt to write to the file can create original_file.

4.3 The Difference Between Soft and Hard Links

Hard links:

- Only link to a file not a directory
- Can not reference a file on a different disk/volume
- Links will reference a file even if it is moved
- Links reference inode/physical locations on the disk. File inode can be examined using **ls -i**

Symbolic (soft) links:

- Can link to directories
- Can reference a file/folder on a different hard disk/volume
- Links remain if the original file is deleted
- Links will NOT reference the file anymore if it is moved
- Links reference abstract filenames/directories and NOT physical locations. They are given their own inode

5 FILE PERMISSIONS

In this section we'll learn about how to set Linux permissions on files and directories. Permissions specify what a particular person may or may not do with respect to a file or directory. As such, permissions are important in creating a secure environment. For instance you don't want other people to be changing your files and you also want system files to be safe from damage (either accidental or deliberate). Luckily, permissions in a Linux system are quite easy to work with.

5.1 Types of Permissions

Linux permissions dictate 3 things you may do with a file: **read, write** and **execute**. They are referred to in Linux by a single letter each.

- **r read** - you may view the contents of the file.
- **w write** - you may change the contents of the file.
- **x execute** - you may execute or run the file if it is a program or script.

For every file we define 3 sets of people for whom we may specify permissions.

- **owner** - a single person who owns the file. (typically the person who created the file but ownership may be granted to some one else by certain users)
- **group** - every file belongs to a single group.
- **others** - everyone else who is not in the group or the owner.

Three permissions and three groups of people. That's about all there is to permissions really. Now let's see how we can view and change them.

To view permissions for a file we use the long listing option for the command ls.

ls -l [path]

```
user@ubuntu:~$ ls -l frog.png
-rwxr----x 1 harry users 2.7K Jan 4 07:32 frog.png
```

In the above example the first 10 characters of the output are what we look at to identify permissions.

The first character identifies the file type. If it is a dash (-) then it is a normal file. If it is a d then it is a directory.

The following 3 characters represent the persmissions for the owner. A letter represents the presence of a permission and a dash (-) represents the absence of a permission. In this example the owner has all permissions (read, write and execute).

The following 3 characters represent the permissions for the group. In this example the group has the ability to read but not write or execute. Note that the order of permissions is always read, then write then execute.

Finally the last 3 characters represent the permissions for others (or everyone else). In this example they have the execute permission and nothing else.

5.2 Change Permissions

To change permissions on a file or directory we use a command called chmod It stands for change file mode bits which is a bit of a mouthfull but think of the mode bits as the permission indicators.

chmod [permissions] [path]

chmod has permission arguments that are made up of 3 components:

- Who are we changing the permission for? [ugoa] - user (or owner), group, others, all
- Are we granting or revoking the permission - indicated with either a plus (+) or minus (-)
- Which permission are we setting? - read (**r**), write (**w**) or execute (**x**)

The following examples will make their usage clearer.

Grant the execute permission to the group. Then remove the write permission for the owner.

```
user@ubuntu:~$ ls -l frog.png
-rwxr----x 1 harry users 2.7K Jan 4 07:32 frog.png
user@ubuntu:~$ chmod g+x frog.png
user@ubuntu:~$ ls -l frog.png
-rwxr-x--x 1 harry users 2.7K Jan 4 07:32 frog.png
user@ubuntu:~$ chmod u-w frog.png
user@ubuntu:~$ ls -l frog.png
-r-xr-x--x 1 harry users 2.7K Jan 4 07:32 frog.png
```

Don't want to assign permissions individually? We can assign multiple permissions at once.

```
user@ubuntu:~$ ls -l frog.png
-rwxr----x 1 harry users 2.7K Jan 4 07:32 frog.png
user@ubuntu:~$ chmod g+wx frog.png
user@ubuntu:~$ ls -l frog.png
-rwxrwx--x 1 harry users 2.7K Jan 4 07:32 frog.png
user@ubuntu:~$ chmod go-x frog.png
user@ubuntu:~$ ls -l frog.png
-rwxrw---- 1 harry users 2.7K Jan 4 07:32 frog.png
```

It may seem odd that as the owner of a file we can remove our ability to read, write and execute that file but there are valid reasons we may wish to do this. Maybe we have a file with data in it we wish not to accidentally change for instance. While we may remove these permissions, we may not remove our ability to set those permissions and as such we always have control over every file under our ownership.

5.3 Setting Permissions Shorthand

The method outlined above is not too hard for setting permissions but it can be a little tedious if we have a specific set of permissions we sould like to apply regularly to certain files. Luckily, there is a shorthand way to specify permissions that makes this easy.

To understand how this shorthand method works we first need a little background in number systems. Our typical number system is decimal. It is a base 10 number system and as such has 10 symbols (0 - 9) used. Another number system is octal which is base 8 (0-7). Now it just so happens that with 3 permissions and each being on or off, we have 8 possible combinations (2^3). Now we can also represent our numbers using binary which only has 2 symbols (0 and 1). The mapping of octal to binary is in the table below.

Octal	Binary	Letters
0	0 0 0	- - -
1	0 0 1	- - x
2	0 1 0	- w -
3	0 1 1	- w x
4	1 0 0	r - -
5	1 0 1	r - x
6	1 1 0	r w -
7	1 1 1	r w x

Now the interesting point to note is that we may represent all 8 octal values with 3 binary bits and that every possible combination of 1 and 0 is included in it. So we have 3 bits and we also have 3 permissions. If you think of 1 as representing on and 0 as off then a single octal number may be used to represent a set of permissions for a set of people. Three numbers and we can specify permissions for the user, group and others. Let's see some examples. (refer to the table above to see how they match)

```
user@ubuntu:~$ ls -l frog.png
-rw-r----x 1 harry users 2.7K Jan 4 07:32 frog.png
user@ubuntu:~$ chmod 751 frog.png
user@ubuntu:~$ ls -l frog.png
-rwxr-x--x 1 harry users 2.7K Jan 4 07:32 frog.png
user@ubuntu:~$ chmod 240 frog.png
user@ubuntu:~$ ls -l frog.png
--w-r----- 1 harry users 2.7K Jan 4 07:32 frog.png
```

People often remember commonly used number sequences for different types of files and find this method quite convenient. For example 755 or 750 are commonly used for scripts.

5.4 Permissions for Directories

The same series of permissions may be used for directories but they have a slightly different behaviour.

- r - you have the ability to read the contents of the directory (ie do an ls)
- w - you have the ability to write into the directory (ie create files and directories)
- x - you have the ability to enter that directory (ie cd)

Let's see some of these in action

```
user@ubuntu:~$ ls testdir
file1 file2 file3
user@ubuntu:~$ chmod 400 testdir
user@ubuntu:~$ ls -ld testdir
-r-------- 1 ryan users 2.7K Jan 4 07:32 testdir
user@ubuntu:~$ cd testdir
cd: testdir: Permission denied
user@ubuntu:~$ ls testdir
file1 file2 file3
user@ubuntu:~$ chmod 100 testdir
user@ubuntu:~$ ls -ld testdir
---x------ 1 ryan users 2.7K Jan 4 07:32 testdir
user@ubuntu:~$ cd testdir
user@ubuntu:~$ ls testdir
ls: cannot open directory testdir/: Permission denied
```

Note, on lines 5 and 14 above when we ran ls I included the -d option which stands for directory. Normally if we give ls an argument which is a directory it will list the contents of that directory. In this case however we are interested in the permissions of the directory directly and the -d option allows us to obtain that.

These permissions can seem a little confusing at first. What we need to remember is that these permissions are for the directory itself, not the files within. So, for example, you may have a directory which you don't have the read permission for. It may have files within it which you do have the read permission for. As long as you know the file exists and it's name you can still read the file.

```
user@ubuntu:~$ ls -ld testdir
--x------- 1 ryan users 2.7K Jan 4 07:32 testdir
user@ubuntu:~$ cd testdir
user@ubuntu:~$ ls testdir
ls: cannot open directory .: Permission denied
user@ubuntu:~$ cat samplefile.txt
Kyle 20
Stan 11
Kenny 37
```

5.5 Basic Security

On a Linux system there are only 2 people usually who may change the permissions of a file or directory. The owner of the file or directory and the root user. The root user is a superuser who is allowed to do anything and everything on the system. Typically the administrators of a system would be the only ones who have access to the root account and would use it to maintain the system. Typically normal users would mostly only have access to files and directories in their home directory and maybe a few others for the purposes of sharing and collaborating on work and this helps to maintain the security and stability of the system.

Your home directory is your own personal space on the system. You should make sure that it stays that way.

Most users would give themselves full read, write and execute permissions for their home directory and no permissions for the group or others however some people for various reasons may have a slighly different set up.

Normally, for optimal security, you should not give either the group or others write access to your home directory, but execute without read can come in handy sometimes. This allows people to get into your home directory but not allow them to see what is there. An example of when this is used is for personal web pages.

It is typical for a system to run a webserver and allow users to each have their own web space. A common set up is that if you place a directory in your home directory called public_html then the webserver will read and display the contents of it. The webserver runs as a different user to you however so by default will not have access to get in and read those files. This is a situation where it is necessary to grant execute on your home directory so that the webserver user may access the required resources.

6 FILTERS

One of the underlying principles of Linux is that every item should do one thing and one thing only and that we can easily join these items together. Think of it like a set of building blocks that we may put together however we like to build anything we want. In this section we will learn about a few of these building blocks. Then we'll look at how we may build them into more complex creations that can do useful work for us.

6.1 What Are Filters For?

A filter, in the context of the Linux command line, is a program that accepts textual data and then transforms it in a particular way. Filters are a way to take raw data, either produced by another program, or stored in a file, and manipulate it to be displayed in a way more suited to what we are after.

These filters often have various command line options that will modify their behaviour so it is always good to check out the man page for a filter to see what is available.

In the examples below we will be providing input to these programs by a file but in the section Piping and Redirection we'll see that we may provide input via other means that add a lot more power.

Let's dive in and introduce you to some of them. (remember, the examples here will only give you a taste of what is possible with these commands. Make sure you explore and use your creativity to see what else you may do with them.)

For each of the demonstrations below we shall be using the following file as an example. This example file contains a list of content purely to make the examples a bit easier to understand but realise that they will work the same with absolutely any other textual data. Also, remember that the file is actually specified as a path and so you may use absolute and relative paths and also wildcards.

```
user@ubuntu:~$ cat mysampledata.txt
Fred apples 20
Susy oranges 5
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
```

```
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
```

6.2 Print First Lines – head

head is a program that prints the first so many lines of it's input. By default it will print the first 10 lines but we may modify this with a command line argument.

head [-number of lines to print] [path]

```
user@ubuntu:~$ head mysampledata.txt
Fred apples 20
Susy oranges 5
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
```

Above was head's default behaviour. And below is specifying a set number of lines.

```
user@ubuntu:~$ head -4 mysampledata.txt
Fred apples 20
Susy oranges 5
Mark watermellons 12
Robert pears 4
```

6.3 Print Last Lines – tail

tail is the opposite of head. Tail is a program that prints the last so many lines of it's input. By default it will print the last 10 lines but we may modify this with a command line argument.

tail [-number of lines to print] [path]

```
user@ubuntu:~$ tail mysampledata.txt
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
```

Above was tail's default behaviour. And below is specifying a set number of lines.

```
user@ubuntu:~$ tail -3 mysampledata.txt
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
```

6.4 Sort Lines – sort

sort will sort it's input, nice and simple. By default it will sort alphabetically but there are many options available to modify the sorting mechanism. Be sure to check out the manu page to see everything it may do.

sort [-options] [path]

```
user@ubuntu:~$ sort mysampledata.txt
Anne mangoes 7
Betty limes 14
Fred apples 20
Greg pineapples 3
Lisa peaches 7
Mark grapes 39
Mark watermellons 12
Oliver rockmellons 2
Robert pears 4
Susy oranges 12
Susy oranges 5
Terry oranges 9
```

6.5 Count Lines – nl

nl stands for number lines and it does just that.

nl [-options] [path]

```
user@ubuntu:~$ nl mysampledata.txt
1 Fred apples 20
2 Susy oranges 5
3 Mark watermellons 12
4 Robert pears 4
5 Terry oranges 9
6 Lisa peaches 7
7 Susy oranges 12
8 Mark grapes 39
9 Anne mangoes 7
10 Greg pineapples 3
11 Oliver rockmellons 2
12 Betty limes 14
```

The basic formatting is ok but sometimes you are after something a little different. With a few command line options, nl is happy to oblige.

```
user@ubuntu:~$ nl -s '. ' -w 10 mysampledata.txt
         1. Fred apples 20
         2. Susy oranges 5
         3. Mark watermellons 12
         4. Robert pears 4
         5. Terry oranges 9
         6. Lisa peaches 7
         7. Susy oranges 12
```

```
 8. Mark grapes 39
 9. Anne mangoes 7
10. Greg pineapples 3
11. Oliver rockmellons 2
12. Betty limes 14
```

In the above example we have used 2 command line options. The first one -s specifies what should be printed after the number while the second one -w specifies how much padding to put before the numbers. For the first one we needed to include a space as part of what was printed. Because spaces are normally used as separator characters on the command line we needed a way of specifying that the space was part of our argument and not just inbetween arguments. We did that by including the argument surrounded by quotes.

6.6 Count Words – wc

wc stands for word count and it does just that (as well as characters and lines. By default it will give a count of all 3 but using command line options we may limit it to just what we are after.

`wc [-options] [path]`

```
user@ubuntu:~$ wc mysampledata.txt
12 36 195 mysampledata.txt
```

Sometimes you just want one of these values. -l will give us lines only, -w will give us words and -m will give us characters. The example below gives us just a line count.

```
user@ubuntu:~$ wc -l mysampledata.txt
12 mysampledata.txt
```

You may combine the command line arguments too. This example gives us both lines and words.

```
user@ubuntu:~$ wc -lw mysampledata.txt
12 36 mysampledata.txt
```

6.7 Search and Replace – sed

sed stands for Stream Editor and it effectively allows us to do a search and replace on our data. It is quite a powerful command but we will use it here in it's basic format.

`sed <expression> [path]`

A basic expression is of the following format:

`s/search/replace/g`

The initial s stands for substitute and specifies the action to perform (there are others but for now we'll keep it simple). Then between the first and second slashes (/) we place what it is we are searching for. Then between the second and third slashes, what it is we wish to replace it with. The g at the end stands for global and is optional. If we omit it then it will only replace the first instance of search on each line. With the g option we will replace every instance of search that is on each line. Let's see an example. Say we ran out of oranges and wanted to instead give those people bananas.

```
user@ubuntu:~$ sed 's/oranges/bananas/g' mysampledata.txt
Fred apples 20
Susy bananas 5
Mark watermellons 12
Robert pears 4
Terry bananas 9
Lisa peaches 7
Susy bananas 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
```

It's important to note that sed does not identify words but strings of characters. Try running the example above yourself but replacing oranges with es and you'll see what I mean. The search term is also actually something called a regular expression which is a means to define a pattern (similar to wildcards we looked at in section 7). We'll learn more about regular expressions in the next section and you can use them to make sed even more powerful.

Also note that we included our expression within single quotes. We did this so that any characters included in it which may have a special meaning on the command line don't get interpreted and acted upon by the command line but instead get passed through to sed.

A common mistake is to forget the single quotes in which case you may get some strange behaviour from the command line. If this happens you may need to press CTRL+c to cancel the program and get back to the prompt.

7 REDIRECTION

In the previous sections we looked at a collection of filters that would manipulate data for us. In this section we will see how we may join them together to do more powerful data manipulation. Even though the mechanisms and their use are quite simple, it is important to understand various characteristics about their behaviour if you wish to use them effectively.

7.1 Data Streams

Every program we run on the command line automatically has three data streams connected to it (see Fig. 1).

- STDIN (0) - Standard input (data fed into the program)
- STDOUT (1) - Standard output (data printed by the program, defaults to the terminal)
- STDERR (2) - Standard error (for error messages, also defaults to the terminal)

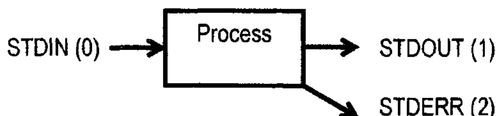

Figure 1 – Standard data streams

Piping and redirection is the means by which we may connect these streams between programs and files to direct data in interesting and useful ways.

We'll demonstrate piping and redirection below with several examples but these mechanisms will work with every program on the command line, not just the ones we have used in the examples.

7.2 Redirecting to a File

Normally, we will get our output on the screen, which is convenient most of the time, but sometimes we may wish to save it into a file to keep as a record, feed into another system, or send to someone else. The greater than operator (>) indicates to the command line that we wish the programs output (or whatever it sends to STDOUT) to be saved in a file instead of printed to the screen. Let's see an example.

```
user@ubuntu:~$ ls
barry.txt bob example.png firstfile foo1 video.mpeg
user@ubuntu:~$ ls > myoutput
user@ubuntu:~$ ls
barry.txt bob example.png firstfile foo1 myoutput video.mpeg
user@ubuntu:~$ cat myoutput
barry.txt
bob
example.png
firstfile
foo1
myoutput
video.mpeg
```

Let's break it down:

Line 1 Let's start off by seeing what's in our current directory.

Line 3 Now we'll run the same command but this time we use the > to tell the terminal to save the output into the file myoutput. You'll notice that we don't need to create the file before saving to it. The terminal will create it automatically if it does not exist.

Line 4 As you can see, our new file has been created.

Line 6 Let's have a look at what was saved in there.

You'll notice that in the above example, the output saved in the file was one file per line instead of all across one line when printed to the screen. The reason for this is that the screen is a known width and the program can format its output to suit that. When we are redirecting, it may be to a file, or it could be somewhere else, so the safest option is to format it as one entry per line. This also allows us to easier manipulate that data later on as we'll see further down the page.

When piping and redirecting, the actual data will always be the same, but the formatting of that data may be slightly different to what is normally printed to the screen. Keep this in mind.

You'll also notice that the file we created to save the data into is also in our listing. The way the mechanism works, the file is created first (if it does not exist already) and then the program is run and output saved into the file.

7.3 Saving to an Existing File

If we redirect to a file which does not exist, it will be created automatically for us. If we save into a file which already exists, however, then it's contents will be cleared, then the new output saved to it.

```
user@ubuntu:~$ cat myoutput
barry.txt
bob
example.png
firstfile
foo1
myoutput
video.mpeg
user@ubuntu:~$ wc -l barry.txt > myoutput
user@ubuntu:~$ cat myoutput
7 barry.txt
```

We can instead get the new data to be appended to the file by using the double greater than operator (>>).

```
user@ubuntu:~$ cat myoutput
7 barry.txt
user@ubuntu:~$ ls >> myoutput
user@ubuntu:~$ cat myoutput
7 barry.txt
barry.txt
bob
example.png
firstfile
foo1
myoutput
video.mpeg
```

7.4 Redirecting from a File

If we use the less than operator (<) then we can send data the other way. We will read data from the file and feed it into the program via it's STDIN stream.

```
user@ubuntu:~$ wc -l myoutput
8 myoutput
user@ubuntu:~$ wc -l < myoutput
8
```

A lot of programs (as we've seen in previous sections) allow us to supply a file as a command line argument and it will read and process the contents of that file. Given this, you may be asking why we would need to use this operator. The above example illustrates a subtle but useful difference. You'll notice that when we ran wc supplying the file to process as a command line argument, the output from the program included the name of the file that was processed. When we ran it redirecting the contents of the file into wc the file name was not printed. This is because whenever we use redirection or piping, the data is sent anonymously. So in the above example, wc recieved some content to process, but it has no knowledge of where it came from so it may not print this information. As a result, this mechanism is often used in order to get ancilliary data (which may not be required) to not be printed.

We may easily combine the two forms of redirection we have seen so far into a single command as seen in the example below.

```
user@ubuntu:~$ wc -l < barry.txt > myoutput
user@ubuntu:~$ cat myoutput
7
```

7.5 Redirecting STDERR

Now let's look at the third stream which is Standard Error or STDERR. The three streams actually have numbers associated with them (in brackets in the list at the top of the page). STDERR is stream number 2 and we may use these numbers to identify the streams. If we place a number before the > operator then it will redirect that stream (if we don't use a number, like we have been doing so far, then it defaults to stream 1).

```
user@ubuntu:~$ ls -l video.mpg blah.foo
ls: cannot access blah.foo: No such file or directory
-rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg
user@ubuntu:~$ ls -l video.mpg blah.foo 2> errors.txt
-rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg
user@ubuntu:~$ cat errors.txt
ls: cannot access blah.foo: No such file or directory
```

Maybe we wish to save both normal output and error messages into a single file. This can be done by redirecting the STDERR stream to the STDOUT stream and redirecting STDOUT to a file. We redirect to a file first then redirect the error stream. We identify the redirection to a stream by placing an & in front of the stream number (otherwise it would redirect to a file called 1).

```
user@ubuntu:~$ ls -l video.mpg blah.foo > myoutput 2>&1
user@ubuntu:~$ cat myoutput
ls: cannot access blah.foo: No such file or directory
-rwxr--r-- 1 ryan users 6 May 16 09:14 video.mpg
```

7.6 Piping

So far we've dealt with sending data to and from files. Now we'll take a look at a mechanism for sending data from one program to another. It's called piping and the operator we use is (|) (found above the backslash (\ ) key on most keyboards). What this operator does is feed the output from the program on the left as input to the program on the right. In the example below we will list only the first 3 files in the directory.

```
user@ubuntu:~$ ls
barry.txt bob example.png firstfile foo1 myoutput video.mpeg
user@ubuntu:~$ ls | head -3
barry.txt
bob
example.png
```

We may pipe as many programs together as we like. In the below example we have then piped the output to tail so as to get only the third file.

```
user@ubuntu:~$ ls | head -3 | tail -1
example.png
```

Any command line arguments we supply for a program must be next to that program.

I often find people try and write their pipes all out in one go and make a mistake somewhere along the line. They then think it is in one point but in fact it is another point. They waste a lot of time trying to fix a problem that is not there while not seeing the problem that is there. If you build your pipes up incrementally then you won't fall into this trap. Run the first program and make sure it provides the output you were expecting. Then add the second program and check again before adding the third and so on. This will save you a lot of frustration.

You may combine pipes and redirection too.

```
user@ubuntu:~$ ls | head -3 | tail -1 > myoutput
user@ubuntu:~$ cat myoutput
example.png
```

Below are some more examples to give an idea of the sorts of things you can do with piping. There are many things you can achieve with piping and these are just a few of them. With experience and a little creative thinking I'm sure you'll find many more ways to use piping to make your life easier.

All the programs used in the examples are programs we have seen before. I have used some command line arguments that we haven't covered yet however. Look up the relevant man pages to find out what they do. Also you can try the commands yourself, building up incrementally to see exactly what each step is doing.

In this example we are sorting the listing of a directory so that all the directories are listed first.

```
user@ubuntu:~$ ls -l /etc | tail -n +2 | sort
drwxrwxr-x 3 nagios nagcmd 4096 Mar 29 08:52 nagios
drwxr-x--- 2 news news 4096 Jan 27 02:22 news
drwxr-x--- 2 root mysql 4096 Mar 6 22:39 mysql
...
```

In this example we will feed the output of a program into the program less so that we can view it easier.

```
user@ubuntu:~$ ls -l /etc | less
```

(Full screen of output you may scroll. Try it yourself to see.)

Identify all files in your home directory which the group has write permission for.

```
user@ubuntu:~$ ls -l ~ | grep '^.....w'
drwxrwxr-x 3 ryan users 4096 Jan 21 04:12 dropbox
```

Create a listing of every user which owns a file in a given directory as well as how many files and directories they own.

```
user@ubuntu:~$ ls -l /projects/ghosttrail | tail -n +2 | sed 's/\s\s*/ /g' | cut -d ' ' -f 3 | sort | uniq -c
8 anne
34 harry
37 tina
18 ryan
```

8 WILDCARDS

Wildcards are a set of building blocks that allow you to create a pattern defining a set of files or directories. As you would remember, whenever we refer to a file or directory on the command line we are actually referring to a path. Whenever we refer to a path we may also use wildcards in that path to turn it into a set of files or directories.

8.1 Basics of Wildcards

Here is the basic set of wildcards:

```
* - represents zero or more characters
? - represents a single character
[] - represents a range of characters
```

As a basic first example we will introduce the *. In the example below we will list every entry beginning with a b.

```
user@ubuntu:~$ pwd
/home/user
user@ubuntu:~$ ls
barry.txt blah.txt bob example.png firstfile foo1 foo2
foo3 frog.png secondfile thirdfile video.mpeg
user@ubuntu:~$ ls b*
barry.txt blah.txt bob
```

The mechanism here is actually kinda interesting. On first glance you may assume that the command above (ls) receives the argument b* then proceeds to translate that into the required matches. It is actually bash (The program that provides the command line interface) that does the translation for us. When we offer it this command it sees that we have used wildcards and so, before running the command (in this case ls) it replaces the pattern with every file or directory (ie path) that matches that pattern. We issue the command:

```
user@ubuntu:~$ ls b*
```

Then the system translates this into:

```
user@ubuntu:~$ ls barry.txt blah.txt bob
```

and then executes the program. The program never sees the wildcards and has no idea that we used them. This is funky as it means we can use them on the command line whenever we want. We are not limited to only certain programs or situations.

For all the examples below, assume we are in the directory ***/home/user*** and that it contains the files as listed above. Also note that I'm using ls in these examples simply because it is a convenient way to illustrate their usage. Wildcards may be used with any command.

Every file with an extension of txt at the end. In this example we have used an absolute path. Wildcards work just the same if the path is absolute or relative.

```
user@ubuntu:~$ ls /home/user/*.txt
barry.txt blah.txt
```

Now let's introduce the **?** operator. In this example we are looking for each file whose second letter is i. As you can see, the pattern can be built up using several wildcards.

```
user@ubuntu:~$ ls ?i*
firstfile video.mpeg
```

Or how about every file with a three letter extension. Note that video.mpeg is not matched as the path name must match the given pattern exactly.

```
user@ubuntu:~$  ls *.???
barry.txt blah.txt example.png frog.png
```

And finally the range operator (**[ ]**). Unlike the previous 2 wildcards which specified any character, the range operator allows you to limit to a subset of characters. In this example we are looking for every file whose name either begins with a s or v.

```
user@ubuntu:~$  ls [sv]*
secondfile video.mpeg
```

With ranges we may also include a set by using a hyphen. So for example if we wanted to find every file whose name includes a digit in it we could do the following:

```
user@ubuntu:~$ ls *[0-9]*
foo1 foo2 foo3
```

We may also reverse a range using the caret (**^**) which means look for any character which is not one of the following.

```
user@ubuntu:~$ ls [^a-k]*
secondfile thirdfile video.mpeg
```

8.2 Use of Wildcards

The examples above illustrate how the wildcards work but you may be wondering what use they actually are. People use them everywhere and as you progress I'm sure you'll find many ways in which you can use them to make your life easier. Here are a few examples to give you a taste of what is possible. Remember, these are just a small sample of what is possible, and they can be used whenever you specify a path on the command line. With a little creative thinking you'll find they can be used in all manner of situations.

Find the file type of every file in a directory.

```
user@ubuntu:~$ file /home/ryan/*
bin: directory
Documents: directory
frog.png: PNG image data
public_html: directory
```

Move all files of type either jpg or png (image files) into another directory.

```
user@ubuntu:~$ mv public_html/*.??g public_html/images/
```

Find out the size and modification time of the .bash_history file in every users home directory. (.bash_history is a file in a typical users home directory that keeps a history of commands the user has entered on the command line. Remember how the . means it is a hidden file?) As you can see in this example, we may use wildcards at any point in the path.

```
user@ubuntu:~$ ls -lh /home/*/.bash_history
-rw-------    1    harry    users    2.7K    Jan    4    07:32
/home/harry/.bash_history
-rw------- 1 ryan users 3.1K Jun 12 21:16 /home/ryan/.bash_history
```

9 REGULAR EXPRESSIONS

Previously we looked at a collection of filters that would manipulate data for us. In this section we will look at another filter which is quite powerful when combined with a concept called regular expressions or re's for short. Re's can be a little hard to get your head around at first so don't worry if this stuff is a little confusing.

9.1 RE's and Grepping

Regular expressions are similar to the wildcards that we looked in previous section. They allow us to create a pattern. They are a bit more powerful however. Re's are typically used to identify and manipulate specific pieces of data. eg. we may wish to identify every line which contains an email address or a url in a set of data.

Re's are used all over the place. We will be demonstrating them here with **grep** but many other programs use them (including sed and vi which you learned about in previous sections) and many programming languages make use of them too.

The characters used in regular expressions are the same as those used in wildcards. Their behaviour is slightly different however. A common mistake is to forget this and get their functions mixed up.

egrep is a program which will search a given set of data and print every line which contains a given pattern. It is an extension of a program called grep. It's name is odd but based upon a command which did a similar function, in a text editor called ed. It has many command line options which modify it's behaviour so it's worth checking out it's man page. ie the -v option tells grep to instead print every line which does not match the pattern.

```
egrep [command line options] <pattern> [path]
```

In the examples below we will use a sample file **mysampledata.txt**:

```
user@ubuntu:~$ cat mysampledata.txt
Fred apples 20
Susy oranges 5
Mark watermellons 12
Robert pears 4
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Mark grapes 39
Anne mangoes 7
Greg pineapples 3
Oliver rockmellons 2
Betty limes 14
```

Let's say we wished to identify every line which contained the string mellon

```
user@ubuntu:~$ egrep 'mellon' mysampledata.txt
Mark watermellons 12
Oliver rockmellons 2
```

The basic behaviour of egrep is that is will print the entire line for every line which contains a string of characters matching the given pattern. This is important to note, we are not searching for a word but a string of characters.

Also note that we included the pattern within quotes. This is not always required but it is safer to get in the habit of always using them. They are required if your pattern contains characters which have a special meaning on the command line.

Sometimes we want to know not only which lines matched but their line number as well.

```
user@ubuntu:~$ egrep -n 'mellon' mysampledata.txt
3:Mark watermellons 12
11:Oliver rockmellons 2
```

Or maybe we are not interested in seeing the matched lines but wish to know how many lines did match.

```
user@ubuntu:~$ egrep -c 'mellon' mysampledata.txt
2
```

9.2 Learning Regular Expressions

The best way to learn regular expressions is to give the examples a try yourself, then modify them slightly to test your understanding. It is common to make mistakes in your patterns while you are learning. When this happens typically every line will be matched or no lines will

be matched or some obscure set. Don't worry if this happens you haven't done any damage and you can easily go back and have another go. Remember you may hit the up arrow on your keyboard to get at your recent commands and also modify them so you don't need to retype the whole command each time.

If you're not getting the output you would like then here are some basic strategies.

First off, check for typo's. If you're like me then you're prone to making them.

Re read the content here. Maybe what you thought a particular operator did was slightly different to what it actually does and re reading you will notice a point you may have missed the first time.

Break your pattern down into individual components and test each of these individually. This will help you to get a feel for which parts of the pattern is right and which parts you need to adjust.

Examine your output. Your current pattern may not have worked the way you want but we can still learn from it. Looking at what we actually did match and using it to help understand what actually did happen will help us to work out what we should try changing to get closer to what we actually want.

Debuggex is an on-line tool that allows you to experiment with regular expressions and allows you to visualise their behaviour. It can be a good way to better understand how they work.

9.3 Regular Expression Syntax

I will outline the basic building blocks of re's below then follow on with a set of examples to demonstrate their usage.

. (dot) - a single character.
? - the preceding character matches 0 or 1 times only.
***** - the preceding character matches 0 or more times.
+ - the preceding character matches 1 or more times.
{n} - the preceding character matches exactly n times.
{n,m} - the preceding character matches at least n times and not more than m times.
[agd] - the character is one of those included within the square brackets.
[^agd] - the character is not one of those included within the square brackets.
[c-f] - the dash within the square brackets operates as a range. In this case it means either the letters c, d, e or f.
() - allows us to group several characters to behave as one.
| (pipe symbol) - the logical OR operation.
^ - matches the beginning of the line.
$ - matches the end of the line.

We'll start with something simple. Let's say we wish to identify any line with two or more vowels in a row. In the example below the multiplier {2,} applies to the preceding item which is the range.

```
user@ubuntu:~$ egrep '[aeiou]{2,}' mysampledata.txt
Robert pears 4
Lisa peaches 7
Anne mangoes 7
Greg pineapples 3
```

How about any line with a 2 on it which is not the end of the line. In this example the multiplier + applies to the . which is any character.

```
user@ubuntu:~$ egrep '2.+' mysampledata.txt
Fred apples 20
```

The number 2 as the last character on the line.

```
user@ubuntu:~$ egrep '2user@ubuntu:~$' mysampledata.txt
Mark watermellons 12
Susy oranges 12
Oliver rockmellons 2
```

And now each line which contains either 'is' or 'go' or 'or'.

```
user@ubuntu:~$ egrep 'or|is|go' mysampledata.txt
Susy oranges 5
Terry oranges 9
Lisa peaches 7
Susy oranges 12
Anne mangoes 7
```

Maybe we wish to see orders for everyone who's name begins with A - K.

```
user@ubuntu:~$ egrep '^[A-K]' mysampledata.txt
Fred apples 20
Anne mangoes 7
Greg pineapples 3
Betty limes 14
```

APPENDIX A. LABORATORY WORK №1

1. Detect the pathname of current working directory. Which type of path have you received? **2.1**
2. Create the following directory structure inside your home dir (see Fig. 2): **2.2, 3.1**

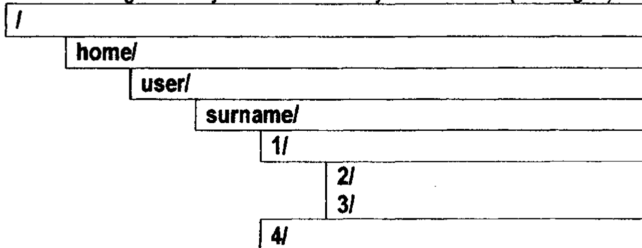

Figure 2 – Directory structure

3. Copy file **/etc/group** to dir **1** using absolute names of file and target directory. **2.2, 3.4**
4. Copy file **/etc/group** to dir **2** using absolute name of file and relative name of target directory. **2.2, 3.4**
5. Copy file **/etc/group** to dir **3** using relative names of file and target directory. **2.2, 3.4**
6. Copy file **/etc/group** to dir **4** using absolute name of file and relative name of target directory with ~. **2.2, 3.4**

7. Enter the dir **3** using one command.
8. Remove the file **group** from directory **4** using one command. — **2.2, 2.3**
9. Come to your home dir. Remove directories **1** and **4**. — **2.2, 3.7**
10. Have a look on manual pages on commands **apropos, head, tail, ls, sort, cat, more**. You will be asked to use this commands during the report defense. — **2.3, 3.8** **1.2**

APPENDIX B. LABORATORY WORK №2

1. Create hard link to any file. View the content of the file using link. Remove the source file. Try to view it's content using the link. Explain the result. — **4.1, 1.1, 3.7**
2. Try to create hard link to directory. Explain the result. — **4.1**
3. Create symbolic link to any file. View the content of the file using link. Remove the source file. Try to view it's content using the link. Explain the result. — **4.2, 1.1, 3.7**
4. Try to create symbolic link to directory. Explain the result. — **4.2**
5. Create hard and symbolic links to the same file. View the inodes of the file and both links. Explain the result. — **4.1, 4.2, 4.3**
6. Get the permissions of the files **/etc/shadow** and **/etc/passwd** and try to view the contents of this files. Yes, of course, explain the result. — **5.1, 1.1**
7. Create the files in your home dir using octal digit masks for permissions: — **3.3, 5.3**
   a. All for you, None for group and others;
   b. Read+Write for you, Read for group, All for others;
   c. Write+Execute for you, None for group, Read for others.
8. Perform previous task using symbolic permission attributes. — **5.2**
9. Convert the number of your student's record-book to the octal system, divide it to the groups of 2-3 digits and create the files with such permissions. Explain the operations permitted on this files to different system users (you, belonging to your group, all other users). — **3.3, 5.3**
10. Try to enter the **/root** dir. Explain the result. — **2.3, 5.1**

APPENDIX C. LABORATORY WORK №3

1. Use **echo** to output the text message to a new file using > redirection. — **1.1, 7.2**
2. Use **echo** to output the text message to a new file using >> redirection. — **1.1, 7.2, 7.3**
3. Use **echo** to output the text message to the existing file using > redirection. — **1.1, 7.3**
4. Use **echo** to output the text message to the existing file using >> redirection. — **1.1, 7.3**
5. Use **cat** to output the content of **/etc/passwd** to a new file **~/mypasswd** — **1.1, 7.2**
6. Create the file **myscript** containing with the following scrypt: — **1.1, 7.2**

```
#!/bin/sh
echo "it is stdout"
echo "it is stderr">&2
exit 0
```

7. Run it **(sh myscript)**
   a. without redirection;
   b. redirecting STDERR to file **errmyscr**;
   c. redirecticting STDOUT to **outmyscr**.

8. Explain the results of the execution in previous task. **7.5**
9. Output the third string from the end of file **/etc/passwd**. **7.3**
10. Output five strings from the beginning of file **/etc/passwd** sorted in reverse order. **7.6, 6.3** **7.6, 6.2, 6.4**

APPENDIX D. LABORATORY WORK №4

1. Create empty files **1a.txt, 2b.jpg, 3c.gif** and **4d.txt** in your current working directory. List all files in current folder with the filenames starting by number. **3.3, 8.1**
2. Move all **.txt** files from current folder to **/home/user/YourName** **3.5, 8.1**
3. Copy all image files from current folder to **/home/user** **3.4, 8.1**
4. Output all strings of **/etc/passwd** containing **x:number_less_than_4** **9.1, 9.2**
5. Output all strings of **/etc/passwd** corresponding to users who have **/home** or **/root** in the home folder path. **9.1, 9.2**
6. Save to a new file strings from **/etc/group** corresponding to groups starting with **d** **9.1. 9.2, 7.2**
7. Use **ls -l** to list all device files from **/dev**. Create pipeline using **grep** to list only block devices. **7.6, 9.1, 9.2**
8. Save to a new file five strings from **/etc/passwd** containing **latin_letters:x:number_without_0_and_3**. **9.1. 9.2, 7.2**
9. List all subdirectories of **/etc** allowing for every user to enter. **7.6, 9.2**
10. Count the number of strings longer than 30 symbols in **/etc/passwd**. **7.6, 9.2, 6.5**

APPENDIX E. SHORT HOW-TO'S

How do I...

...execute the command **command** without options on the file **file1**?	**command file1**
...execute the command **command** with options **-p** and **-r** on the files **file2** and **file3**?	**command -p -r file2 file3**
...create empty file **myfile**?	**touch myfile**
...print message **Hello**?	**echo Hello**
...list files in current directory?	**ls**
...list files in parent directory?	**ls ..**
...create new directory **mydir**?	**mkdir mydir**
...navigate to directory **path/to/mydir**?	**cd path/to/mydir**
...copy file **myfile** to directory **path/to/dir**?	**cp myfile path/to/dir**
...remove file **myfile**?	**rm myfile**
...remove directory **mydir**?	**rm -r mydir**
...view manual on the command **ls**?	**man ls**
...create hard link **hlink** to file **myfile**?	**ln myfile hlink**
...create symbolic link **slink** to file **myfile**?	**ln -s myfile slink**
...view the inodes of files in current directory?	**ls -i**
...view the permissions and other attributes of files in current directory?	**ls -l**

...change permissions of file **myfile** to read, write, execute by everybode? **chmod 777 myfile**
...change owner of file **myfile** to user **jackson**? **chown jackson myfile**
...print message **Hello** to the file **myfile**? **echo Hello >myfile**
...append message **world** to the file **myfile**? **echo world >>myfile**
...execute shell script **myscript**? **sh myscript**
...run command **command** redirecting STDOUT to STDERR? **command 1>&2**
...run command **command** redirecting STDERR to STDOUT? **command 2>&1**
...run command **command** redirecting STDERR to **myfile**? **command 2>myfile**
...run command **command** redirecting STDIN from **myfile**? **command <myfile**
...use STDOUT of command **command1** as STDIN of command **command2**? **command1 | command2**
...list the files of current directory and sort them in reverse order? **ls | sort -r**
...remove files starting with **abc**? **rm abc***
...copy files **file1**, **file2**, **file3** ... **file9** to directory **mydir**? **cp file[1-9] mydir**
...show the strings of file **myfile** starting with number? **egrep '^[0-9]' myfile**
...show the strings of file **myfile** having **home/user** inside? **egrep 'home/user' myfile**
...list only readable files form current directory? **ls -l | egrep '^?r??r??r'**

APPENDIX F. THE REPORT

You have to write/print and submit the report on every laboratory work in a standard form:

- paper size A4;
- the title page contains your name, group, discipline name, number of the work, it's topic (see Fig. 3);
- the report contains "The Goal" – what do you think you had to study during the work; "Workflow" – what was you asked to do, what have you done, which results have you got; "The Conclusion" – your conclusion on what have you learned during this work (see Fig. 4).

MINISTRY OF EDUCATION OF THE REPUBLIC OF BELARUS
EDUCATIONAL INSTITUTION «BREST STATE TECHNICAL UNIVERSITY»
DEPARTMENT OF INTELLIGENT INFORMATION TECHNOLOGY

REPORT
Laboratory Work «#»
«TOPIC»

Performed by student of **«Group» «Student Name»**
Checked by **«Professor Name»**

BREST 2015

Figure 3 – Title page of the report

Goal: to learn the commands for file and directory manipulation.

Workflow:

1. Detect the pathname of current working directory. Which type of path have you received?

```
user@ubuntu:~$ pwd
/home/user
```

This is absolute path.

2. ...

Figure 4 – Sample page of the report

REFERENCES

1. Chadwick, R. Linux Tutorial [Electronic Resource]. – Mode of access: http://ryanstutorials.net/linuxtutorial/. – Date of access: 04.10.2015.
2. Shotts, W. The Linux Command Line: A Complete Introduction / W.Shotts, Jr. – A LinuxCommand.org Book, 2013. – 513 p.
3. Bates, M. Conquering the Command Line: Unix and Linux Commands for Developers [Electronic Resource]. – Mode of access: http://conqueringthecommandline.com/book. – Date of access: 04.10.2015.
4. Garrels, M. Bash Guide for Beginners [Electronic Resource]. – Mode of access: http://www.tldp.org/LDP/Bash-Beginners-Guide/html/. – Date of access: 04.10.2015.

УЧЕБНОЕ ИЗДАНИЕ

Составители:
Кочурко Павел Анатольевич
Кочурко Вячеслав Анатольевич

ПОСОБИЕ

ДЛЯ ВЫПОЛНЕНИЯ ЛАБОРАТОРНЫХ РАБОТ ПО КУРСУ

«Системное программное обеспечение»

для студентов дневной и заочной формы обучения специальности 1–53 01 02 Автоматизированные системы обработки информации

ЧАСТЬ I

«Интерфейс командной строки Linux»
(на английском языке)

Ответственный за выпуск: Кочурко П.А.
Редактор: Боровикова Е.А.
Компьютерная верстка: Боровикова Е.А.

Подписано к печати 09.10.2015 г. Бумага «Снегурочка». Формат 60x84 1/16.
Гарнитура Arial Narrow. Усл. печ. л. 2,55. Уч. изд. л. 2,75.
Заказ № 1033. Тираж 35 экз. Отпечатано на ризографе Учреждения образования «Брестский государственный технический университет»
224017, г. Брест, ул. Московская, 267.