# Contents

# 1 Introduction

## 1.1 Introduction of the Course

The goal of the present course is to introduce a command line interface of a Linux system to users who intend to work on a super computing cluster but have not worked with the command line or have a little experience. The course provides a minimum knowledge needed for performing such daily tasks as

- navigation through the file system;
- organization of personal data;
- manipulation with files and directories;
- running applications and programs;
- starting computations on the cluster.

This also includes knowledge of shell scripting needed for understanding shell scripts and for writing own scripts. The course is divided into two parts: the basics of Linux shell is introduced in the first part while shell scripting is discussed in the second part.

For the course it is not relevant what kind of work the user does on a cluster. It can be development of code with using compilers or performing simulations with applying pre-installed software. In all cases, the user needs to perform basic operations listed above.

Throughout the present manuscript used as an additional material to the course (handout), most of the basic operations are demonstrated by exercises. The exercises are organized in an order, and results of previous exercises are used in further exercises.

## 1.2 Introduction of Linux Shell

Most of super computing clusters are controlled by operating systems of a Linux-family (a Linux system or just Linux). On such clusters, a usual graphical user interface (GUI) is not available, and the communication with the operating system is done by means of the command line interface (CLI). The CLI does not have menus and buttons[1], and the use of the mouse is very limited (see appendix 18). The user has to input exact commands together with required options, file names and so on.

**Shell** is a program which realizes the command line interface in a Linux system. There are many implementations of shell, and all of them have a similar functionality but can differ in syntax of commands. In this course the main focus is on the **bash** shell, and all the information provided in the handout is related to this type of shell.

# 2 First Look

## 2.1 Start and Ending

After logging in to the cluster the shell is run, and the user is invited by a **shell prompt**:

```
username@computername:~>
```

[1] The auxiliary tool called Midnight Commander is a text-based file manager, which can assist in working with files and directories, and has many additional options.

It consists of the user name and the computer name separated by the '`@`' character and can be slightly adjusted. When the shell prompt appears, the user can input commands and execute them by pressing `Enter`.

Due to the variety of shell programs, it is important to know the current shell program. The easiest way to check it (without going into details) is to apply the command

```
echo $SHELL
```

If the output of the command is similar to

```
/bin/bash
```

one can be sure that the bash shell is run.

The shell can be closed by pressing `Ctrl+d` or by the command

```
exit
```

This will close the program and brings back to the system where the login has been done.

**Remark**: It is recommended to close the shell explicitly by one of the above methods due to security reasons (especially if it is on a cluster).

## 2.2 Getting Help

Almost all commands of the bash shell are described thoroughly on so-called **manual pages** (or **man pages**) run by the `man` command. For example, the command

```
man COMMAND
```

opens the man page on the `COMMAND` command. The `man` command has also the man page:

```
man man
```

When the man page is opened, important hints are written at the highlighted bottom line. The user can navigate through the man page by using the following keys:

a) the '`q`' character to quit the manual;

b) the '`h`' character to get help on the manual;

c) the `Down`/`Up` arrows to scroll down/up (one line);

d) the `PageDown`/`PageUp` keys to scroll down/up (one page);

e) the '`/`' character to enter a keyword for searching following by the `Enter` key;

f) while searching, press the '`n`' character to move to the next found element (if any).

Many commands are described in several sections listed after executing the `man` command. A particular section of the list can be opened by typing its name and pressing `Enter`. A default section is opened just by pressing `Enter` (without entering).

# 3 File System

## 3.1 Hierarchy of Files

Any user in a system has personal data for working (e.g. source files, programs, input data, results, and so on). In Linux all information is stored in a **file** and each file is located in some **directory**. Any directory can contain many files and other directories, which are called **subdirectories** for that particular directory, and that directory is called a **parent directory** for those subdirectories. All directories and files in the operating system are organized in a **file system**. Some files and directories are created by the operating system. Other files and directories are created by users during their work. To keep the file system with its complex structure clean, all files and directories have access rights discussed in section 15.

The file system has a start point which is a special unique directory called a **root directory** and denoted by the '/' character. Roughly speaking all files and directories which exist in the operating system are located in that root directory. Every user of the operating system has a so-called **user home directory** for storing personal files and directories. The names of user home directories are the same as the user names (they must be also unique). Traditionally all user home directories are located in the directory called `home`, which in turn is located in the root directory. In figure 1 the example of a simplified file system, which "grows" from the root directory to the left, is shown. In the figure the directory called `home` contains the user home directories, `user1`, `user2`, `user3`. The user home directory `user2`, which belongs to the user with the `user2` name, contains the files `MyData1`, `MyData2`, `MyData3`.

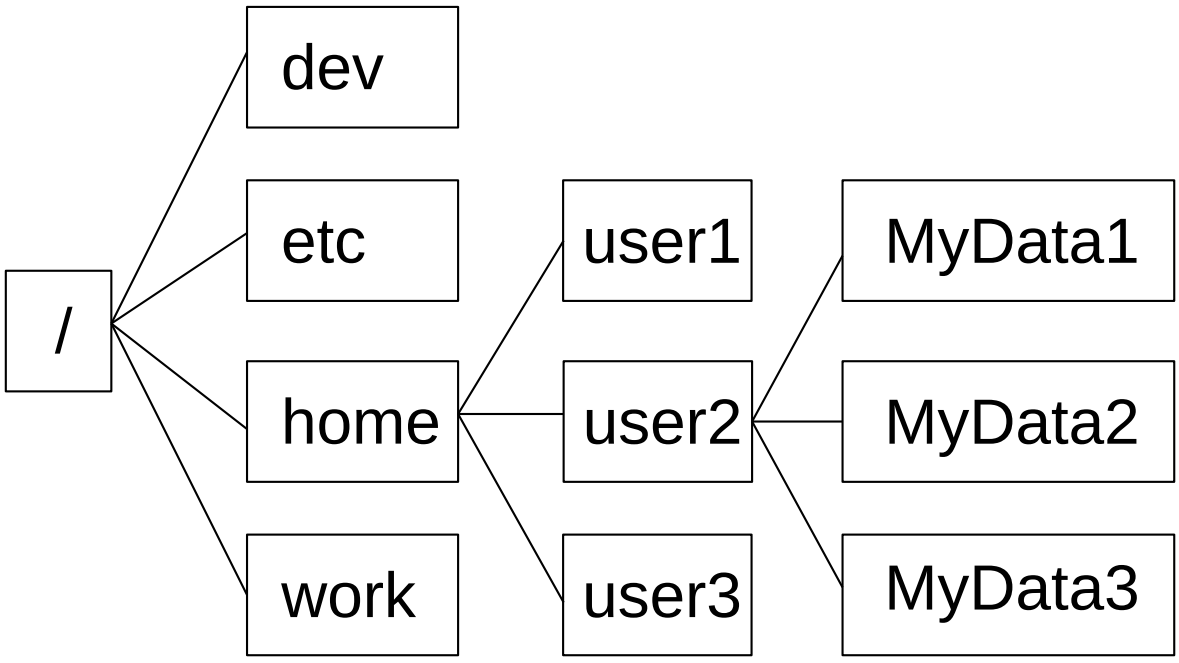


Figure 1: The sketch of the simplified structure of the file system in Linux.

Each file (and directory) has the name and is determined by an **absolute path** or just path[2]. This path is formed from the names of the directories needed to achieve the particular file (or directory) starting from the root directory. In other words, the absolute path to a file (or directory) is its location in the file system. The names of the directories included in the path are divided by the '/' character (slash). A **full name** of a file (or directory) consists of the absolute

[2] The alternative to the absolute path is the relative path introduced in section 6.

path and the file name appended at the end of the path. In figure 1 the file called `MyData1` has the full name

```
/home/user2/MaData1
```

The above line tells that the `MaData1` file is located in the `user2` directory, which is located in the directory called `home`, and `home` is located in the root directory, `/` (the slash is placed at the very beginning). Similarly, the full name of the `user1` directory:

```
/home/user1
```

## 3.2 Remarks on files and directories

For the shell it is important to give a proper file name taking into account some rules discussed in the present section. If the name is not proper, the shell can treat it specifically, which results in an unexpected behaviour and possible problems. If the user does not follow the rules, a mistake can lead to loss of data. Thus, it is important to follow the rules (naming conventions).

1. Names of files and directories are case sensitive.

   For example, the files with the names "file" and "**F**ile" are two different files for the shell. Both files can coexist in the same directory.

2. Extensions of files are not used by the shell (they are simply parts of names).

   A file extension can be needed only by applications which is used to work with particular files or by users to quickly recognize file types.

3. Whitespace characters must not be used in names of files and directories.

   As an alternative, the '_' character can be used to separate parts of a name.

4. There cannot be two files or directories with the same name in the same directory.

   In most cases an old file is simply replaced by a new one with the same name **without prompting**. The same is relevant to directories.

The following characters can be applied in names of files and directories:

- the underscore '_', and the dot '.' (the latter must be applied with other characters);
- the small and capital letters of the English alphabet, 'A'-'Z', 'a'-'z';
- the ten numerical digits, '0'-'9'.

If the dot is put at the beginning of the name of a file or directory, it becomes **hidden**. Hidden files and directories are mostly applied for storing settings of the system and applications. In this case, the shell does not show hidden files and directories by default. It is not recommended to edit or remove such files and directories without knowing.

# 4 Basic Navigation

## 4.1 Introduction

One important daily task on the cluster is to organize personal data stored in files which are located in different directories. All files and directories of the file system are organized in a tree similar to the one in figure 1. Because the shell does not provide any file browser to show and navigate in the file system, there is a special mechanism for browsing used in the shell.

In any time the user has a **current working directory** (i.e. a current location in the file system). This is a special directory, because for many commands of the shell it is enough to specify only names of files and subdirectories of the current directory. If a file or directory is not located in the current working directory, the user has to specify the absolute path to it. The current directory can be changed, and the shell tracks it.

## 4.2 Exercises

For navigating, it is necessary to know the current working directory, and the files and subdirectories located in it (i.e. the content of the directory). The absolute path of the current working directory is shown by the command

```
pwd
```

The content of the current working directory can be shown by the command

```
ls
```

The `pwd` command is simple while `ls` has many options described on the man pages:

```
man ls
```

By specifying the path to another directory appended to the `ls` command, the content of that directory is shown. Thus, the user does not need to change the current directory to see the content of another directory. For example, the content of the root directory:

```
ls /
```

It can be more convenient to show the content in a long listing format by adding `-l`:

```
ls -l /
```

The output of the `ls` command with the `-l` option is a list which consists of several columns, and each line corresponds to a file or directory. The names of files and directories are given at the **very right** column (the output of `ls -l` is discussed in section 5).

In the output of the above command, one can find the directory called `home` (the subdirectory of the root directory). To check the content of `home`, the path starting from the root directory is specified:

```
ls -l /home
```

It can be seen that the `home` directory contains a lot of subdirectories and files making the output very long. To see the long list in a more convenient way, the program called `less` can be applied in combination with the `ls` command:

```
ls -l /home | less
```

The above command called a pipeline (see section 17) consists of two actions.

1. Generating the list of files and (sub-) directories of the `home` directory.

2. Redirecting the generated output to the `less` program for viewing.

Thus, the `less` program is run with using the output of `ls` and is used as a viewer, where the navigation is done in the same way as in the viewer of man pages (see section 2.2).

# 5 Attributes of files and directories

Files and directories have important attributes shown in the long listing format generated by the `ls` command with the `-l` option (see the exercises in section 4). The example of the output of the command `ls -l` is shown in figure 2, where each file (or directory) together with its attributes is given in one line. The file attributes are shown in the first six columns of the output while the seventh column contains the file names (see figure 2).

1) The first column consists of 10 characters.
   (a) The first character shows whether the object is a file or directory:
   - the '`-`' character for a (ordinary) file[3];
   - the '`d`' character for a directory.

   (b) The next 9 characters are related to access rights discussed in section 15.
2) Column 2 reveals the numbers of links to files and directories.
3) Column 3 shows the names of owners of files and directories.
4) Column 4 contains the names of groups which own files and directories.
5) Column 5 provides information on the file size[4].
6) Column 6 shows the time of creation or modification.

```
drwxr-xr-x 2 mo48xoxi group  512 13. Nov 13:47 Dir_tmp
-rw-r--r-- 1 mo48xoxi group  177 13. Nov 12:59 myfile.txt
-rw-r--r-- 1 mo48xoxi group 1387 13. Nov 13:11 README
-rwx------ 1 mo48xoxi group   15 13. Nov 12:59 script1
    1    2    3      4     5       6         7
```

Figure 2: The example of the output of the command `ls -l`. The column numbers are shown below and correspond to attributes. The names of files and directories are shown in column 7.

Files and directories have other attributes, but the above set is frequently used.

[3] In addition to a file and directory, in Linux there are other objects of different types shown by `ls`.
[4] If the object is not an ordinary file, column 5 can contain another information.

# 6 Special Directories and Relative Paths

The `ls` command does not show hidden objects by default, because in most cases it is better not to touch them. Nevertheless, it is still possible to list all files and directories including hidden ones by applying the `-a` option of the `ls` command:

```
ls -a
```

The same content can be shown in the long listing format by adding the `-l` option:

```
ls -l -a
```

If the output is too long, the `less` program introduced in section 4 can be also applied:

```
ls -l -a | less
```

The content of the `home` directory is shown similarly by adding the absolute path to it:

```
ls -l -a /home | less
```

Among other objects one can note two special directories, namely "`.`" and "`..`", which are present in all directories in Linux. Basically they are references to other directories:

- the "`.`" directory (dot) is a reference to the directory itself;
- the "`..`" directory (two dots) is a reference to the corresponding parent directory.

The role of these special directories is tremendous, because they allow to form **relative paths** to files and directories, which are alternatives to absolute paths (see section 3.1). While absolute paths always start from the root directory, relative paths can start from any directory. The main advantage of relative paths is that they allow to make commands shorter. Moreover, the use of relative paths makes commands universal and independent of the location of the directory with respect to which the relative paths are formed.

The easiest example with the reference directory "`.`" is demonstrated with `ls`:

```
ls -l -a .
```

This trivial example with the reference directory "`.`" can be alternatively done as

```
ls -l -a
```

It should not reduce the importance of this special directory applied in much more complicated cases. The use of the directory "`..`" with `ls` allows to see the parent directory:

```
ls -l -a ..
```

Because the special directories are present in all directories, one can use the command

```
ls -l -a ../..
```

The above command shows the content of the directory which is the parent directory of the parent directory of the current working directory. As it has been mentioned, the use of the relative paths allow us to avoid specifying the names of the corresponding directories.

# 7 Change and Create Directories

## 7.1 Introduction

Many commands of the shell are simpler, if they are applied in the current working directory. First of all, if only names of objects are specified with a command, the shell searches for the

objects in the current directory. Therefore, it is useful to change the current working directory to a new one and to do some work there.

The current working directory is changed to another one by the `cd` command:

```
cd  path/to/another_directory
```

If the directory for changing is located in the current working directory, it is enough to specify only its name without giving the path:

```
cd  another_directory
```

A new directory is created in another directory by the command

```
mkdir  path/to/another/directory/new_directory_name
```

Again, if the only name of a new directory is specified (without any path), it is created in the current working directory.

```
mkdir  new_directory_name
```

**Remark**: In the shell the user home directory is denoted by the '~' character (tilde). Thus, it is enough to use the tilde instead of the path to the user home directory.

## 7.2 Exercises

The next three commands are used to change to the `etc` directory located in the root directory and to show the current working directory before and after changing:

```
pwd
cd /etc
pwd
```

The user can easily change to its home directory by using the tilde with the `cd` command:

```
cd ~
pwd
```

After changing the `pwd` command shows the absolute path to the user home directory.

In the user home directory the new directories `testdir` and `mydir` will be created:

```
mkdir testdir
ls -l
mkdir mydir
ls -l
```

After creating each directory, it is checked by the `ls` command. The user can change the current directory to these directories by the `cd` command (one by one):

```
cd mydir
cd ../testdir
```

In the first case, only the name of the directory is given, because it is located in the current working directory (i.e. the user home directory). For the second command the `testdir` directory is given with the relative path, because changing is done from the `mydir` directory, which is the subdirectory of the user home directory, where `testdir` is located. Alternatively, one can use the absolute path to `testdir` instead of the relative one:

```
cd ~/testdir
```

# 8 Text Files

## 8.1 Introduction

Almost all files are created by applying different programs, and the shell is used to run such programs. As it is said in section 3.2, the shell does not use filename extensions, and they can be needed only by applications. Thus, one should be aware of programs applied for opening corresponding files. Nevertheless, the shell can recognize some well-known file formats by the `file` command (e.g. ASCII, jpeg, pdf, and so on):

```
file file_name
```

Text files of the ASCII format are very common in Linux, and several files are created in the frame of the course. The `nano` editor can be applied because of its simplicity, though there are a lot of other programs for processing text available in Linux. The editor is run similar to any command of the shell, and a file name can be specified:

```
nano text_file_name
```

If the specified file does not exist in the current working directory, it will be created. However, if the file with the same name exists, the editor will open it. If the user does not specify a file name in the command line, it can be done in the editor while saving text.

**Remark**: In the shell any program is run similar to the `nano` editor[5].

## 8.2 Exercises

The goal of the present section is to create a text file used in further examples, and the content of the file should be fixed. It should be noted that `nano` is a text-based editor and does not have any menu. Thus, all work is done by using shortcuts shown at the bottom part of the editor: one has to press `Ctrl` and a letter written near a corresponding action.

The text file called `myfile` is created in the `testdir` directory located in the user home directory (see section 7.2). This directory has to be the current working directory:

```
cd ~/testdir
```

After changing, the `nano` editor is run with the specified file name:

```
nano myfile
```

In the text editor `nano` the user can type text, for example:

```
Hello world!!!
```

After typing, the following actions are used to save the file and to close the editor:

(1) press `Ctrl+o` to save the text in the file;

(2) press `Enter` to confirm saving;

(3) press `Ctrl+x` to exit `nano`.

If a file with the same name exists in the same directory, the editor asks about overwriting.

After saving and exiting, one can check that the file has been created by `ls`:

```
ls -l
```

[5] Some applications can require additional arguments specified in the command line.

The file can be viewed by the `less` utility (see the navigation in section 2.2):

```
less myfile
```

In addition, there is a command for printing the content of a text file on the screen:

```
cat myfile
```

The `cat` command can print several files simultaneously and can be applied for concatenating files. Finally, the user can check the file type by the `file` command:

```
file myfile
```

# 9 Copy Files and Directories

## 9.1 Introduction

For copying files and directories, the `cp` command with slightly different options is applied. Similar to any other command, it is described on the man pages:

```
man cp
```

If a file is copied, a new identical file is created. In the case of a **directory**, a new directory is created and contains the same files and subdirectories as the original one. In all cases, the `cp` command requires at least two arguments: the source and the destination.

A **single file** is copied by applying the `cp` command as follows:

```
cp  original_file  file_copy
```

In the above command only the file names are specified. This means that the original file must be located in the current working directory and its copy is created in the same directory. Thus, both files must have different names. The same file can be copied into another directory, if the absolute or relative path to the new file is given:

```
cp  original_file  path/to/directory/file_copy
```

If the directory of the new file is given without name, the copy will have the same name as the original file. It is possible to copy a file located in a directory which is not the current working directory:

```
cp  path/to/directory/original_file  path/to/directory/file_copy
```

In this case, the user does not need to change the current working directory. Again, if the name of the copy is omitted, the new file will have the same name as the original one.

**Remark**: If a file with the same name exists in the directory where the copy is created, the shell will overwrite that file without asking. The shell will ask about overwriting, if the `-i` option is used with the `cp` command.

A **single directory** is copied by adding the `-r` option to the `cp` command:

```
cp  -r  original_directory  directory_copy
```

Similar to the case of files, copying directories without specifying paths is done within the current working directory. If the user needs to copy a directory into another directory, the absolute or relative path to the new directory must be specified:

```
cp  -r  original_directory  path/to/directory/directory_copy
```

If the name of the copy is omitted, it will have the same name as the original directory. Finally, if the directory to be copied is located in another directory, the path is needed:

```
cp  -r  path/to/directory/original_directory  path/to/directory/directory_copy
```

**Remark**: If a directory located in the directory for copying has the same name as the copy, the content of the original directory is appended to the old directory. In this case, the files with the same name will be overwritten without prompting. The shell will ask about overwriting, if the `-i` option is used with the `cp` command.

## 9.2 Exercises

In the previous examples the two directories, `mydir` and `testdir`, are created in the user home directory:

```
cd ~
ls -l
```

The `mydir` directory should be empty, and `testdir` contains the `myfile` file (see section 8.2). The directories are checked easily by using the `ls` command:

```
ls -l mydir
ls -l testdir
```

The first step is to copy the `myfile` file into `myfile2` inside the `testdir` directory:

```
cd testdir
cp myfile myfile2
```

The `ls` command shows, if the new file `myfile2` has been created:

```
ls -l
```

The `myfile3` file is created in the `mydir` directory by copying `myfile` from `testdir`:

```
cp myfile ~/mydir/myfile3
```

In this case, the original file can be specified without path, because it is located in the current working directory (i.e. `testdir`). However, the name of the file copy must be given with its path (i.e. `~/mydir`). The copy can be checked by the `ls` command without changing the current working directory:

```
ls ~/mydir/
ls ../mydir/
```

In the first case, the absolute path is used, while the relative path is specified in the second variant. Note, the copies are done without changing the working directory `testdir`.

The next example demonstrates copying of the file from `mydir` into `testdir` without renaming, which means that the name of the copy is not needed. The `mydir` directory should have the `myfile3` file to be copied, and `testdir` contains `myfile` and `myfile2`. Because `testdir` is the current working directory, the path to the copy is not needed, but the path to the original file, `myfile3`, is needed. The possible but wrong variant can be

```
cp ~/mydir/myfile3
```

The problem is that the `cp` command requires at least two arguments: the source and the destination. There are a lot of opportunities to correct the command, but the simplest variant is to use the reference to the current directory, "`.`" (see section 6):

```
cp ~/mydir/myfile3 .
```

The copy of `myfile3` must have the same name, and the new content of `testdir` is shown:

```
ls -l
```

The example of copying directories is done in the user home directory:

```
cd ~
ls -l
```

The `ls` command should show the two directories, `mydir` and `testdir`. The `mydir` directory is copied by the `cp` command with the `-r` option:

```
cp -r mydir mydir2
```

Again, the paths to the directories are not needed, because the original directory and its copy are located in the current working directory, and its new content can be checked:

```
ls -l
```

The `mydir2` directory must contain the same file as the original one (i.e. the `myfile3` file):

```
ls -l mydir2
```

# 10 Move and Rename Files and Directories

## 10.1 Introduction

The `mv` command is used for moving and/or renaming files and directories without creating copies. It has a lot of useful options described on the man pages:

```
man mv
```

Here it is assumed that the operation is done on a single object (file or directory) and the object to be moved/renamed is located in the current working directory. The second condition means that one can specify only the name of the object without its path. In all cases, the `mv` command requires at least two arguments: the source and the destination.

A file is moved from the current working directory into another one by the command

```
mv  file_name  path/to/another_directory
```

The new location of the file must be specified by the absolute or relative path. If the path to the new location is appended by a file name, the original file is moved and renamed:

```
mv  file_name  path/to/another_directory/new_file_name
```

If the path to the new location is omitted, but the new file name is given, the original file is only renamed in the current working directory:

```
mv  file_name  new_file_name
```

**Remark**: If a file with the same name exists in the directory of destination, the shell will overwrite it without asking. The shell will ask about overwriting, if the `-i` option is used with the `mv` command.

The commands applied for moving and/or renaming directories are similar[6]:

```
mv  directory_name  path/to/another_directory
mv  directory_name  path/to/another_directory/new_directory_name
mv  directory_name  new_directory_name
```

The meanings of these operations are similar to those applied to files and discussed above in the present section.

**Remark**: If a directory is moved (renamed), it should have the name different from names of files and directories located in the directory of destination.

### 10.2 Exercises

After performing the exercises of section 9.2, the three directories, `mydir`, `mydir2`, and `testdir`, are located in the user home directory:

```
cd ~
ls -l
```

The `mydir` directory contains the `myfile3` file to be renamed:

```
cd mydir
ls -l
```

After changing to `mydir`, it is the current working directory, and renaming is done easily:

```
mv myfile3 myfile4
ls -l
```

The next step will be done in the user home directory:

```
cd ~
```

The `mydir2` directory is moved into the `mydir` directory with renaming `mydir100`:

```
mv mydir2 mydir/mydir100
```

To check that the operation has been done, the `ls` command is used:

```
ls -l
ls -l mydir
```

It can be seen that the current working directory (i.e. the user home directory) does not contain `mydir2`, but the `mydir` directory contains `mydir100`. The `mydir100` directory must have the same content as `mydir2` had (basically it is `mydir2` but with the new name).

## 11 Delete Files and Directories

### 11.1 Introduction

Any file or directory can be deleted by applying the `rm` command:

```
rm  file_name
rm  -r  directory_name
```

In the case of directories, the `-r` option is added. When a directory is deleted, all its files and subdirectories are deleted as well. If only the name of the file (or directory) is specified, is is

[6] The `mv` command does not require any option for directories as it is in the case of the `cp` command.

assumed to be located in the current working directory. Otherwise, the absolute or relative path is needed.

**Remark**: If a file or a directory was deleted, it cannot be recovered. The shell does not have any mechanism similar to Recycle or Trash bin.

**Remark**: The shell does not ask about deleting files and directories. The shell will ask about deleting, if the `-i` option is used with the `rm` command.

## 11.2 Exercises

After completing the exercises of section 10.2, the `mydir` directory located in the user home directory should contain the `mydir100` directory and the `myfile4` file:

```
cd ~
ls -l
ls -l mydir
```

We will create the copy of `mydir` to delete it afterwards:

```
cp -r mydir mydircopy
ls -l
```

The content of the created directory must be the same as that of the `mydir` directory:

```
cd mydircopy
ls -l
```

The `myfile4` file is deleted easily (without further recovering):

```
rm myfile4
ls -l
```

The `mydir100` directory is deleted by the same command but with the `-r` option. In addition, the `-i` option is used to demonstrate its functionality. When the `-i` option is used with a directory, the shell will ask about deleting each file and each subdirectory. If any object has to be deleted, the user types "y" or "yes" and presses `Enter` (otherwise "n" or "no"). The user can interrupt the process of deleting by pressing `Ctrl+c`. The final command for deleting `mydir100`:

```
rm -r -i mydir100
ls -l
```

The `mydircopy` directory must be empty and is also deleted (without prompting):

```
cd ..
ls -l
rm -r mydircopy
ls -l
```

The `ls` command is used before and after deleting to show the difference. It should be noted once again that there is no possibility to recover the deleted directory and all its content. Thus, the `ls` command should be always used at least before deleting.

# 12 Wildcards and Name Patterns

Wildcards are special characters applied in the shell for creating name patterns to select a group of files and directories. Such patterns are applied instead of ordinary names of files and

directories with many commands of the shell. In the present course the two types of wildcards, '`*`' and '`?`', are discussed[7]. It is important to note that not all commands of the shell are used with wildcards.

**Remark**: Do not apply wildcards for creating a group of files or directories according to a pattern. Only one single file or directory can be created in a time with using the allowed characters (see section 3.2).

**Remark**: Wildcards cannot be applied for renaming a group of files and directories by the `mv` command. If a pattern with wildcards is still applied with `mv`, the shell tries only to move a group of corresponding objects to another location (without renaming).

The '`*`' character is applied to replace **any set of characters of any length**. A pattern name is formed by adding some prefix or suffix. Having set the pattern, a group of files and directories which have the same prefix and/or suffix is selected. For example, the "`abc*`" pattern is applied to select all files and directories which have the "`abc`" prefix (e.g. the objects with the names `abc`, `abc123`). This pattern can be applied with the `ls` command to show only such files and directories:

```
ls abc*
```

The above command also shows the content of the directories which correspond to the "`abc*`" pattern[8]. As another example, the "`*.txt`" pattern is applied to select all the objects which have the "`.txt`" suffix (e.g. `myfile.txt`, `123.txt`):

```
ls *.txt
```

The '`?`' character is applied to replace **any single character** in a name. A name pattern is again formed by adding any suffix and/or prefix. For example, the "`abc?123`" pattern is applied to select files and directories which have the names with the "`abc`" prefix, the "`123`" suffix and any character in the middle (e.g. `abcX123`, `abc_123`). In this particular case, the length of the names is fixed (seven characters). The `ls` command with this pattern shows only such files and directories:

```
ls abc?123
```

Both wildcards can be combined in one pattern giving more variants for selecting files and directories. For example, the "`a?cd*`" pattern is applied to select files and directories according to the following rules:

- the first character is '`a`';
- the second position is occupied by any character;
- the "`cd`" part occupies the third and fourth positions;
- the rest of the name can be any set of characters of any length.

The names which correspond to the "`a?cd*`" pattern are `a1cd`, `a1cd123`, `a_cd123`, etc.

**Remark**: Before using name patterns with other commands (especially with `rm`), it is highly recommended to test these patterns with the `ls` command, which is harmless.

[7] More details can be found in other sources (e.g. `http://linuxcommand.org/tlcl.php`).
[8] The `-d` option is applied to list directories themselves rather than their contents.

# 13 Searching Files and Directories

## 13.1 Introduction

To find a file or a directory in the file system, the `find` command is applied:

```
find  location_for_searching  -name  file_name
```

where `location_for_searching` is the name of the directory where searching is done. If it is omitted, the current working directory is used by default. The `-name` option specifies the name of the object to be found, which can be an exact name a pattern with wildcards (see section 12). If the exact name is unknown, the name pattern can be used instead:

```
find  location_for_searching  -name  'name_pattern'
```

If the pattern is used, it must be enclosed by **single quotes**.

The `-maxdepth N` option, `N=1,2...`, is applied to limit the area for searching. Without this option the shell searches in all subdirectories of the directory-location, in their subdirectories, and so on. For example, the `-maxdepth 1` option restricts searching until the directory-location without going deeper in its subdirectories (i.e., level 1). In addition, the type of objects to be found is specified by the `-type` option:

- `-type f` is used for searching only files;
- `-type d` is used for searching only directories.

A lot of other useful options can be found on the internet[9]and on man pages:

```
man find
```

## 13.2 Exercises

In the user home directory denoted by `~` all files and directories which correspond to the `'my*'` pattern can be found as follows:

```
find ~ -name 'my*'
```

The level for searching can be restricted by applying `-maxdepth` option:

```
find -maxdepth 1 -name 'my*'
```

Finally, to find only files without directories, the `-type` option is applied:

```
find -type f -name 'my*'
```

# 14 Searching Text Patterns in Files

## 14.1 Introduction

The `grep` command is applied for searching for a given text fragment in files:

```
grep  'text_fragment'  file1  file2...
```

[9] A nice tutorial on the use of the `find` command can be found on `https://www.binarytides.com/linux-find-command-examples/`.

The text fragment can be a word or a combination of words, or a complex expression with wildcards. Similar to the file pattern of the `find` command, the text fragment must be enclosed in single quotes. If the files are given without paths, the shell checks those located in the current working directory. The files for searching can be specified by a name pattern with wildcards, and this pattern is written without any quotes:

```
grep  'text_fragment'  file_pattern
```

By default the output of the command is a list of files with lines which contain the text fragment. The output can be adjusted by using multiple options of the `grep` command described on man pages:

```
man grep
```

## 14.2 Exercises

If the exercises of section 8.2 were done accordingly, we can test the `grep` command easily. The files of the `testdir` directory (in the user home directory) will be used:

```
cd ~/testdir
ls -l
```

There should be the three identical files, `myfile`, `myfile2`, `myfile3`. The `grep` command can be used to find files with the text fragment "Hello":

```
grep 'Hello' my*
```

where all the files are specified by the name pattern `'my*'`. It is important to say that the command is case-sensitive, and the following variant shows another result:

```
grep 'hello' my*
```

To ignore case distinctions, the `-i` option can be applied:

```
grep -i 'hello' my*
```

# 15 Access Rights to Files and Directories

## 15.1 Introduction

Any file or directory in Linux is accessible for reading, writing, and executing. While for files these actions are clear, for directories they have the following meanings:

- "to read a directory" = to show its files and subdirectories (e.g. by the `ls` command);
- "to write a directory" = to create/delete a file/subdirectory within the directory;
- "to execute a directory" = to enter this directory (e.g. by the `cd` command).

Note that reading and writing are not possible (for directories), if executing is forbidden.

There are three categories of users for whom access rights (or permissions) to files and directories are set. First, a file or directory is created by the user, and the user becomes the owner of that file automatically[10]. Then, the user belongs to a group of users (at least one group) and all members of the group have access rights to a particular file or directory (the second category

[10] The owner of a file or directory can be changed by the `chown` command.

of permissions). Finally, the access rights for other users are also specified and form the third category (the users who are not the owner of a particular file or directory or do not belong to a corresponding group).

The access rights to files and directories are set by the same command:

```
chmod  access_rule  file_name
```

The first argument of `chmod` (the access rule) can be specified by a symbolic notation[11]. The notation is followed by the name of a file or a directory. If the same rule is applied to several files and directories, they are given in the command line one by one. A name pattern with wildcards can be applied instead of the series of names.

The symbolic notation is studied in the present course for the sake of simplicity, and the corresponding access rule forms from three characters. The **first character** denotes the category: '**u**' is for the owner (user), '**g**' for the group, '**o**' for other users, '**a**' for all users. The **second position** is reserved for the permission itself: the '+' character (plus) is to allow an action; the '**-**' character (minus) is to forbid. The **third position** specifies the single action: to **r**ead, to **w**rite, to e**x**ecute denoted by the characters '**r**', '**w**', '**x**', respectively. For example, the access rule '`u-w`' applied to a file means that the owner is not allowed to change the file. Note, different access rules for every category of users are set separately, but if the rule is the same for all users, it can be set for them by one command with using the notation '**a**' (e.g. '`a-w`', and all users are not allowed to change).

A simple way to check permissions is to use the `ls -l` command (see section 5). The very left column starting from the second position contains access rules for the three actions and for the three categories of users (i.e. totally 9 positions):

```
drwxrwxrwx
-rwxrwxrwx
```

The first character of the column (from left to right) is reserved and used to denote a directory ('d') or a file ('-'). The next three characters describe the permissions for the owner (the first category): the first character denotes reading (`r`), the second one writing (`w`), and the third executing (`x`). If any of these actions is allowed, the corresponding letter is written, otherwise the '-' character is shown. For example, the `rw-` combination related to a file means that the file can be read and rewritten (or changed) but cannot be executed (by the owner). Characters on positions 5-7 and on positions 8-9 describe access rights for the group and for other users, respectively (the second and third categories). The actions and their notations are the same for all the categories: `r`, `w`, `x`.

## 15.2 Exercises

The `testdir` directory created in previous exercises and located in the user home directory is used for testing the `chmod` command:

```
cd ~
ls -l
```

The `ls` command shows that reading, writing, and executing are allowed for the owner (user). For example, the content of this directory can be shown:

```
ls -l testdir
```

To forbid ('-') reading ('r') the directory content for the user ('u'):

[11] An octal notation can be applied with the `chmod` command instead of the symbolic notation.

```
chmod u-r testdir
```

The permissions should be changed, and the content of the `testdir` directory is hidden:

```
ls -l
ls -l testdir
```

To allow ('+') reading ('r') for the user ('u') again:

```
chmod u+r testdir
```

The permissions for access to files are set similarly, and for testing one of the files of `testdir` will be used, for example the `myfile` file:

```
cd testdir
ls -l
```

The `ls` command shows that this file can be read and rewritten. Reading is checked easily:

```
cat myfile
```

To forbid ('-') reading ('r') for the user ('u'):

```
chmod u-r myfile
cat myfile
```

# 16 Process Control

## 16.1 Introduction

Sometimes programs stop to work properly and cannot be closed in a usual way. The shell has a mechanism to terminate such programs, but the user has to know a unique number to identify a program to be terminated. In a Linux system every program being executed has an identification number called a **process ID** or a **PID**. A list of programs run by one user and their PIDs are shown by the command

```
ps x
```

If the output is too long, the `less` utility can be applied (see section 4.2):

```
ps x | less
```

When the PID is known, the program can be terminated by the command

```
kill program_pid
```

In this case, the system sends a signal to stop, and the program terminates by itself. If the program does not respond at all, the user can use another variant of the `kill` command:

```
kill -9 program_pid
```

The program is terminated by the system and does not do anything (e.g. saving data), the system does all the work for terminating. Thus, it is recommended to apply the first variant of the `kill` command and then to use it with the `-9` option.

There are situations when it is possible to stop a program by pressing `Ctrl+C` (e.g. a program produces too much output on the screen for a long time). In such cases the PID is not needed.

**Remark**: A lot of useful information can be found by using the `top` command (see corresponding man pages, `man top`).

# 17 Redirection and Pipelines

## 17.1 Introduction

Many commands of the shell generate output shown on the screen. Sometimes it is convenient to write output to an ordinary file stored on a disk. This can be done by taking into account a special file called **standard output**, which is usually linked with the screen. The output generated as a result of the execution of a command comes into this file. The standard output file can be replaced by an ordinary file in two ways:

```
COMMAND > file_name
```

```
COMMAND >> file_name
```

If a file with the same name does not exist (in the same directory), a new file with the output of the command is created in both cases. The significant difference between the operators `>` and `>>` is revealed, if a file with the same name exist. In the first variant of redirection by `>`, the old file is rewritten (without prompting). In the second variant (with `>>`), the output of the command is appended to the old file.

In addition to standard output, there are other two special files called **standard error** (attached often to the screen) and **standard input** (usually connected with the keyboard). They can also be replaced by ordinary files, but the syntax is different and more complex than for the standard output.

The concept of redirection is applied further for forming a combination of several commands divided by the operator "`|`" and called a **pipeline**:

```
COMMAND1 | COMMAND2 | COMMAND3 | ...
```

The idea of the pipeline is to use output of one command as input of the next command, which in turn generates new output for the next command, and so on[12]. This allows to change the output of the first command and to represent it in a different form (e.g. to sort data, to reduce information). The simplest pipelines are applied in some exercises of the present manual (e.g. see section 4.2).

## 17.2 Exercises

The exercises of the present section are completed in the user home directory:

```
cd ~
```

The redirection of output is demonstrated by using the `ls` command:

```
ls -l > ls_output
```

The `ls_output` file should appear in the current working directory:

```
ls -l
```

The content of the file can be checked by the `less` program or by the `cat` command:

```
less ls_output
cat ls_output
```

It can be seen that the file contains the output of `ls` as it would be printed on the screen. One can also note that the `ls_output` file is present in the list of files, which means that it had been created before `ls` was executed. If the command "`ls -l > ls_output`" is repeated, the content of

[12] The output of the whole pipeline can be redirected finally into an ordinary file.

the file should be the same, because the file is rewritten each time. If the redirection is done by using the second operator, `>>`, the content is appended:

```
ls -l >> ls_output
less ls_output
```

As an example of a pipeline, the `grep` program (see section 14) is used to find a program in the list of processes generated by the `ps` command. First, a new session of the shell is opened (a new login to the cluster), and the `ls_output` located in the user home directory is opened in the `nano` editor:

```
nano ~/ls_output
```

After opening the file, in the old session we will try to find `nano` by the command

```
ps x
```

In this example `grep` takes the output of the command `ps x` forming the pipeline:

```
ps x | grep 'nano'
```

A possible output can be the following one:

```
15844 pts/8    S+     0:00 nano ls_output
20467 pts/2    S+     0:00 grep --color=auto nano
```

Note that there are two records found among processes by using the text pattern `'nano'`. One record identifies the program which is run in the second session, and this is exactly what is needed. Another one is connected with the `grep` command, which was run in the system when the `ps` command was executed. The first column of the output contains PIDs, and the pipeline shown above is a simple way to know the PID of any program which is running in the system (it can be used then in the `kill` command).

# 18 Module System

Any program the user needs for working can be run similar to a command of the shell. There are a lot of applications and programs installed on the cluster, and many programs have several versions installed simultaneously. To make the work more ordered, almost all software packages available on the cluster are organized in a module system. The user has to find and to load a particular program or application to use it. A module of a particular program can be found as follows:

```
module avail name_software
```

where `name_soft` is the name or a part of the name of the program. After searching, the list of available modules is shown. Because any program can have more than one version, a particular version can be specified. For example, the software package called `my_software` has version `x.y`. The corresponding module is loaded as follows:

```
module load my_software/x.y
```

If the user specifies only the name, the default version of the software is loaded:

```
module load my_software
```

The list of all modules loaded during the current session is shown by the command

```
module list
```

If some module of the list is not needed, it can be unloaded:

```
module unload my_software/x.y
```

Finally, all the modules of the list can be unloaded by the single command:

```
module purge
```

Note that all loaded modules are unloaded automatically after closing the session. If a set of modules is frequently used, they can be loaded by one command. First, the commands for loading the corresponding modules are written to a text file, for example, the `my_modules` file. Then, the set of these modules is loaded by using the file as follows:

```
. path/to/my_modules
```

The dot at the beginning is necessary. Alternatively one can do the same by the command

```
source path/to/my_modules
```

# Appendix 1: Assistance in Shell

The work in the shell without using menus, dialog boxes, and the mouse assumes only typing of commands, paths to files, and so on. Nevertheless, there are some tricks which help to work more efficiently and faster with the command line interface.

1. Auto completion of commands.

   Commands of the shell and commands for running many programs (e.g. `nano`) can be input almost automatically. To make auto completion of a command, one has to type its first several letters and press the `Tab` button. If the number of letters is not enough to recognize the command, more letters should be added.

2. Auto completion of names of files and directories works similar to auto completion of commands.

3. Copying and pasting text by the mouse.

   To copy any part of output shown on the screen, one has to highlight it by holding the right button of the mouse. By pressing the middle button of the mouse, the text is copied into the shell prompt.

4. Commands previously executed can be invoked by pressing the `up` or `down` arrow.

5. Commands previously executed can be also found in a history of commands.

   The history of commands is shown by the `history` command. Because the output can be long, the `less` program can be used to view the history:

   ```
   history | less
   ```

   Any command of the history can be repeated by using its number `N`:

   ```
   !N
   ```

6. The path to the user home directory is denoted by the tilde, `~`.

   This notation can be used in all commands to form any path to files and subdirectories of the user home directory.

7. One can return back to the previous working directory by the command

```
cd -
```

The dash notation of the previous directory, `-`, cannot be used in paths and with other commands.

8. Options of many commands denoted by one letter can be combined.

   For example, the command `ls -l -a` used with the two options is written simpler:

```
ls -la
```

# Appendix 2: Quick Reference

The list of commands of the bash shell covered in the present manual:

- `exit` - to close the shell (section 2);
- `man` - to open man pages of commands (section 2);
- `pwd` - to show the current working directory (section 4);
- `ls` - to list content of a directory (section 4);
- `less` - to view output or content of a text file (section 4);
- `cd` - to change the current working directory (section 7);
- `mkdir` - to create a new directory (section 7);
- `file` - to determine a file type (section 8);
- `cat` - to concatenate files and print their content (section 8);
- `cp` - to copy files and directories (section 9);
- `mv` - to move and/or rename files and directories (section 10);
- `rm` - to delete files and directories (section 11);
- `find` - to search for files and directories (section 13);
- `grep` - to search text fragments in files (section 14);
- `chmod` - to change access rights to files and directories (section 15);
- `ps` - to print a list of running programs (section 16);
- `kill` - to terminate programs (section 16);
- `history` - to print the history of commands (section 18).

In addition, the simple text editor `nano` is introduced in section 8.

Hessisches Kompetenzzentrum für Hochleistungsrechnen (HKHLR) is funded by the Hessian Ministry of Sciences and Arts.