The operating system Linux

An introduction

Joachim Puls and Michael Wegner

Contents:

First Afternoon:

1 General remarks on the operating system UNIX/Linux

2 First steps at the computer

3 File systems

4 Editing and printing text files
More important commands

Second Afternoon:

5 UNIX-shells

6 Process administration

Additional material

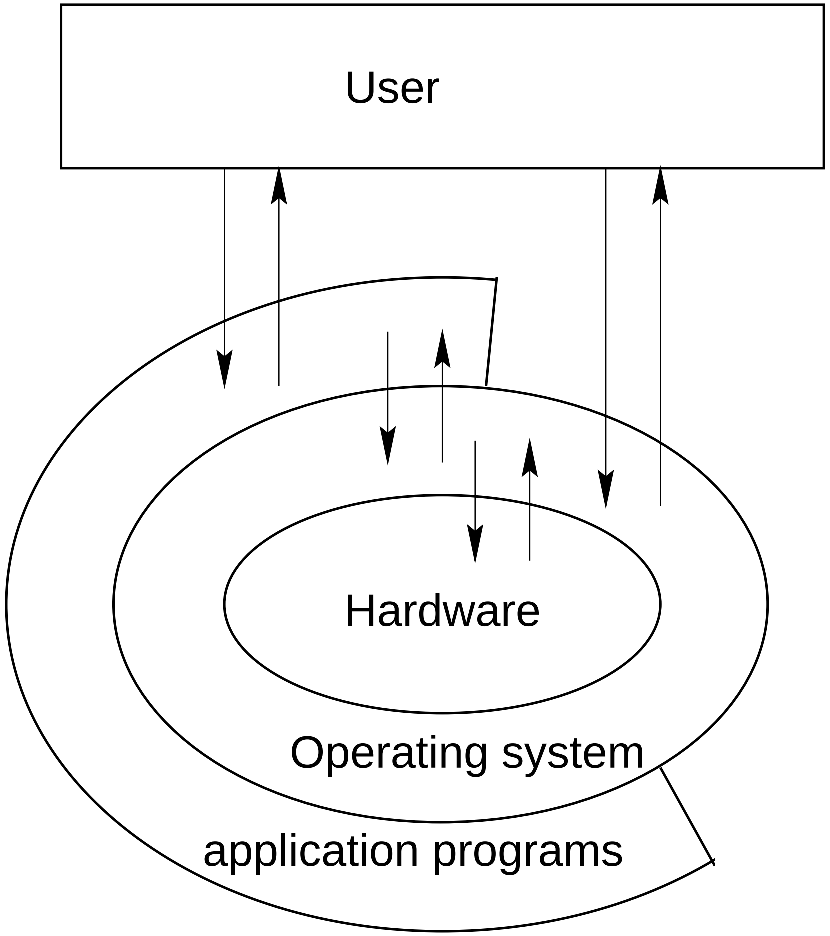
- `vi` basics: `vi_brief.pdf`
- reference for `vi`: `vi_reference.pdf`
- reference for `emacs`: `emacs_reference.pdf`

Acknowledgements. Many thanks to Tadziu Hoffmann for carefully reading the manuscript and useful comments.

1 General remarks on the operating system UNIX/Linux

Classification of UNIX/Linux

UNIX is a *Multi-User/Multi-Tasking operating system* and exists in many different versions ("derivates"): Solaris, AIX, XENIX, HP-UX, SINIX, **Linux**.



Operating system (OS): Sum of all programs which are *required to operate a computer* and which control and monitor the application programs.

Essential features

UNIX

- has been originally written in the programming language C, and is therefore a classical platform for C-programs. UNIX contains well suited environments for program development (C, C++, Java, Fortran, ...).

- is mainly used for scientific-technical applications on mainframes and workstations, but has become, because of **Linux**, also popular for classical PC-applications throughout the last years.

- is perfectly suited for application in networks. Larger systems and networks require an administrator.

- offers various alternatives for the solution of most tasks. The multitude of commands (more than in any other OS) are brief and flexible.

- is originally command-line oriented, but can be used via a graphical user interface (*X Window system*).

Linux is available (also via internet) in different *distributions* (S.u.S.E., Fedora, Debian etc.). Meanwhile there is a variety of *direct-start (live) systems*, which can be started, without installation, directly from CD or other bootable storage devices (Knoppix, Ubuntu, . . .). The source code of Linux is free.

Literature

- **Peek, J., et al.**: *Unix Power Tools.*
  O'Reilly Media 2002 (3rd edition).

- **Gilly, D., et al.**: *UNIX in a Nutshell.*
  O'Reilly, Köln. 1998 (1st edition).

- **Wielsch,M.**: *Das große Buch zu UNIX.*
  Data Becker, Düsseldorf. 1994 (1st edition).

- and numerous other text books

- **online-tutorial**
  http://www.ee.surrey.ac.uk/Teaching/Unix

2 First steps at the computer

User, logon, logoff

Since UNIX is a *multi-user* operating system, it can deal with several users simultaneously. Each user needs a *user account*.

Each user has a personal environment (*home directory, shell*), which can be accessed only by her-/himself (and by the system adminstrator and those people who know the password – legitimate or by hacking).

Inside the system the user is identified by his *user ID* (UID) and his group identity (*group ID, GID*).

There are two user types:

- ‘normal’ users with *restricted* rights and the
- system administrator (*root*) with all privileges. The latter is responsible for the installation, configuration and maintenance of the system as well as the user administration.

Each user has to logon and to logoff from the system (*login/logout*).
Each user account is protected by a *password*.

Exercise:

Login to the system with your user account.

Graphical user interface

Originally, UNIX is command-line oriented. The *X Window system* enables convenient interaction via a window-oriented graphical interface, similar to other OS.

The *window manager* is responsible for the management and display of the individual windows. Each window manager (and there are a variety of such managers) can be distinguished by its own *Look and Feel* (appearance of window decorations and control devices etc.). Most window managers can be choosen at the login-menu.

Examples for simple window managers:

- `twm`: very simple and resource-saving
- `mwm`: Motif window manager, more common and more advanced
- `xfce`: convenient, simple, and resource-saving (recommended for use in virtual machines)

Moreover, almost all Linux distributions provide graphical *desktop environments* such as **KDE** or GNOME, which have a functionality far beyond simple window managers.

command `xterm`

Syntax:

```
xterm [options]
```

Though there is a graphical interface, UNIX needs the possibility for direct command input for practical use. Therefore, at least one terminal window needs to be open. This can be accomplished via the window manager or the desktop environment (‘console’)

More windows can then be opened with the command `xterm`.

Generally, all UNIX commands have a variety of *options*, which usually begin with `-`. For the commands which will be introduced in the following, we will provide only the most important ones.

Example:

```
wegner@arber:~ > xterm -geo 80x40 -fn 10x20
```

The command `xterm` is called with two options `-geo, -fn`, which, in this case, need additional *arguments* (width and height of window, font name & size).

Exercise:

1. *Open a terminal window (“terminal program”) via KDE.*
2. *From there, start another* `xterm`*.*

command `man`

Syntax:

```
man command
man -k expression
```

displays the manual pages ("man pages") for the provided `command`. `man -k` searches for man pages containing the `expression` in the `NAME` section. A man page usually consists of the following sections

- `NAME` command and purpose
- `SYNOPSIS` syntax of command
- `DESCRIPTION` of command effect
- `FILES` which are modified and/or needed
- `OPTIONS` if present

- `EXAMPLE(S)` for application (rarely)

- `BUGS` errors, if known

- `SEE ALSO` other commands in the same context

Exercise:

Display information about the command `xterm`*.*

command `passwd`

Syntax:

```
 passwd
```

sets a new password.

Passwords should be constructed from a combination of letters, digits and special characters, and should not appear in any dictionary or similar list. Otherwise, the password can be hacked by systematic search algorithms.

The command to set/change the password and the required conventions (length, number of digits/letters/special characters) vary from system to system. When a new account is created for you, the administrators should tell you how to change your password (`passwd`, `kpasswd`, ...).

The following example is a common one, e.g., valid for the workstations at the CIP Pool - but not for the workstations of the USM.

Example:

```
wegner@arber:~ > passwd
Changing password for wegner
Old password: myoldpasswd
Enter the new password
  (minimum of 5, maximum of 8 characters)
Please use a combination
  of upper and lower case letters and numbers.
New password: mynewpasswd
Re-enter new password: mynewpasswd
Password changed.
```

Example:

```
----> Ihr neues Passwort ist in 5 Minuten
      im gesamten Pool aktiv! <-----
Connection to 141.84.136.1 closed.
wegner@arber:~ >
```

command `who, whoami`

Syntax:

```
who
whoami
```

`who` displays information about all users which are logged into the system

- user name,
- terminal where the corresponding user is working,
- time of login.

`whoami` is self-explanatory.

Example:

```
wegner@arber:~ > whoami
arber!wegner    pts/5    Oct 20 12:45
```

Working at external terminals

To login to a remote host, one has to provide the corresponding IP address, either numerical or as the complete host name `name.domain`. In local networks (CIP-Pool), the brief host name (without domain) is sufficient. To establish the connection and to encrypt the transmitted data, one should use exclusively the so-called “secure” commands. Avoid `ftp` and use `sftp` instead. With `ftp`, even the password is *not* encoded!

command `ssh`

Syntax:

```
 ssh (-4) -X -l username hostname
 ssh (-4) -X username@hostname
```

Enables logging in to an arbitrary host which can be located via an IP address (if one knows the user account and the password). Logoff with `exit`, `Ctrl-D` or `logout`.

In case, the option -4 (without brackets) forces an IPv4 connection (if IPv6 is not working)

Example:

```
wegner@arber:~ > ssh -X -l wegner lxsrv1.lrz-muenchen.de
Password: mypasswd
Last login: Sun Oct 22 ...
*********************
Mitteilungen
*********************
wegner@lxsrv1:~ > logout
Connection to lxsrv1.lrz-muenchen.de closed.
```

OR (if connection within “own” cluster)

Example:

```
wegner@arber:~ > ssh -X wegner@arber
Last login: Sun Oct 22 ...
etc. (keine Passwort-Abfrage)
```

An additional advantage of the `secure shell` is that the remote host `hostname` can display X applications on the local terminal. For certain hosts, the command `ssh` requires the option `-X` to enable this feature. The option `-X` should only be used with high bandwidth connections (i.e. when connecting to a remote host in the same network). If you want to work remotely with a lower bandwidth, you should use a VNC or similar remote desktop solution.

command `scp`

To copy files from one host to another, the command `scp` (“secure copy”) is used, see also `cp`. More on this later.

Syntax:

```
 scp (-4) file1 username@hostname:file2
 scp (-4) username@hostname:file1 file2
```

The first command copies the local file `file1` to the external host under name `file2`, the second command vice versa. Note the colon! `scp -r` enables to copy complete directories *recursively*, compare `cp -r`.

3 File systems

Logics, file types

“In UNIX everything is a file.”

The following *file-system objects* can be found

- ‘normal’ (text-) files
- executable files (binary files or *shell scripts*)
- directories
- device files
- *pipes*
- symbolic or hard *links* (references to files)

All files and file system objects are ordered within a hierarchical *file tree* with exactly one *root directory* '/'.

In contrast to the MS-Windows file system, the UNIX file system does not distinguish between different drives. All physical devices (hard disks, DVD, CDROM, USB, floppy) are denoted by specific files inside a certain directory within the file tree (usually within `/dev`). Often these directories are linked to other directories like `/home/moon` on the USM machines. Links are symbolic connections let you access a file/directory from more than one directory.

File names consist of a sequence of letters, digits and certain special characters, and must not contain *slashes* (for convenience, they should neither contain empty spaces).

Avoid characters which might be interpreted by the *shell* in a special way.

A file can be referenced within the file tree by either an *absolute* or a *relative path name*. An absolute path name consists of all directories leading to the file and the file name, and always begins with a / (the root directory).

In many shells and application programs, the tilde denotes the home directory.

command `pwd`

Syntax:

```
 pwd
```

displays the current directory.

Example:

```
wegner@arber:~ > pwd
/home/wegner
wegner@arber:~ >
```

Exercise:

Display the current directory.

command `cd`

Syntax:

cd [directory]

Changes into the given directory, or into the home directory when no parameter is provided.

As in MS-DOS/Windows, ".." denotes the parent and "." the current directory.

Example:

```
wegner@arber:~ > cd /home/puls
wegner@arber:/home/puls > pwd
/home/puls
wegner@arber:/home/puls > cd ..
wegner@arber:/home > pwd
/home
```

```
wegner@arber:/home > cd
wegner@arber:~ > pwd
/home/wegner
wegner@arber:~ >
```

Exercise:

Change to the directory `/usr/share/templates` *and back to your home directory. (→ file name completion with* `TAB`*)*

Check for successful change with `pwd`*.*

Search pattern for file names

In principle, the *shell* is a specific program which deals with the interpretation of input commands. If these commands have parameters which are file names, several files can be addressed simultaneously by means of a search pattern, which is *expanded* by the shell. In any case, the file name expansion is performed *prior* to the execution of the command.

expression	meaning
`*`	‘almost’ arbitrary (incl. empty) string of characters
`?`	a *single* ‘almost’ arbitrary character
`[...]`	a range of characters
`[!...]`	a negated range of characters

‘almost’ arbitrary: leading dot (e.g., hidden files, `../` etc.) excluded

command `ls`

Syntax:

```
ls [-alR] [file/directory]
```

displays the names (and, optionally, the properties) of files or lists the content of a directory. File and directory names can be be absolute or relative.

Important options

`-a` list also files/directories which begin with a dot (hidden)

`-l` long listing format. Displays permissions, user and group, time stamp, size, etc.

`-R` for directories, all sub-directories will be displayed recursively.

Example:

```
wegner@arber:~ > ls
hello*  hello.cpp  hello.f90  hello.py
wegner@arber:~ > ls -a
./           .bash_history  .netscape/  hello.cpp
../          .bashrc*       .ssh/       hello.f90
.Xauthority  .history       hello*      hello.py
wegner@arber:~ > ls /var/X11R6
app-defaults/  bin/    lib@  sax/
scores/        xfine/  xkb/
```

```
wegner@arber:~ > ls .b*
.bash_history  .bashrc*
wegner@arber:~ > ls [a-h]*
hello*  hello.cpp  hello.f90 hello.py
wegner@arber:~ > ls *.?[9p]?
hello.cpp  hello.f90
wegner@arber:~ >
```

Exercise:

List the complete content of your home directory.
What is displayed with `ls .*` *?*

Copy, move and delete files/directories

In addition to `ls` there are other commands for working with files which can be used together with file name patterns.

command `mkdir, rmdir`

Syntax:

```
mkdir directory
rmdir directory
```

`mkdir` creates an empty directory, `rmdir` deletes an *empty* directory.

Example:

```
wegner@arber:~ > ls
hello*  hello.cpp  hello.f90  hello.py
wegner@arber:~ > mkdir numerik
wegner@arber:~ > ls
```

```
hello*  hello.cpp  hello.f90  hello.py  numerik/
wegner@arber:˜ > rmdir numerik
wegner@arber:˜ > ls
hello*  hello.cpp  hello.f90  hello.py
wegner@arber:˜ >
```

Exercise:

Create a directory `yourname_exercise` *within your home directory, where* `yourname` *is your* actual *name.*

command $\boxed{\texttt{cp}}$

Syntax:

```
cp file1 file2
cp file1 [file2 ...] directory
cp -r dir1 dir2
cp -r dir1 [dir2 ...] directory
```

copies files or directories. The original file/directory remains unmodified.

option:

`-r` directories are copied recursively with all sub-directories.

Several possibilities:

```
cp file1 file2
```

`file1` is copied to `file2`. Attention: if `file2` already exists, it is overwritten (mostly without warning), and the original `file2` is lost!!!

```
cp file1 [file2 file3] dir
```

If `dir` exists, `file1 [, file2, file3]` are copied *into* `dir`. If `dir` does not exist, you get an error warning (for more than two arguments), or, for two arguments, `dir` is interpreted as a file name and `file1` is copied to a *file* named `dir`.

```
cp -r dir1 dir2
```

If `dir2` already exists, `dir1` is recursively copied *into* `dir2`. If `dir2` does not exist, a recursive copy of `dir1` is created and named `dir2`.

```
cp -r dir1 dir2 dir3 dir4
```

If `dir4` already exists, `dir1`, `dir2`, `dir3` are copied *into* `dir4`. If `dir4` does not exist, you get an error warning, as well as for other combinations of files and directories within the command.

Example:

```
wegner@arber:~ > ls
hello*  hello.cpp  hello.f90  hello.py  numerik/
wegner@arber:~ > cp hello.py hello2.py
wegner@arber:~ > ls
hello*     hello.f90   hello.cpp
hello.py   hello2.py   numerik/
wegner@arber:~ > cp hello.py numerik
wegner@arber:~ > ls numerik
```

```
hello.py
wegner@arber:~ >
```

Exercise:

a) Check whether the directory `ubung0` *is present in your home directory. If not, copy, via* `scp`*, the directory* `ubung0` *from account/host* `numprakt@ltsp08.usm.uni-muenchen.de` *to your home directory.*
b) Copy the files from `ubung0` *into your directory* `yourname_exercise`*.*

command `mv`

Syntax:

```
mv file1 file2
mv file1 [file2 ...] directory
mv dir1 dir2
mv dir1 [dir2 ...] directory
```

Rename or move files or directories. Similar to `cp`, but original is ‘destroyed’. First command from above renames files, other commands move files/directories. (Actually, only the pointer in the ‘inode table’ is changed, but there is no physical move – except if you move the file to another file system).

Note: *no* option `[-r]` required

Several possibilities, analogue to `cp`.

Example:

```
wegner@arber:~ > ls
hello*     hello.f90   hello.cpp
hello.py   hello2.py   numerik/
wegner@arber:~ > mv hello2.py hello3.py
wegner@arber:~ > ls
hello*     hello.f90   hello.cpp
hello.py   hello3.py   numerik/
wegner@arber:~ > ls numerik
hello.py
wegner@arber:~ > mv hello3.py numerik
wegner@arber:~ > ls
hello*  hello.cpp  hello.f90  hello.py  numerik/
wegner@arber:~ > ls numerik
hello.py  hello3.py
wegner@arber:~ >
```

Exercise:

1. *Rename your directory* `yourname_exercise` *to* `yourname_ exercise0`*. This will be your working directory for the following exercises.*

2. *Move the file* `.plan` *from* `yourname_exercise0` *to your home directory. Try to move an arbitrary file from your home directory to the root directory. What happens?*

command `rm`

Syntax:

```
rm [-irf] file(s)/directory(ies)
```

Delete files and/or directories. After deleting, the deleted files cannot be recovered! Use `rm` only with greatest caution. E.g., the command `rm -r *` deletes recursively (in most cases without further inquiry) the complete file tree below the current directory (leaving the hidden files/directories beginning with `.` though).

Options:

- `-i` delete only after confirmation
- `-r` directories will be recursively deleted (with all sub-directories)
- `-f` force: suppress all safety inquiries.

Note: Varying from system to system, `rm` without the option `-f` might need a confirmation or not (the latter is the standard).

Example:

```
wegner@arber:~/numerik > ls
hello.py  hello3.py
wegner@arber:~/numerik > rm -i hello3.py
rm: remove ‘hello3.py’? y
wegner@arber:~/numerik > ls
hello.py
wegner@arber:~/numerik >
```

File permissions/Access rights

The UNIX file system distinguishes between three different access rights or *file mode bits*. (Note: actually, there are more access rights, but these are of interest only for administrators.)

`r` read: permits the reading of file contents, or, for directories, the listing of their content.

`w` write: permits the modification of files (incl. delete). To create or delete files, the parent directory(ies) need write access as well!

`x` execute: permits the execution of binary files (commands, programs) and of shell scripts from the command line. For directories, the `x` bit is required to change into this directory and to access the files/directories inside.

Access rights are individually defined for

`u` the owner of the object

`g` the group to which the object belongs

`o` all other users

`a` all users (i.e., u + g + o)

The access rights of a file can be changed by means of the command `chmod`.

command `chmod`

Syntax:

```
chmod [ugoa][+-=][rwx] file(s)/directory(ies)
```

Change the access rights of files or directories. These rights are displayed by `ls -l` according to the pattern

```
uuugggooo
rwxrwxrwx
```

Example:

```
wegner@arber:~/numerik > ls -l
total 4
-rw-r--r-- 1 wegner stud 100 Oct 20 15:02 hello.cpp
wegner@arber:~/numerik > chmod go+w hello.cpp
wegner@arber:~/numerik > ls -l
total 4
```

```
-rw-rw-rw- 1 wegner stud 100 Oct 20 15:02 hello.cpp
wegner@arber:~/numerik >
```

Exercise:

1. *Remove the execution right for the directory* `yourname_exercise0`. *Try to change to the directory.*

2. *Remove all rights for the file* `linux.txt`*! How can this be undone?*

4 Editing and printing text files

To modify (= edit) the content of a text file, an *editor* is needed. Within UNIX there is a variety of editors, which can be distinguished mostly with respect to ease of use and memory requirements.

The editor `vi` and `vim`

`vi` is the only editor which is present on *all* UNIX systems. The editor `vi`

- can be completely keyboard controlled
- is extremely flexible
- rather difficult to learn

`vim` is a derivate from `vi`, and can be controlled also by the mouse.

Those of you who enjoy a challenge should learn using this editor.

A somewhat simpler and more convenient alternative, which is also implemented in (almost) all UNIX systems, is

The editor `emacs`

The editor `emacs` works in an own window, and can be controlled (in addition to keys) by menus and mouse. `emacs` can do much more than only editing - from Org Mode to controlling a coffee maker.

Exercise:

1. *Edit the program* `hello.py`*.*

   Start `emacs` *with* `emacs hello.py &` *from the command line. The* `ampersand, &`*, ensures that* `emacs` *runs in the* background*, so that you can continue your work from the command line, independent from the* `emacs` *window (see Section 'Process administration').*

   Try to change the comments in those lines starting with `!`

2. *Split the screen with* `Ctrl X 2`*. Return to one screen with* `Ctrl X 1`

3. *Save the file with* `Ctrl X Ctrl S`*!*

4. *Quit* `emacs` *with* `Ctrl X Ctrl C`*!*

Note: Whenever you save a `file` in `emacs`, a backup of the previous version is automatically created under name `file~`.

Examples for additional possibilities

- Advanced use of man pages (e.g., searching for certain strings): In `emacs` , type `Esc X man CR xterm` to open the `xterm` man pages.

To search for ‘terminal’, type `Ctrl S terminal`, and then `Ctrl S` for the next instance.

- Spell checking within `emacs` via the the command `Esc x ispell`. Try it!

Try to learn the most important *key controlled* commands. After a while, you can edit your files much faster than by using mouse and menus. A quick reference is provided in the appendix.

command `cat`

Syntax:

```
cat  file
```

displays the content of a file on the standard output channel (usually the screen).

As many other UNIX commands, `cat` is a *filter*, which can read not only from files, but also from the standard input channel (usually the keyboard via the command line). Thus, `cat` can be used to directly create smaller text files. In this case, the output has to be *re-directed* into a file via `>`. `cat` then expects some input from the command line, which must be finished with `Ctrl D`.

Example:

```
wegner@arber:~ > cat > test
This is a test.
```

```
^D
wegner@arber:~ > cat test
This is a test.
wegner@arber:~ > more test
This is a test.
wegner@arber:~ >
```

Exercise:

1. *View the file* `.plan`*.*

2. *View the file* `linux.txt`*! Is* `cat` *a suitable tool?*

command `more`

Syntax:

```
  more file
```

`more` permits to view also larger files page by page. Important commands within `more` are `b` to scroll back and `q` to quit.

Example:

```
  wegner@arber:~ > more hello.f90
```

Exercise:

View the file `linux.txt` *with* `more`*. Which effect do the keys* `CR` *and* `SPACE` *have?*

command `lpr, lpq, lprm`

Syntax:

```
lpr -Pprintername file
lpq -Pprintername
lprm job_id
```

`lpr` prints a file on the printer named `printername`. To find out the `printername`, ask a colleague or your administrator.

`lpq` lists all print jobs on the printer `printername` and provides the corresponding `job_ids`.

`lprm` deletes the print job with id `job_id` from the printing queue.

Example:

```
wegner@arber:~ > a2ps hello.py -o hello.ps
[hello.py (Python): 1 page on 1 sheet]
```

```
[Total: 1 page on 1 sheet] saved into the file ‘hello.ps’
wegner@arber:~ > ls
hello*     hello.f90  hello.py   test
hello.cpp  hello.ps   numerik/
wegner@arber:~ > lpr -Plp0 hello.ps
wegner@arber:~ >
```

Exercise:

Print the file `linux.txt`*.*

More important commands

`a2ps` converts ASCII text to PostScript. Often required to print text under Linux.

```
a2ps [options] textfile
-1, -2, ..., -9 predefined font size and page layout.
                E.g., with -2 two pages of text
                are displayed side-by-side on one
                output page.
-o              output file (*.ps)
-P NAME         send output to printer NAME
```

`diff` `file1 file2` compares two files. If they are identical, *no* output.

`touch` `file` sets the current time stamp for a file. Can be used to create an empty file.

`finger` `account` displays additional information for the user of a certain account (name of user, project, etc.)

`gv` `datei.ps` displays PostScript files and files of related formats (e.g., `*.eps, *.pdf`).

`okular` or `evince file.pdf` display (among other formats) `pdf` files and allow for simple manipulations.

`gimp` `file` starts the image manipulation program `gimp` (similar to `photoshop`). Allows to view, manipulate and print image files (e.g., `*.jpg, *.tif, *.png`).

`ps2pdf` `file.ps` converts ps-files to pdf-files. The file `file.pdf` will be automatically created.

`gzip` `file`. Compresses file via Lempel-Ziv algorithm. The file `file.gz` is created and the file file deleted. Typical compression factor ~3.

`gunzip` `file.gz`. Corresponding decompression.

`tar` "tape archive". Nowadays mainly used to create one single file from a file tree, which then, e.g., can be sent by email. Reverse process also with `tar`.

```
tar -cvf direc.tar direc
         creates (c) file (f) direc.tar from
         directory direc. Verbose progress
         is displayed (v).
tar -xvf direc.tar
         re-creates original file tree under
         original name (./direc).
tar -zcvf direc.tgz direc
tar -zxvf direc.tgz
```

```
          additional compression/dekompression
          via gzip.
```

Note: This command is extremely ‘powerful’. Either read the man pages, or use the command as given.

`locate search expression`. Lists all files and directories in the local database, which correspond to the search expression. Extremely well suited to search for files (if the database is frequently updated → system administrator)

`find` searches recursively for files corresponding to search expression within the given `path`.
Example: `find . -name '*.txt'` searches recursively for all `*.txt` files, starting within the current directory.

`grep` searches for text *within* given files.
Example: `grep 'test' ../*.f90` searches for the text `test` in all `*.f90` files in the parent directory. The most important option is `[-i]`, which forces `grep` to ignore any distinction between upper and lower case.

5 UNIX shells

The *shell* is a service program through which the user communicates with the OS and which is responsible for the interpretation of the input commands.

Different UNIX shells

Since the shell does not directly belong to the OS, a number of different shells have been developed in the course of time:

- *Bourne shell (sh)*. A well-known and widespread shell, named after its inventor Steven Bourne. An advanced derivate, the *bash*, *Bourne again shell* (note the pun) is most popular under Linux.

- *C-Shell (csh)*. Developed in Berkeley, and uses a more C-like syntax. An improved version of the C-shell is the *tcsh*.

- *Bash shell (bash)*. Advanced Bourne shell and standard on many systems.

- There is also *Zsh Shell* and *Korn Shell (ksh)*

Each shell contains a set of *system variables*, which can be augmented by user-defined variables. This set comprises the process environment for the programs running inside the shell.

Moreover, the shell can be used to run (system-) programs via *shell scripts*.

Shell scripts

Shell scripts are small programs consisting of UNIX commands and shell-specific program constructs (branches, loops etc), which behave like UNIX commands but are present in text form (instead of binary). These scripts are *interpreted* by the shell.

The syntax of shell scripts differs (considerably) from shell to shell.

Some shell scripts are *automatically* called under certain conditions:

- `.profile` and/or `.login` are executed, if present, at login (i.e., for the *login shell*), and only once.

- `.bashrc` and `.cshrc` /`.tcshrc` are called whenever a new `bash` or `csh/tcsh` is opened, respectively.

Exercise:

1. *Copy the file* `.tcshrc` *to your home directory and inspect the file.*

2. *Open a (new)* tcsh *by typing* `tcsh` *on the command line. What happens? Exit the* tcsh *with* `exit`*.*

Re-directing input and output

All UNIX commands use *input and output channels* to read data and to output data. Usually, these are the keyboard and the screen assigned to the specific user, respectively.

These standard channels can be redirected within the shell such that a command can either read directly from a file (instead from the keyboard) and/or write into a file (instead of the screen). For re-direction, use the characters '>' (for output) and '<' (for input)

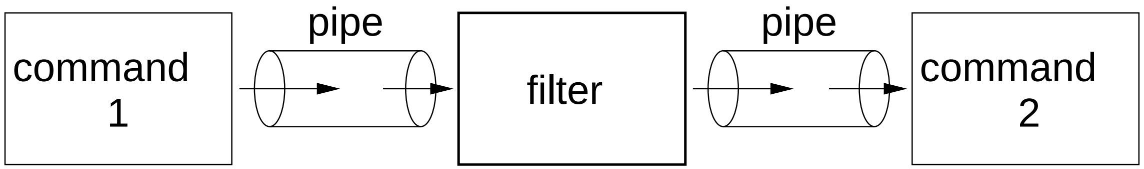
With '>>', the output will be *appended* to an existing file. If the file does not exist, this command behaves as '>'.

Example:

```
wegner@arber:~ > ls
hello.cpp  linux.txt  numerik/
hello.f90  hello.py
wegner@arber:~ > cat linux.txt > linux2.txt
wegner@arber:~ > ls
hello.cpp  linux.txt   nsmail/
hello.py  linux2.txt  numerik/
```

Pipes

Furthermore, many UNIX commands act as so-called *filters*: They read from the standard input and write to the standard output. Thus, they can be combined via so-called *pipes* such that the output of one command acts as the input of another:



Pipes are constructed on the command line by using the '|' character between commands.

A re-direction to a file with '>' or '>>' can be present only at the *end* of such a chain.

Example:

```
wegner@arber:~ > man g++ | a2ps -P printer
[Total: 151 pages on 76 sheets]
wegner@arber:~ >
```

With this pipe, the man pages for `g++` are formatted and printed via one command.

6 Process administration

A *process* is a *running* program or script and consists of

- the program/script itself and
- the corresponding environment, which consists of all required additional information necessary to ensure a correct program flow.

Characteristics of a process are (among others)

- a unique process ID (PID),
- PID of the *parent process* (PPID),
- User and group number of the *owner* and
- *priority* of the process.

Normally, when a process has been started from a shell, the shell cannot be used for other input until the end of the process. But processes and programs can also be run in the *background*. To enable this feature, the command line which calls the process/program must end with an *ampersand*, '`&`'.

Example:

```
wegner@arber:~ > firefox &
[1] 21749
wegner@arber:~ >
```

Exercise:

Start the program `xeyes` *in the background.*

command `ps`

Syntax:

```
ps [-al] [-u user]
```

Display running processes with their characteristics. Without options, only the user's own processes running in the current shell are displayed.

Important options:

- `-a` display all processes assigned to any terminal (tty)
- `-l` long format display. Additional information about owner, parent process etc.
- `-u` display all processes which are owned by a specific `user`.

Example:

```
wegner@arber:~ > ps
  PID TTY          TIME CMD
21733 pts/4    00:00:00 bash
22197 pts/4    00:00:00 xterm
22198 pts/5    00:00:00 bash
22212 pts/4    00:00:00 ps
wegner@arber:~ >
```

Exercise:

View all current processes within your shell.

command `kill`

Syntax:

```
kill [-9] PID
```

Terminates the process with number `PID`. Can be executed only by the owner of the process or by *root*.

Important option:

`-9` for ‘obstinate’ processes which cannot be terminated by a normal `kill`.

Example:

```
wegner@arber:~ > ps
  PID TTY          TIME CMD
21733 pts/4    00:00:00 bash
22197 pts/4    00:00:00 xterm
```

```
22198 pts/5    00:00:00 bash
22212 pts/4    00:00:00 ps
wegner@arber:~ > kill 22197
wegner@arber:~ > ps
  PID TTY          TIME CMD
21733 pts/4    00:00:00 bash
22214 pts/4    00:00:00 ps
[1]+  Exit 15   xterm
wegner@arber:~ >
```

Exercise:

Terminate `xeyes` *via* `kill`*.*