Chapter 27

Unix intro

Unix is an *Operating System (OS)*, that is, a layer of software between the user or a user program and the hardware. It takes care of files and screen output, and it makes sure that many processes can exist side by side on one system. However, it is not immediately visible to the user. Most of the time that you use Unix, you are typing commands which are executed by an interpreter called the *shell*. The shell makes the actual OS calls. There are a few possible Unix shells available, but in this tutorial we will assume that you are using the *sh* or *bash* shell, although many commands are common to the various shells in existence.

Most of this tutorial will work on any Unix-like platform, however, there is not just one Unix:

- Traditionally there are a few major flavors of Unix: ATT and BSD. Apple has Darwin which is close to BSD; IBM and HP have their own versions of Unix, and Linux is yet another variant. The differences between these are deep down and if you are taking this tutorial you probably won't see them for quite a while.
- Within Linux there are various *Linux distributions* such as *Red Hat* or *Ubuntu*. These mainly differ in the organization of system files and again you probably need not worry about them.
- As mentioned just now, there are different shells, and they do differ considerably. Here you will learn the *bash* shell, which is an improved version of the old *sh* shell. For a variety of reasons, `bash` is to be preferred over the *csh* or *tcsh* shell. Other shells are the *ksh* and *zsh*, which is itself an improvement over the `bash` shell.

27.1 Files and such

Purpose. In this section you will learn about the Unix file system, which consists of *directories* that store *files*. You will learn about *executable* files and commands for displaying data files.

27.1.1 Looking at files

Purpose. In this section you will learn commands for displaying file contents.

command	function
`ls`	list files or directories
`touch`	create new/empty file or update existing file
`cat > filename`	enter text into file
`cp`	copy files
`mv`	rename files
`rm`	remove files
`file`	report the type of file
`cat filename`	display file
`head,tail`	display part of a file
`less,more`	incrementally display a file

27.1.1.1 `ls`

Without any argument, the `ls` command gives you a listing of files that are in your present location.

Exercise 27.1. Type `ls`. Does anything show up?

Intended outcome. If there are files in your directory, they will be `listed`; if there are none, no output will be given. This is standard Unix behavior: no output does not mean that something went wrong, it only means that there is nothing to report.

Exercise 27.2. If the `ls` command shows that there are files, do `ls name` on one of those. By using an option, for instance `ls -s name` you can get more information about `name`.

Things to watch out for. If you mistype a name, or specify a name of a non-existing file, you'll get an error message.

27.1.1.2 `cat`

The `cat` command (short for 'concatenate') is often used to display files, but it can also be used to create some simple content.

Exercise 27.3. Type `cat > newfilename` (where you can pick any filename) and type some text. Conclude with `Control-d` on a line by itself: press the `Control` key and hold it while you press the `d` key. Now use `cat` to view the contents of that file: `cat newfilename`.

Intended outcome. In the first use of `cat`, text was appended from the terminal to a file; in the second the file was cat'ed to the terminal output. You should see on your screen precisely what you typed into the file.

Things to watch out for. Be sure to type `Control-d` as the first thing on the last line of input. If you really get stuck, `Control-c` will usually get you out. Try this: start creating a file with `cat > filename` and hit `Control-c` in the middle of a line. What are the contents of your file?

Remark 19 *Instead of* `Control-d` *you will often see the notation* `^D`. *The capital letter is for historic reasons: you use the control key and the lowercase letter.*

The `ls` command can give you all sorts of information.

27.1.1.3 `man`

Exercise 27.4. Read the manual seciton, or 'man page' of the `ls` command: `man ls`. Find out the size and the time / date of the last change to some files, for instance the file you just created.

Intended outcome. Did you find the `ls -s` and `ls -l` options? The first one lists the size of each file, usually in kilobytes, the other gives all sorts of information about a file, including things you will learn about later.

Things to watch out for. The `man` command puts you in a mode where you can view long text documents. This viewer is common on Unix systems (it is available as the `more` or `less` system command), so memorize the following ways of navigating: Use the space bar to go forward and the `u` key to go back up. Use `g` to go to the beginning fo the text, and `G` for the end. Use `q` to exit the viewer. If you really get stuck, `Control-c` will get you out.

Remark 20 *There are several dates associated with a file, corresponding to changes in content, changes in permissions, and access of any sort. The* `stat` *command gives all of them.*

Remark 21 *If you already know what command you're looking for, you can use man to get online information about it. If you forget the name of a command,* `man -k keyword` *can help you find it.*

27.1.1.4 `touch`

The `touch` command creates an empty file, or updates the timestamp of a file if it already exists. Use `ls -l` to confirm this behavior.

27.1.1.5 `cp, mv, rm`

The `cp` can be used for copying a file (or directories, see below): `cp file1 file2` makes a copy of `file1` and names it `file2`.

Exercise 27.5. Use `cp file1 file2` to copy a file. Confirm that the two files have the same contents. If you change the original, does anything happen to the copy?

Intended outcome. You should see that the copy does not change if the original changes or is deleted.

Things to watch out for. If `file2` already exists, you will get an error message.

A file can be renamed with `mv`, for 'move'.

Exercise 27.6. Rename a file. What happens if the target name already exists?

Files are deleted with `rm`. This command is dangerous: there is no undo.

27.1.1.6 `head, tail`

There are more commands for displaying a file, parts of a file, or information about a file.

Exercise 27.7. Do `ls /usr/share/words` or `ls /usr/share/dict/words` to confirm that a file with words exists on your system. Now experiment with the commands `head`, `tail`, `more`, and `wc` using that file.

Intended outcome. *`head`* displays the first couple of lines of a file, *`tail`* the last, and *`more`* uses the same viewer that is used for man pages. Read the man pages for these commands and experiment with increasing and decreasing the amount of output. The *`wc`* ('word count') command reports the number of words, characters, and lines in a file.

Another useful command is *`file`*: it tells you what type of file you are dealing with.

Exercise 27.8. Do `file foo` for various 'foo': a text file, a directory, or the `/bin/ls` command.

Intended outcome. Some of the information may not be intelligible to you, but the words to look out for are 'text', 'directory', or 'executable'.

At this point it is advisable to learn to use a text *editor*, such as *emacs* or *vi*.

27.1.2 Directories

Purpose. Here you will learn about the Unix directory tree, how to manipulate it and how to move around in it.

command	function
`ls`	list the contents of directories
`mkdir`	make new directory
`cd`	change directory
`pwd`	display present working directory

A unix file system is a tree of directories, where a directory is a container for files or more directories. We will display directories as follows:

```
/ ........................................ The root of the directory tree
├── bin ...................................... Binary programs
└── home ............................... Location of users directories
```

The root of the Unix directory tree is indicated with a slash. Do `ls /` to see what the files and directories there are in the root. Note that the root is not the location where you start when you reboot your personal machine, or when you log in to a server.

Exercise 27.9. The command to find out your current working directory is *`pwd`*. Your home directory is your working directory immediately when you log in. Find out your home directory.

Intended outcome. You will typically see something like `/home/yourname` or `/Users/yourname`. This is system dependent.

Do `ls` to see the contents of the working directory. In the displays in this section, directory names will be followed by a slash: `dir/` but this character is not part of their name. You can get this output by using `ls -F`, and you can tell your shell to use this output consistently by stating `alias ls=ls -F` at the start of your session. Example:

```
/home/you/
 |__ adirectory/
 |__ afile
```

The command for making a new directory is *`mkdir`*.

Exercise 27.10. Make a new directory with *`mkdir`* `newdir` and view the current directory with `ls`.

Intended outcome. You should see this structure:

```
/home/you/
 |__ newdir/ ........................................ the new directory
```

The command for going into another directory, that is, making it your working directory, is *`cd`* ('change directory'). It can be used in the following ways:

- `cd` Without any arguments, `cd` takes you to your home directory.
- `cd <absolute path>` An absolute path starts at the root of the directory tree, that is, starts with `/`. The `cd` command takes you to that location.
- `cd <relative path>` A relative path is one that does not start at the root. This form of the `cd` command takes you to `<yourcurrentdir>/<relative path>`.

Exercise 27.11. Do `cd newdir` and find out where you are in the directory tree with `pwd`. Confirm with `ls` that the directory is empty. How would you get to this location using an absolute path?

Intended outcome. `pwd` should tell you `/home/you/newdir`, and `ls` then has no output, meaning there is nothing to list. The absolute path is `/home/you/newdir`.

Exercise 27.12. Let's quickly create a file in this directory: *`touch`* `onefile`, and another directory: `mkdir otherdir`. Do `ls` and confirm that there are a new file and directory.

Intended outcome. You should now have:

```
/home/you/
 |__ newdir/ ............................................ you are here
      |__ onefile
      |__ otherdir/
```

The `ls` command has a very useful option: with `ls -a` you see your regular files and hidden files, which have a name that starts with a dot. Doing `ls -a` in your new directory should tell you that there are the following files:

```
/home/you/
 |__ newdir/ ............................................ you are here
      |__ .
      |__ ..
      |__ onefile
      |__ otherdir/
```

The single dot is the current directory, and the double dot is the directory one level back.

Exercise 27.13. Predict where you will be after `cd ./otherdir/..` and check to see if you were right.

> *Intended outcome.* The single dot sends you to the current directory, so that does not change anything. The `otherdir` part makes that subdirectory your current working directory. Finally, `..` goes one level back. In other words, this command puts your right back where you started.

Since your home directory is a special place, there are shortcuts for `cd`'ing to it: `cd` without arguments, `cd ~`, and `cd $HOME` all get you back to your home.

Go to your home directory, and from there do `ls newdir` to check the contents of the first directory you created, without having to go there.

Exercise 27.14. What does `ls ..` do?

> *Intended outcome.* Recall that `..` denotes the directory one level up in the tree: you should see your own home directory, plus the directories of any other users.

Exercise 27.15. Can you use `ls` to see the contents of someone else's home directory? In the previous exercise you saw whether other users exist on your system. If so, do `ls ../thatotheruser`.

> *Intended outcome.* If this is your private computer, you can probably view the contents of the other user's directory. If this is a university computer or so, the other directory may very well be protected – permissions are discussed in the next section – and you get `ls: ../otheruser: Permission denied`.

Make an attempt to move into someone else's home directory with `cd`. Does it work?

You can make copies of a directory with `cp`, but you need to add a flag to indicate that you recursively copy the contents: `cp -r`. Make another directory `somedir` in your home so that you have

```
/home/you/
 |__newdir/ ............................. you have been working in this one
 |__somedir/ ................................... you just created this one
```

What is the difference between `cp -r newdir somedir` and `cp -r newdir thirddir` where `thirddir` is not an existing directory name?

27.1.3 Permissions

> **Purpose.** In this section you will learn about how to give various users on your system permission to do (or not to do) various things with your files.

Unix files, including directories, have permissions, indicating 'who can do what with this file'. Actions that can be performed on a file fall into three categories:

- reading `r`: any access to a file (displaying, getting information on it) that does not change the file;
- writing `w`: access to a file that changes its content, or even its metadata such as 'date modified';
- executing `x`: if the file is executable, to run it; if it is a directory, to enter it.

The people who can potentially access a file are divided into three classes too:

- the user `u`: the person owning the file;

- the group g: a group of users to which the owner belongs;
- other o: everyone else.

These nine permissions are rendered in sequence

user	*group*	*other*
rwx	*rwx*	*rwx*

For instance `rw-r--r--` means that the owner can read and write a file, the owner's group and everyone else can only read.

Permissions are also rendered numerically in groups of three bits, by letting `r` = 4, `w` = 2, `x` = 1:

rwx
421

Common codes are 7 = `rwx` and 6 = `rw`. You will find many files that have permissions 755 which stands for an executable that everyone can run, but only the owner can change, or 644 which stands for a data file that everyone can see but again only the owner can alter. You can set permissions by the *chmod* command:

```
chmod <permissions> file         # just one file
chmod -R <permissions> directory # directory, recursively
```

Examples:

```
chmod 766 file  # set to rwxrw-rw-
chmod g+w file  # give group write permission
chmod g=rx file # set group permissions
chod o-w  file  # take away write permission from others
chmod o=  file  # take away all permissions from others.
chmod g+r,o-x file # give group read permission
                   # remove other execute permission
```

The man page gives all options.

Exercise 27.16. Make a file `foo` and do `chmod u-r foo`. Can you now inspect its contents? Make the file readable again, this time using a numeric code. Now make the file readable to your classmates. Check by having one of them read the contents.

Intended outcome. 1. A file is only accessible by others if the surrounding folder is readable. Can you figure out how to do this? 2. When you've made the file 'unreadable' by yourself, you can still `ls` it, but not `cat` it: that will give a 'permission denied' message.

Make a file `com` with the following contents:

```
#!/bin/sh
echo "Hello world!"
```

This is a legitimate shell script. What happens when you type `./com`? Can you make the script executable?

In the three permission categories it is clear who 'you' and 'others' refer to. How about 'group'? We'll go into that in section 27.12.

Remark 22 *There are more obscure permissions. For instance the* setuid *bit declares that the program should run with the permissions of the creator, rather than the user executing it. This is useful for system utilities such* `passwd` *or* `mkdir`*, which alter the password file and the directory structure, for which* root privileges *are needed. Thanks to the setuid bit, a user can run these programs, which are then so designed that a user can only make changes to their own password entry, and their own directories, respectively. The setuid bit is set with* `chmod`*:* `chmod 4ugo file`.

27.1.4 Wildcards

You already saw that `ls filename` gives you information about that one file, and `ls` gives you all files in the current directory. To see files with certain conditions on their names, the *wildcard* mechanism exists. The following wildcards exist:

`*`	any number of characters
`?`	any character.

Example:

```
%% ls
s       sk      ski     skiing  skill
%% ls ski*
ski     skiing  skill
```

The second option lists all files whose name start with `ski`, followed by any number of other characters'; below you will see that in different contexts `ski*` means 'sk followed by any number of `i` characters'. Confusing, but that's the way it is.

27.2 Text searching and regular expressions

Purpose. In this section you will learn how to search for text in files.

For this section you need at least one file that contains some amount of text. You can for instance get random text from `http://www.lipsum.com/feed/html`.

The *grep* command can be used to search for a text expression in a file.

Exercise 27.17. Search for the letter `q` in your text file with `grep q yourfile` and search for it in all files in your directory with `grep q *`. Try some other searches.

Intended outcome. In the first case, you get a listing of all lines that contain a `q`; in the second case, `grep` also reports what file name the match was found in: `qfile:this line has q in it`.

Things to watch out for. If the string you are looking for does not occur, `grep` will simply not output anything. Remember that this is standard behavior for Unix commands if there is nothing to report.

In addition to searching for literal strings, you can look for more general expressions.

`^`	the beginning of the line
`$`	the end of the line
`.`	any character
`*`	any number of repetitions
`[xyz]`	any of the characters xyz

This looks like the wildcard mechanism you just saw (section 27.1.4) but it's subtly different. Compare the example above with:

```
%% cat s
sk
ski
skill
skiing
%% grep "ski*" s
sk
ski
skill
skiing
```

In the second case you search for a string consisting of `sk` and any number of `i` characters, including zero of them.

Some more examples: you can find

- All lines that contain the letter 'q' with `grep q yourfile`;
- All lines that start with an 'a' with `grep "^a" yourfile` (if your search string contains special characters, it is a good idea to use quote marks to enclose it);
- All lines that end with a digit with `grep "[0-9]$" yourfile`.

Exercise 27.18. Construct the search strings for finding

- lines that start with an uppercase character, and
- lines that contain exactly one character.

Intended outcome. For the first, use the range characters `[]`, for the second use the period to match any character.

Exercise 27.19. Add a few lines `x = 1`, `x   = 2`, `x     = 3` (that is, have different numbers of spaces between `x` and the equals sign) to your test file, and make `grep` commands to search for all assignments to `x`.

The characters in the table above have special meanings. If you want to search that actual character, you have to *escape* it.

Exercise 27.20. Make a test file that has both `abc` and `a.c` in it, on separate lines. Try the commands `grep "a.c" file`, `grep a\.c file`, `grep "a\.c" file`.

Intended outcome. You will see that the period needs to be escaped, and the search string needs to be quoted. In the absence of either, you will see that `grep` also finds the `abc` string.

27.2.1 Cutting up lines with cut

Another tool for editing lines is *cut*, which will cut up a line and display certain parts of it. For instance,

```
cut -c 2-5 myfile
```

will display the characters in position 2–5 of every line of `myfile`. Make a test file and verify this example.

Maybe more useful, you can give `cut` a delimiter character and have it split a line on occurrences of that delimiter. For instance, your system will mostly likely have a file `/etc/passwd` that contains user information[1], with every line consisting of fields separated by colons. For instance:

```
daemon:*:1:1:System Services:/var/root:/usr/bin/false
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
```

The seventh and last field is the login shell of the user; `/bin/false` indicates that the user is unable to log in.

You can display users and their login shells with:

```
cut -d ":" -f 1,7 /etc/passwd
```

This tells `cut` to use the colon as delimiter, and to print fields 1 and 7.

27.3 Other useful commands: tar

The *tar* command stands for 'tape archive', that is, it was originally meant to package files on a tape. (The 'archive' part derives from the *ar* command.) These days, it's used to package files together for distribution on web sites and such: if you want to publish a library of hundreds of files this bundles them into a single file.

The two most common options are for

1. creating a tar file:
   ```
   tar fc package.tar directory_with_stuff
   ```
   pronounced 'tar file create', and
2. unpacking a tar file:
   ```
   tar fx package.tar
   # this creates the directory that was packaged
   ```
   pronounced 'tar file extract'.

Text files can often be compressed to a large extent, so adding the `z` compressiong for *gzip* is a good idea:

1. This is traditionally the case; on Mac OS information about users is kept elsewhere and this file only contains system services.

```
tar fcz package.tar.gz directory_with_stuff
tar fx package.tar.gz
```

Naming the ‘gzipped’ file `package.tgz` is also common.

27.4 Command execution

27.4.1 Search paths

Purpose. In this section you will learn how Unix determines what to do when you type a command name.

If you type a command such as `ls`, the shell does not just rely on a list of commands: it will actually go searching for a program by the name `ls`. This means that you can have multiple different commands with the same name, and which one gets executed depends on which one is found first.

Exercise 27.21. What you may think of as ‘Unix commands’ are often just executable files in a system directory. Do *`which`* `ls`, and do an `ls -l` on the result.

Intended outcome. The location of `ls` is something like `/bin/ls`. If you `ls` that, you will see that it is probably owned by root. Its executable bits are probably set for all users.

The locations where unix searches for commands is the *search path*, which is stored in the *environment variable* (for more details see below) *`PATH`*.

Exercise 27.22. Do `echo $PATH`. Can you find the location of `cd`? Are there other commands in the same location? Is the current directory ‘`.`’ in the path? If not, do `export PATH=".:$PATH"`. Now create an executable file `cd` in the current director (see above for the basics), and do `cd`.

Intended outcome. The path will be a list of colon-separated directories, for instance `/usr/bin:/usr/local/bin:/usr/X11R6/bin`. If the working directory is in the path, it will probably be at the end: `/usr/X11R6/bin:.` but most likely it will not be there. If you put ‘`.`’ at the start of the path, unix will find the local `cd` command before the system one.

Some people consider having the working directory in the path a security risk. If your directory is writable, someone could put a malicious script named `cd` (or any other system command) in your directory, and you would execute it unwittingly.

It is possible to define your own commands as aliases of existing commands.

Exercise 27.23. Do `alias chdir=cd` and convince yourself that now `chdir` works just like `cd`. Do `alias rm='rm -i'`; look up the meaning of this in the man pages. Some people find this alias a good idea; can you see why?

Intended outcome. The `-i` ‘interactive’ option for `rm` makes the command ask for confirmation before each delete. Since unix does not have a trashcan that needs to be emptied explicitly (as on Windows or the Mac OS), this can be a good idea.

27.4.2 Command sequencing

There are various ways of having multiple commands on a single commandline.

27.4.2.1 Simple sequencing

First of all, you can type

```
command1 ; command2
```

This is convenient if you repeat the same two commands a number of times: you only need to up-arrow once to repeat them both.

There is a problem: if you type

```
cc -o myprog myprog.c ; ./myprog
```

and the compilation fails, the program will still be executed, using an old version of the executable if that exists. This is very confusing.

A better way is:

```
cc -o myprog myprog.c && ./myprog
```

which only executes the second command if the first one was successful.

27.4.2.2 Pipelining

Instead of taking input from a file, or sending output to a file, it is possible to connect two commands together, so that the second takes the output of the first as input. The syntax for this is `cmdone | cmdtwo`; this is called a pipeline. For instance, `grep a yourfile | grep b` finds all lines that contains both an `a` and a `b`.

Exercise 27.24. Construct a pipeline that counts how many lines there are in your file that contain the string `th`. Use the `wc` command (see above) to do the counting.

27.4.2.3 Backquoting

There are a few more ways to combine commands. Suppose you want to present the result of `wc` a bit nicely. Type the following command

```
echo The line count is wc -l foo
```

where `foo` is the name of an existing file. The way to get the actual line count echoed is by the *backquote*:

```
echo The line count is `wc -l foo`
```

Anything in between backquotes is executed before the rest of the command line is evaluated.

Exercise 27.25. The way `wc` is used here, it prints the file name. Can you find a way to prevent that from happening?

There is another mechanism for out-of-order evaluation:

```
echo "There are $( cat Makefile | wc -l ) lines"
```

This mechanism makes it possible to nest commands, but for compatibility and legacy purposes backquotes may still be preferable when nesting is not neeeded.

27.4.2.4 Grouping in a subshell

Suppose you want to apply output redirection to a couple of commands in a row:

```
configure ; make ; make install > installation.log 2>&1
```

This only catches the last command. You could for instance group the three commands in a subshell and catch the output of that:

```
( configure ; make ; make install ) > installation.log 2>&1
```

27.4.3 Exit status

Commands can fail. If you type a single command on the command line, you see the error, and you act accordingly when you type the next command. When that failing command happens in a script, you have to tell the script how to act accordingly. For this, you use the *exit status* of the command: this is a value (zero for success, nonzero otherwise) that is stored in an internal variable, and that you can access with `$?`.

Example. Suppose we have a directory that is not writable

```
[testing] ls -ld nowrite/
dr-xr-xr-x  2 eijkhout  506  68 May 19 12:32 nowrite//
[testing] cd nowrite/
```

and write try to create a file there:

```
[nowrite] cat ../newfile
#!/bin/bash
touch $1
echo "Created file: $1"
[nowrite] newfile myfile
bash: newfile: command not found
[nowrite] ../newfile myfile
touch: myfile: Permission denied
Created file: myfile
[nowrite] ls
[nowrite]
```

The script reports that the file was created even though it wasn't.

Improved script:

```
[nowrite] cat ../betterfile
#!/bin/bash
touch $1
if [ $? -eq 0 ] ; then
    echo "Created file: $1"
else
    echo "Problem creating file: $1"
fi

[nowrite] ../betterfile myfile
touch: myfile: Permission denied
Problem creating file: myfile
```

27.4.4 Processes and jobs

`ps`	list (all) processes
`kill`	kill a process
`CTRL-c`	kill the foreground job
`CTRL-z`	suspect the foreground job
`jobs`	give the status of all jobs
`fg`	bring the last suspended job to the foreground
`fg %3`	bring a specific job to the foreground
`bg`	run the last suspended job in the background

The Unix operating system can run many programs at the same time, by rotating through the list and giving each only a fraction of a second to run each time. The command *ps* can tell you everything that is currently running.

Exercise 27.26. Type `ps`. How many programs are currently running? By default `ps` gives you only programs that you explicitly started. Do `ps guwax` for a detailed list of everything that is running. How many programs are running? How many belong to the root user, how many to you?

Intended outcome. To count the programs belonging to a user, pipe the `ps` command through an appropriate `grep`, which can then be piped to `wc`.

In this long listing of `ps`, the second column contains the *process numbers*. Sometimes it is useful to have those: if a program misbehaves you can *kill* it with

```
kill 123456
```

where `12345` is the process number.

The `cut` command explained above can cut certain position from a line: type `ps guwax | cut -c 10-14`.

To get dynamic information about all running processes, use the `top` command. Read the man page to find out how to sort the output by CPU usage.

Processes that are started in a shell are known as *jobsjob (unix)*. In addition to the process number, they have a job number. We will now explore manipulating jobs.

When you type a command and hit return, that command becomes, for the duration of its run, the *foreground process*. Everything else that is running at the same time is a *background process*.

Make an executable file `hello` with the following contents:

```
#!/bin/sh
while [ 1 ] ; do
  sleep 2
  date
done
```

and type `./hello`.

Exercise 27.27. Type `Control-z`. This suspends the foreground process. It will give you a number like `[1]` or `[2]` indicating that it is the first or second program that has been suspended or put in the background. Now type `bg` to put this process in the background. Confirm that there is no foreground process by hitting return, and doing an `ls`.

Intended outcome. After you put a process in the background, the terminal is available again to accept foreground commands. If you hit return, you should see the command prompt. However, the background process still keeps generating output.

Exercise 27.28. Type `jobs` to see the processes in the current session. If the process you just put in the background was number 1, type `fg %1`. Confirm that it is a foreground process again.

Intended outcome. If a shell is executing a program in the foreground, it will not accept command input, so hitting return should only produce blank lines.

Exercise 27.29. When you have made the `hello` script a foreground process again, you can kill it with `Control-c`. Try this. Start the script up again, this time as `./hello &` which immediately puts it in the background. You should also get output along the lines of `[1] 12345` which tells you that it is the first job you put in the background, and that `12345` is its process ID. Kill the script with `kill %1`. Start it up again, and kill it by using the process number.

Intended outcome. The command `kill 12345` using the process number is usually enough to kill a running program. Sometimes it is necessary to use `kill -9 12345`.

27.4.5 Shell customization

Above it was mentioned that `ls -F` is an easy way to see which files are regular, executable, or directories; by typing `alias ls='ls -F'` the `ls` command will automatically expanded to `ls -F` every time it is invoked. If you would like this behavior in every login session, you can add the `alias` command to your `.profile` file. Other shells than `sh`/`bash` have other files for such customizations.

27.5 Input/output Redirection

Purpose. In this section you will learn how to feed one command into another, and how to connect commands to input and output files.

So far, the unix commands you have used have taken their input from your keyboard, or from a file named on the command line; their output went to your screen. There are other possibilities for providing input from a file, or for storing the output in a file.

27.5.1 Input redirection

The `grep` command had two arguments, the second being a file name. You can also write `grep string < yourfile`, where the less-than sign means that the input will come from the named file, `yourfile`. This is known as *input redirection*.

27.5.2 Standard files

Unix has three standard files that handle input and output:

Standard file	Purpose
`stdin`	is the file that provides input for processes.
`stdout`	is the file where the output of a process is written.
`stderr`	is the file where error output is written.

In an interactive session, all three files are connected to the user terminal. Using input or output redirection then means that the input is taken or the output sent to a different file than the terminal.

27.5.3 Output redirection

Just as with the input, you can redirect the output of your program. In the simplest case, `grep string yourfile > outfile` will take what normally goes to the terminal, and *redirect* the output to `outfile`. The output file is created if it didn't already exist, otherwise it is overwritten. (To append, use `grep text yourfile >> outfile`.)

Exercise 27.30. Take one of the grep commands from the previous section, and send its output to a file. Check that the contents of the file are identical to what appeared on your screen before. Search for a string that does not appear in the file and send the output to a file. What does this mean for the output file?

Intended outcome. Searching for a string that does not occur in a file gives no terminal output. If you redirect the output of this `grep` to a file, it gives a zero size file. Check this with `ls` and `wc`.

Sometimes you want to run a program, but ignore the output. For that, you can redirect your output to the system *null device*: `/dev/null`.

```
yourprogram >/dev/null
```

Here are some useful idioms:

Idiom	Meaning
`program 2>/dev/null`	send only errors to the null device
`program >/dev/null 2>&1`	send output to dev-null, and errors to output Note the counterintuitive sequence of specifications!
`program 2>&1 | less`	send output and errors to `less`

27.6 Shell environment variables

Above you encountered `PATH`, which is an example of an shell, or environment, variable. These are variables that are known to the shell and that can be used by all programs run by the shell. You can see the full list of all variables known to the shell by typing *`env`*.

You can get the value of a shell variable by prefixing it with a dollar sign. Type the following two commands and compare the output:

```
echo PATH
echo $PATH
```

Exercise 27.31. Check on the value of the `HOME` variable by typing `echo $HOME`. Also find the value of `HOME` by piping `env` through `grep`.

Environment variables can be set in a number of ways. The simplest is by an assignment as in other programming languages.

Exercise 27.32. Type `a=5` on the commandline. This defines a variable `a`; check on its value by using the `echo` command.

Intended outcome. The shell will respond by typing the value `5`.

Things to watch out for. Beware not to have space around the equals sign; also be sure to use the dollar sign to print the value.

A variable set this way will be known to all subsequent commands you issue in this shell, but not to commands in new shells you start up. For that you need the *`export`* command. Reproduce the following session (the square brackets form the command prompt):

```
[] a=20
[] echo $a
20
[] /bin/bash
[] echo $a
```

```
[] exit
exit
[] export a=21
[] /bin/bash
[] echo $a
21
[] exit
```

You can also temporarily set a variable. Replay this scenario:

1. Find an environment variable that does not have a value:
   ```
   [] echo $b

   []
   ```
2. Write a short shell script to print this variable:
   ```
   [] cat > echob
   #!/bin/bash
   echo $b
   ```
   and of course make it executable: `chmod +x echob`.
3. Now call the script, preceding it with a setting of the variable `b`:
   ```
   [] b=5 ./echob
   5
   ```
   The syntax where you set the value, as a prefix without using a separate command, sets the value just for that one command.
4. Show that the variable is still undefined:
   ```
   [] echo $b

   []
   ```
   That is, you defined the variable just for the execution of a single command.

In section 27.7 you will see that the `for` construct also defines a variable; section 27.8.1 shows some more built-in variables that apply in shell scripts.

If you want to un-set an environment variable, there is the *`unset`* command.

27.7 Control structures

Like any good programming system, the shell has some control structures. Their syntax takes a bit of getting used to. (Different shells have different syntax; in this tutorial we only discuss the bash shell.

27.7.1 Conditionals

The *conditional* of the bash shell is predictably called `if`, and it can be written over several lines:

```
if [ $PATH = "" ] ; then
  echo "Error: path is empty"
fi
```

or on a single line:

```
if [ `wc -l file` -gt 100 ] ; then echo "file too long" ; fi
```

(The backquote is explain in section 27.4.2.3.) There are a number of tests defined, for instance `-f somefile` tests for the existence of a file. Change your script so that it will report `-1` if the file does not exist.

The syntax of this is finicky:

- `if` and `elif` are followed by a conditional, followed by a semicolon.
- The brackets of the conditional need to have spaces surrounding them.
- There is no semicolon after `then` of `else`.

Exercise 27.33. Bash conditionals have an `elif` keyword. Still you can write the sequence `else if`, as in:

```
if [ something ] ; then
  foo
else if [ something_else ] ; then
  bar
fi
```

Can you predict what the error is here?

27.7.2 Looping

There are also loops. A `for` loop looks like

```
for var in listofitems ; do
  something with $var
done
```

This does the following:

- for each item in `listofitems`, the variable `var` is set to the item, and
- the loop body is executed.

As a simple example:

```
[] for x in a b c ; do echo $x ; done
a
b
c
```

In a more meaningful example, here is how you would make backups of all your `.c` files:

```
for cfile in *.c ; do
  cp $cfile $cfile.bak
done
```

Shell variables can be manipulated in a number of ways. Execute the following commands to see that you can remove trailing characters from a variable:

```
[] a=b.c
[] echo ${a%.c}
b
```

(See the section 27.9 on expansion.) With this as a hint, write a loop that renames all your `.c` files to `.x` files.

The above construct loops over words, such as the output of `ls`. To do a numeric loop, use the command *seq*:

```
[shell:474] seq 1 5
1
2
3
4
5
```

Looping over a sequence of numbers then typically looks like

```
for i in `seq 1 ${HOWMANY}` ; do echo $i ; done
```

Note the *backtick*, which is necessary to have the `seq` command executed before evaluating the loop.

27.8 Scripting

The unix shells are also programming environments. You will learn more about this aspect of unix in this section.

27.8.1 How to execute scripts

It is possible to write programs of unix shell commands. First you need to know how to put a program in a file and have it be executed. Make a file `script1` containing the following two lines:

```
#!/bin/bash
echo "hello world"
```

and type `./script1` on the command line. Result? Make the file executable and try again.

In order write scripts that you want to invoke from anywhere, people typically put them in a directory `bin` in their home directory. You would then add this directory to your *search path*, contained in `PATH`; see section 27.4.1.

27.8.2 Script arguments

You can invoke a shell script with options and arguments:

```
./my_script -a file1 -t -x file2 file3
```

You will now learn how to incorporate this functionality in your scripts.

First of all, all commandline arguments and options are available as variables `$1`,`$2` et cetera in the script, and the number of command line arguments is available as `$#`:

```
#!/bin/bash

echo "The first argument is $1"
echo "There were $# arguments in all"
```

Formally:

variable	meaning
`$#`	number of arguments
`$0`	the name of the script
`$1,$2,...`	the arguments
`$*,$@`	the list of all arguments

Exercise 27.34. Write a script that takes as input a file name argument, and reports how many lines are in that file.

Edit your script to test whether the file has less than 10 lines (use the `foo -lt bar` test), and if it does, `cat` the file. Hint: you need to use backquotes inside the test.

Add a test to your script so that it will give a helpful message if you call it without any arguments.

The standard way to parse argument is using the *`shift`* command, which pops the first argument off the list of arguments. Parsing the arguments in sequence then involves looking at `$1`, shifting, and looking at the new `$1`.

Code:

```
// arguments.sh
while [ $# -gt 0 ] ; do
    echo "argument: $1"
    shift
done
```

Output
[code/shell] arguments:

Exercise 27.35. Write a script `say.sh` that prints its text argument. However, if you invoke it with

```
./say.sh -n 7 "Hello world"
```

it should be print it as many times as you indicated. Using the option `-u`:

```
./say.sh -u -n 7 "Goodbye cruel world"
```

should print the message in uppercase. Make sure that the order of the arguments does not matter, and give an error message for any unrecognized option.

The variables `$@` and `$*` have a different behavior with respect to double quotes. Let's say we evaluate `myscript "1 2" 3`, then

- Using `$*` is the list of arguments after removing quotes: `myscript 1 2 3`.
- Using `"$*"` is the list of arguments, with quotes removed, in quotes: `myscript "1 2 3"`.
- Using `"$@"` preserved quotes: `myscript "1 2" 3`.

27.9 Expansion

The shell performs various kinds of expansion on a command line, that is, replacing part of the commandline with different text.

Brace expansion:

```
[] echo a{b,cc,ddd}e
abe acce addde
```

This can for instance be used to delete all extension of some base file name:

```
[] rm tmp.{c,s,o}  # delete tmp.c tmp.s tmp.o
```

Tilde expansion gives your own, or someone else's home directory:

```
[] echo ~
/share/home/00434/eijkhout
[] echo ~eijkhout
/share/home/00434/eijkhout
```

Parameter expansion gives the value of shell variables:

```
[] x=5
[] echo $x
5
```

Undefined variables do not give an error message:

```
[] echo $y
```

There are many variations on parameter expansion. Above you already saw that you can strip trailing characters:

```
[] a=b.c
[] echo ${a%.c}
b
```

Here is how you can deal with undefined variables:

```
[] echo ${y:-0}
0
```

The backquote mechanism (section 27.4.2.3 above) is known as command substitution. It allows you to evaluate part of a command and use it as input for another. For example, if you want to ask what type of file the command `ls` is, do

```
[] file `which ls`
```

This first evaluates `which ls`, giving `/bin/ls`, and then evaluates `file /bin/ls`. As another example, here we backquote a whole pipeline, and do a test on the result:

```
[] echo 123 > w
[] cat w
123
[] wc -c w
       4 w
[] if [ `cat w | wc -c` -eq 4 ] ; then echo four ; fi
four
```

Unix shell programming is very much oriented towards text manipulation, but it is possible to do arithmetic. Arithmetic substitution tells the shell to treat the expansion of a parameter as a number:

```
[] x=1
[] echo $((x*2))
2
```

Integer ranges can be used as follows:

```
[] for i in {1..10} ; do echo $i ; done
1
2
3
4
5
6
7
8
9
10
```

27.10 Startup files

In this tutorial you have seen several mechanisms for customizing the behavior of your shell. For instance, by setting the PATH variable you can extend the locations where the shell looks for executables. Other environment variables (section 27.6) you can introduce for your own purposes. Many of these customizations will need to apply to every sessions, so you can have *shell startup files* that will be read at the start of any session.

Popular things to do in a startup file are defining *alias*es:

```
alias grep='grep -i'
alias ls='ls -F'
```

and setting a custom commandline *prompt*.

Unfortunately, there are several startup files, and which one gets read is a complicated functions of circumstances. Here is a good common sense guideline[2]:

- Have a .profile that does nothing but read the .bashrc:
```
# ~/.profile
if [ -f ~/.bashrc ]; then
    source ~/.bashrc
fi
```
- Your .bashrc does the actual customizations:
```
# ~/.bashrc
# make sure your path is updated
if [ -z "$MYPATH" ]; then
  export MYPATH=1
  export PATH=$HOME/bin:$PATH
fi
```

27.11 Shell interaction

Interactive use of Unix, in contrast to script writing (section 27.8), is a complicated conversation between the user and the shell. You, the user, type a line, hit return, and the shell tries to interpret it. There are several cases.

- Your line contains one full command, such as `ls foo`: the shell will execute this command.
- You can put more than one command on a line, separated by semicolons: `mkdir foo; cd foo`. The shell will execute these commands in sequence.
- Your input line is not a full command, for instance `while [ 1]`. The shell will recognize that there is more to come, and use a different prompt to show you that it is waiting for the remainder of the command.

2. Many thanks to Robert McLay for figuring this out.

- Your input line would be a legitimate command, but you want to type more on a second line. In that case you can end your input line with a backslash character, and the shell will recognize that it needs to hold off on executing your command. In effect, the backslash will hide (*escape*) the return.

When the shell has collected a command line to execute, by using one or more of your input line or only part of one, as described just now, it will apply expansion to the command line (section 27.9). It will then interpret the commandline as a command and arguments, and proceed to invoke that command with the arguments as found.

There are some subtleties here. If you type `ls *.c`, then the shell will recognize the wildcard character and expand it to a command line, for instance `ls foo.c bar.c`. Then it will invoke the `ls` command with the argument list `foo.c bar.c`. Note that `ls` does not receive `*.c` as argument! In cases where you do want the unix command to receive an argument with a wildcard, you need to escape it so that the shell will not expand it. For instance, `find . -name \*.c` will make the shell invoke `find` with arguments `. -name *.c`.

27.12 The system and other users

Unix is a multi-user operating system. Thus, even if you use it on your own personal machine, you are a user with an *account* and you may occasionally have to type in your username and password.

If you are on your personal machine, you may be the only user logged in. On university machines or other servers, there will often be other users. Here are some commands relating to them.

`whoami` show your login name.

`who` show the other users currently logged in.

`finger` `otheruser` get information about another user; you can specify a user's login name here, or their real name, or other identifying information the system knows about.

`top` which processes are running on the system; use `top -u` to get this sorted the amount of cpu time they are currently taking. (On Linux, try also the `vmstat` command.)

`uptime` how long has it been since your last reboot?

27.12.1 Groups

In section 27.1.3 you saw that there is a permissions category for 'group'. This allows you to open up files to your close collaborators, while leaving them protected from the wide world.

When your account is created, your system administrator will have assigned you to one or more groups. (If you admin your own machine, you'll be in some default group; read on for adding yourself to more groups.)

The command *`groups`* tells you all the groups you are in, and `ls -l` tells you what group a file belongs to. Analogous to `chmod`, you can use *`chgrp`* to change the group to which a file belongs, to share it with a user who is also in that group.

Creating a new group, or adding a user to a group needs system privileges. To create a group:

```
sudo groupadd new_group_name
```

To add a user to a group:

```
sudo usermod -a -G thegroup theuser
```

27.12.2 The super user

Even if you own your machine, there are good reasons to work as much as possible from a regular user account, and use *root privileges* only when strictly needed. (The root account is also known as the *super user*.) If you have root privileges, you can also use that to 'become another user' and do things with their privileges, with the *sudo* ('superuser do') command.

- To execute a command as another user:
  ```
  sudo -u otheruser command arguments
  ```
- To execute a command as the root user:
  ```
  sudo command arguments
  ```
- Become another user:
  ```
  sudo su - otheruser
  ```
- Become the *super user*:
  ```
  sudo su -
  ```

27.13 Other systems: ssh and scp

No man is an island, and no computer is either. Sometimes you want to use one computer, for instance your laptop, to connect to another, for instance a supercomputer.

If you are already on a Unix computer, you can log into another with the 'secure shell' command *ssh*, a more secure variant of the old 'remote shell' command *rsh*:

```
ssh yourname@othermachine.otheruniversity.edu
```

where the `yourname` can be omitted if you have the same name on both machines.

To only copy a file from one machine to another you can use the 'secure copy' *scp*, a secure variant of 'remote copy' *rcp*. The `scp` command is much like `cp` in syntax, except that the source or destination can have a machine prefix.

To copy a file from the current machine to another, type:

```
scp localfile yourname@othercomputer:otherdirectory
```

where `yourname` can again be omitted, and `otherdirectory` can be an absolute path, or a path relative to your home directory:

```
# absolute path:
scp localfile yourname@othercomputer:/share/
# path relative to your home directory:
scp localfile yourname@othercomputer:mysubdirectory
```

Leaving the destination path empty puts the file in the remote home directory:

```
scp localfile yourname@othercomputer:
```

Note the colon at the end of this command: if you leave it out you get a local file with an 'at' in the name.

You can also copy a file from the remote machine. For instance, to copy a file, preserving the name:

```
scp yourname@othercomputer:otherdirectory/otherfile .
```

27.14 The `sed` and `awk` tools

Apart from fairly small utilities such as `tr` and `cut`, Unix has some more powerful tools. In this section you will see two tools for line-by-line transformations on text files. Of course this tutorial merely touches on the depth of these tools; for more information see [3, 55].

27.14.1 Stream editing with `sed`

Unix has various tools for processing text files on a line-by-line basis. The stream editor *sed* is one example. If you have used the `vi` editor, you are probably used to a syntax like `s/foo/bar/` for making changes. With `sed`, you can do this on the commandline. For instance

```
sed 's/foo/bar/' myfile > mynewfile
```

will apply the substitute command `s/foo/bar/` to every line of `myfile`. The output is shown on your screen so you should capture it in a new file; see section 27.5 for more on output *redirection*.

- If you have more than one edit, you can specify them with
  ```
  sed -e 's/one/two/' -e 's/three/four/'
  ```
- If an edit needs to be done only on certain lines, you can specify that by prefixing the edit with the match string. For instance
  ```
  sed '/^a/s/b/c/'
  ```
  only applies the edit on lines that start with an `a`. (See section 27.2 for regular expressions.)
- Traditionally, `sed` could only function in a stream, so the output file always had to be different from the input. The GNU version, which is standard on Linux systems, has a flag `-i` which edits 'in place':

```
sed -e 's/ab/cd/' -e 's/ef/gh/' -i thefile
```

27.14.2 awk

The *awk* utility also operates on each line, but it can be described as having a memory. An awk program consists of a sequence of pairs, where each pair consists of a match string and an action. The simplest awk program is

```
cat somefile | awk '{ print }'
```

where the match string is omitted, meaning that all lines match, and the action is to print the line. Awk breaks each line into fields separated by whitespace. A common application of awk is to print a certain field:

```
awk '{print $2}' file
```

prints the second field of each line.

Suppose you want to print all subroutines in a Fortran program; this can be accomplished with

```
awk '/subroutine/ {print}' yourfile.f
```

Exercise 27.36. Build a command pipeline that prints of each subroutine header only the subroutine name. For this you first use `sed` to replace the parentheses by spaces, then `awk` to print the subroutine name field.

Awk has variables with which it can remember things. For instance, instead of just printing the second field of every line, you can make a list of them and print that later:

```
cat myfile | awk 'BEGIN {v="Fields:"} {v=v " " $2} END {print v}'
```

As another example of the use of variables, here is how you would print all lines in between a `BEGIN` and `END` line:

```
cat myfile | awk '/END/ {p=0} p==1 {print} /BEGIN/ {p=1} '
```

Exercise 27.37. The placement of the match with `BEGIN` and `END` may seem strange. Rearrange the awk program, test it out, and explain the results you get.

27.15 Review questions

Exercise 27.38. Devise a pipeline that counts how many users are logged onto the system, whose name starts with a vowel and ends with a consonant.

Exercise 27.39. Pretend that you're a professor writing a script for homework submission: if a student invokes this script it copies the student file to some standard location.

```
submit_homework myfile.txt
```

For simplicity, we simulate this by making a directory `submissions` and two different files `student1.txt` and `student2.txt`. After

```
submit_homework student1.txt
submit_homework student2.txt
```

there should be copies of both files in the `submissions` directory. Start by writing a simple script; it should give a helpful message if you use it the wrong way.
Try to detect if a student is cheating. Explore the *`diff`* command to see if the submitted file is identical to something already submitted: loop over all submitted files and

1. First print out all differences.
2. Count the differences.
3. Test if this count is zero.

Now refine your test by catching if the cheating student randomly inserted some spaces.
For a harder test: try to detect whether the cheating student inserted newlines. This can not be done with *`diff`*, but you could try *`tr`* to remove the newlines.