Unix

Crash Course

This document is written as part of a course manual for the Unix course of the Graduate School at the Academic Medical Center in Amsterdam, the Netherlands.
Written by Rob Wolfram (rsw@hamal.nl), Unix systems administrator at the AMC.
This document is redistributable under the GNU Free Documentation License available via http://www.gnu.org/licenses/fdl.txt
The latest version of this document is available as both pdf and OpenDocument versions at http://crash.hamal.nl/.
This is the fifth edition of the document, dated March 2016. The first edition was dated April 2006.
The image on the title page originated from “The Unix-Haters Handbook” by Simson Garfinkel, Daniel Weise and Steven Strassman.

Command-line interaction is denoted in a non-proportional font like this:

```
$ echo foo

foo
```

The “dollar space” prefix denotes the “prompt”, i.e. where input from the user is expected.

Table of Contents

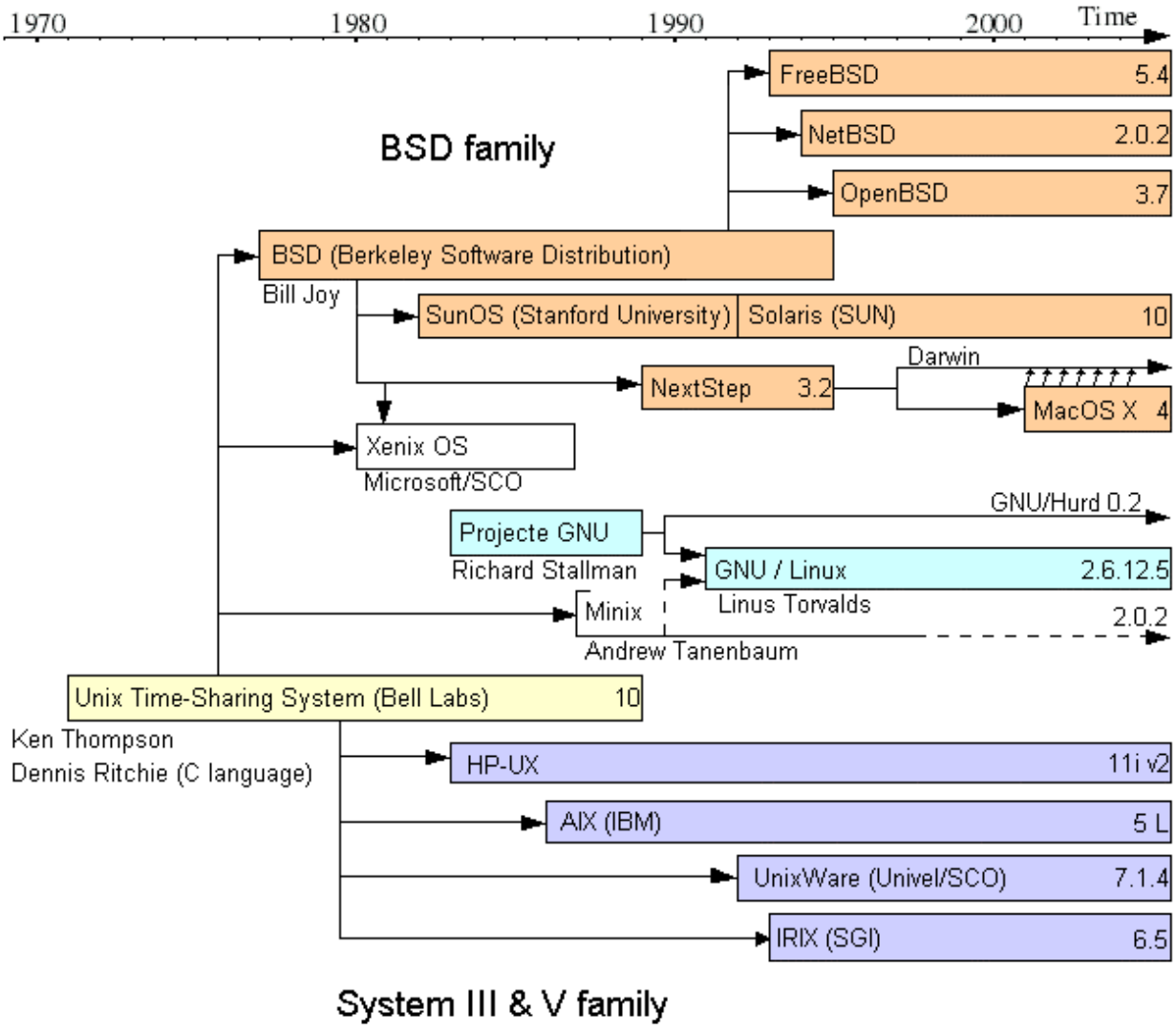
1 A brief history of Unix

In 1969 AT&T abandoned a project to develop an operating system named MULTICS (short for Multiplexed Information and Computing Service). One of the participants of that project was Ken Thompson and together with a colleague at Bell Labs (then part of AT&T) named Dennis Ritchie they created a simplified version of MULTICS with a twist. Another colleague (Brian Kernighan) punned the name Unics (Uniplexed i.o. Multiplexed) and shortly thereafter the name was changed to Unix. The purpose of the system was mainly for internal research use. Ritchie was also responsible for the programming language “C”, which was based on Ken Thompson's “B” which again was based on BCPL. Because of the minimalistic approach of C, the language was very suitable for system-level programming and soon Unix was ported from PDP assembly language (a.k.a. machine language) to C. This made the system highly portable and was the main cause of its popularity, especially in academic environments.

Dennis Ritchie (standing) and Ken Thompson working with Unix on a PDP11/20
Permission to use this picture kindly provided by Dennis Ritchie (Lucent Technologies)

One academic licensee of Unix, namely the Berkeley University in California, modified it and published their own version named BSD (Berkeley Software Distribution). This was the first of many specific versions of Unix, though most were either based primarily on BSD or the AT&T version of Unix, generally known as “System V”. Examples of Unix systems that are available today are IBM's AIX, Solaris from Sun Microsystems (now Oracle), Unixware from Novell, HPUX from Hewlett Packard and various BSD derivations like MacOS X from Apple. It would take until the late eighties before the first effort was made to unify the various Unix version on the system level. This effort was named POSIX (short for Portable Operating System Interface for uniX). This enabled a programmer to create a program for one system and only need a recompile to enable the program on another system.

One system demands special notice since it is not officially Unix, but its look and feel is indistinguishable from other Unices. This system is generally known as GNU/Linux or simply as Linux. This started in 1984 when Richard Stallman, member of the Artificial Intelligence team at MIT started a project to create an operating system named GNU that could be freely modified and distributed. He used Unix as a model of his system and started with the user-land programs. A Finnish Computer Science student named Linus Torvalds completed the project by writing a kernel for Intel-based computers that worked with the GNU programs. The system gained momentum and enabled people at home to use a Unix based system.



1.1 Linux distributions

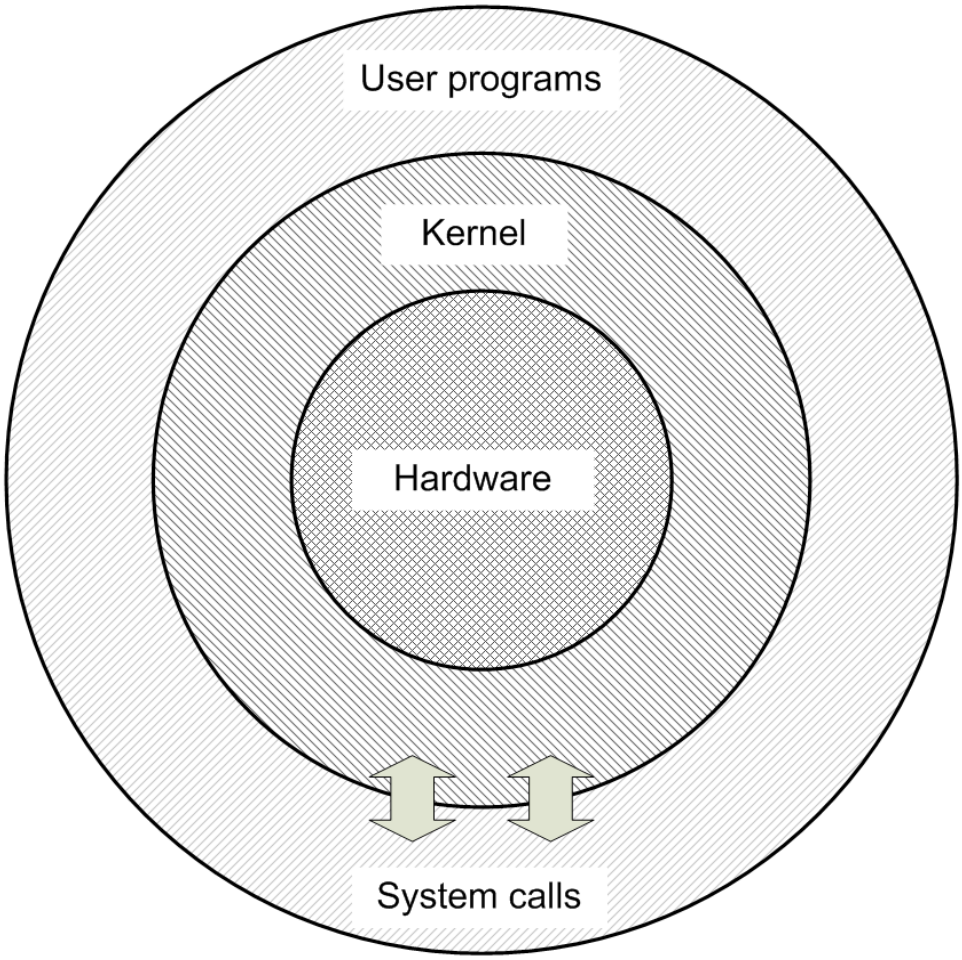
Both in private and business use, Linux (or GNU/Linux) has become the "*de facto*" standard for Unix use, ranging from being the embedded OS on small devices to very large multi-million dollar computing engines. But even for casual use, there is far from a standard of what the Linux system looks like. In Linux, there are so called "distributions". All distributions have the Linux kernel and the standard GNU user land, but they differ primarily on their software and configuration management and in the extra software packages that are included.

Some popular distributions are:

- Slackware. This is one of the first distributions with a “package system” and dates to the early nineties. The development community is not that large so many compromises are made regarding the availability of features.
- Debian. This is the first distribution with a modern software management system. It dates back to the early nineties and their focus is on free (a.k.a. open source) software and multiple hardware platforms. There are many derivative distributions, the most popular of which is Ubuntu, a distribution with a focus on ease of use for end users.
- Red Hat. This is world wide the most popular distribution for enterprise use. It started as a spin off from Slackware where they added modern software management, comparable to Debian's. For this there are also many derivative distributions like Fedora (the community version of Red Hat) and SuSE (another enterprise distribution now owned by Novell, mostly used in Europe).
- Gentoo. This is a more recent distribution (from the early naughts) where most software is supposed to be compiled from source on the destination system.

2 The structure of a Unix system

A Unix system has an onion-like ring structure where the hardware is completely managed by the kernel and the kernel provides an interface for user-land programs (named “system calls”) to access privileged resources. Besides accessing hardware the kernel is also responsible for scheduling processor time for various processes, privilege separation between various user programs, the file-system etc.



One of the main user programs is the *shell*. This is the program with which the user interacts with a Unix system. Most Unix systems also host a number of user programs named *daemons*. These programs are not interactive but run in the background and serve a special purpose. The name is derived from “Maxwell's demon” and has no religious connotations. Examples of daemons are web servers, task schedulers, mail transport agents, a database server program, login programs etc.

Unix is a multi-user operating system. This implies that programs from different users can run concurrently on the same system. This includes interactive programs, so multiple users can interact with the OS at the same time. A user is determined by the OS by her numeric *user id*, but the system provides a user name that is coupled to the user id to ease the recognition of the various users. There is one special user with user id 0 and generally named “root”. This user has access to various resources and files that normal users do not. E.g., when a system program is being installed this is usually done by root.

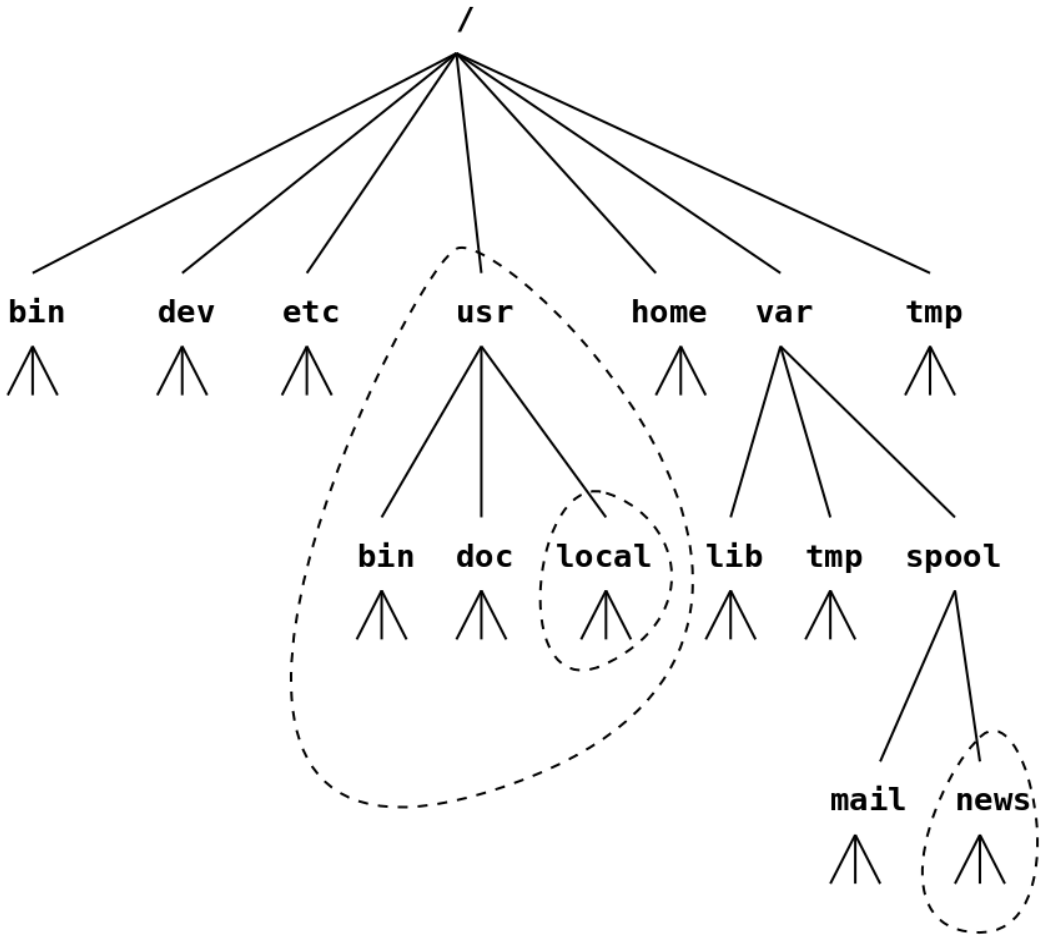
Aside from a user name (and corresponding user id) a user is also assigned to one or more groups, determined by the group name and corresponding numeric group id. Every user is assigned to at least one group. Groups enable collective permissions for certain resources. This is handled in the subject about file permissions below.

When a user wishes to interact with a Unix system she needs to authenticate herself to the system. In general this is done by providing the user name and a password to a login daemon on the system. If the login is successful, the user's shell is started and a TTY is assigned to the shell. The TTY is short for “TeleTYpe” and the name is a remnant of the first Unix systems where interaction happened by typing in commands and have the results printed on an attached printer. The TTY is necessary for the system to know where the response to commands must be sent to. This TTY can either be hardwired to some physical connection (like a “dumb terminal” that is connected to the system via a serial line) or it can be a logically assigned name when the user interacts with the system via a network connection. In the latter case we name it a “Pseudo-TTY” since no real hardwired device is used.

When a user interacts with the system, there is a myriad of small tools that can be started. The Unix philosophy is to have a toolbox with many small tools that can do very little (e.g. sort lines of text) but are extremely powerful in the small domain that they address. By combining the tools you can create many computing functionalities for specific tasks. This is treated more in-depth in section 4.2 about redirection and piping.

3 Files and file-systems

A Unix system has one single hierarchy of files and directories. Unlike e.g. DOS and Windows, Unix does not know “drive letters” or similar system. Devices (like hard disk partitions, CDRoms, Flash cards etc.) can contain a file-system. This is an organization of files and directories in a tree structure. The various file-systems are combined into a single tree by connecting the root node of every file-system to a directory of the tree that existed at that moment. This connecting is called “*mounting*” and the directory where the file-system is mounted is called the “*mount point*”. After a file-system has been mounted into the unified tree, its files can be accessed in the same manner as you access files in other directories of the file-system. The root node of the unified tree is denoted with a single forward slash (`/`) and is called “root”. The slash character is also the directory separator when denoting the complete path of a file (e.g. **`/usr/bin/ssh`**). The next figure gives an idea of what a Unix file-system might look like. The areas that are marked with a dashed line indicate separate file-systems.



Note that **`/usr/local`** is mounted on a file-system that is itself mounted on the “usr” file-system.

Though there are differences between the layout of the unified file-system on various Unix variants, there are some general guidelines of the purpose of various directories:

`/bin`: contains binaries that are used by all users
`/sbin`: contains binaries that are primarily used by the superuser (user root)
`/dev`: contains device files (explained later)
`/lib`: contains shared libraries for the programs
`/etc`: contains initialization and configuration files especially for daemons
`/home`: placeholder for the home directories of different users
`/var`: variable files (log files, mail spool etc)
`/tmp`: a directory that is writable by all users, suitable for temporary files
`/mnt`: serves as a mount point for temporarily mounted file-systems
`/usr`: a “mini unified tree” which contains binaries, libraries etc which are not needed during system maintenance mode and thus can be on a separate file-system. Most additional programs go into the **`/usr`** tree
`/usr/local`: a placeholder for non-standard and locally built software
`/usr/src`: a placeholder for software source code that is being built locally
`/usr/share`: a repository for documentation, dictionaries and the like

On Unix, files are used to access most resources. On most Unix file-systems a file is denoted by an *inode*. This is a block of “meta-data” on the file-system and contains the following items:

- An inode number
- The file type
- The file size
- The owner of the file
- The goup that has “group ownership”
- The file permissions
- A link count
- The (last) access time (atime)
- The modification time (mtime)
- The modification time of the inode (ctime)
- The used blocks (direct, indirect and double indirect pointers)

Note the glaring omission of a filename. More on that in a short while. As the list shows, a file is denoted by a number, but this number is not unique throughout the unified directory tree. It is only unique within a single file-system. The file type can either be an ordinary file, a directory, a symbolic link or a device file. Indeed, a directory is a file just like all others. Its contents is a table of filenames and the inode that the filename points to. There are always at least two entries present in each directory, one is denoted by a single dot (`.`) and refers to the inode of the current directory and the other is denoted by two dots (`..`) and refers to the inode of the parent directory. As you see here, a filename is just a label that points to an inode. There seems to be no hindrance for multiple labels to point to the same inode. Indeed there isn't. A file *can* have multiple filenames. This is called “*hard linking*”. In the inode a count is kept of all filenames that link to it. If a file is being “deleted”, in fact just the directory entry is removed and the link count in the inode is decremented. Only if the link count reaches zero will the blocks used by the file be freed. Technically it is possible to create a hard link to a directory (`.` and `..` are hard links) but administratively it is prevented because that could lead to ambiguities. Note that since a filename links to an inode, hard links (i.e. multiple filenames to the same inode) cannot cross the file-system boundary.

A Unix filename can contain almost any ASCII character. The only characters that are disallowed are the `/` (being a directory separator) and the null character (not to be confused with the character that represents the numeric 0, the null character is the character that denotes a string end in the C language). Though “control” characters such as a carriage return or a bell character are possible in filenames, their use is discouraged. Files on a Unix system don't require mandatory “filename extensions”. A dot is a valid filename character just like any other. Nonetheless using trailing characters in a filename comparable to filename extensions increase transparency and are quite common. Some commands consider filenames that start with a dot as “hidden”. This is not a feature of the filesystem, but of the individual command. Early Unix file-systems had a filename limit of 14 characters but most modern file-systems allow filenames up to 255 characters. Another important aspect to note is that filenames on a Unix system are “case sensitive” (as is most of Unix), which means that the filenames **`README`**, **`ReadMe`** and **`readme`** can denote three different files.

A symbolic link is a special file that is a pointer to a filename (optionally including a directory path). Note that this is a feature of the file-system, so a program can access a symbolic link just like it can any file and it will get the data in both cases. A symbolic link has the advantage that it can cross file-system boundaries and you can easily recognize what file it points to. On the other hand, given a file there is no direct way to determine if there is a symbolic link pointing to it, let alone which. The main disadvantage is that, since it's a simple pointer, there is no sanity check to determine if the file or directory pointed to really exists. If a symbolic link points to a non-existing file, we call it a “dangling link”.

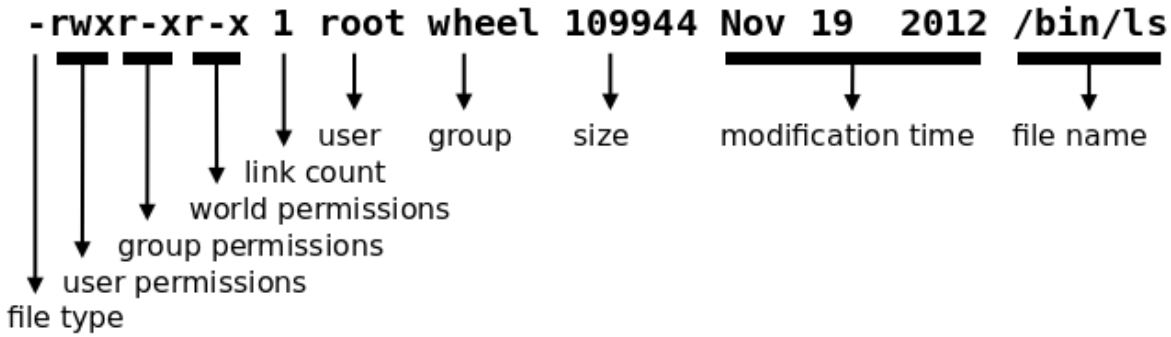
Device files are, as their name implies, files that can be used to access peripheral devices. There are two kind of devices, namely block devices files and character device files. They either access the devices in blocks of 512 bytes or single bytes. A device file has two numbers instead of the filesize, a *major* and a *minor* number. The major number defines which kernel driver will be accessed and the minor number is used by the kernel driver to distinguish between devices. E.g. two disk partitions will have the same major number but different minor numbers. Only the superuser can create device files but they can be used (depending on permissions) by ordinary users. The main advantage of this approach is that a program can access data on a device just like like it would with any other file. This keeps the programs simpler. One device file that is always present on Unix is **`/dev/null`**. All data that is sent to this device file is discarded.

There are also other kind of files like “named pipes” and “socket files”, but these are outside the scope of a crash course.

3.1 File permissions

All Unix-based systems have a permissions system based on access rights on the file that can be set or unset at any time. This is called a “DAC” system i.e., discretionary access controls. Modern Unix-based systems can often also have an extra mechanism that cannot be altered at any time. That is a “MAC” system (mandatory access controls). The working of such a system is outside the scope of this course. We will now discuss the standard permissions system, sometimes referred to as “ugo permissions”.

Of every file the ownership is determined by the “user” and “group” entries in the inode. Together with the file permissions this enables a pretty fine grained but still easily manageable access system. A file listing might look like the following:



Permissions are defined for 3 sets of users, namely the owner of the file, all users that belong to the group that holds the group ownership, and users that do not belong to the two previous sets. For every set a bit is set or cleared for read(**`r`**), write(**`w`**) and execute(**`x`**) permissions. It is also possible to represent the permissions with octal numbers, where “**`r`**” has a value of **`4`**, “**`w`**” has a value of **`2`** and “**`x`**” has a value of **`1`**. The sum of those define the permissions. E.g., the octal permissions “**`751`**” are equivalent to “**`rwxr-x--x`**”.

When considering ordinary files, reading and writing refer to accessing the content of the file itself. The execute bit denotes that the file is a program or script that can be run. The filename does not need to have a special format (like ending with *.exe*) to be executable.

On directories the permissions work the same way, if you keep in mind that a directory is a file just like any other. The read-bit denotes that you can view the content of a directory, i.e. you can see which files are in the directory, but you might not be able to access the files or query the permission bits of thefiles. For that, you will need to be able to enter into the directory and that's where the execute-bit comes in. It is possible to execute or modify a file in a directory but not being able to see what is in the directory. This happens if the directory has the **x** bit set, but the **r** bit cleared. The write-bit on a directory denotes being able to create or delete files or change filenames. Remember that these actions make a change to the directory, not the file. So it is possible to delete a file that you don't own yourself if you only have write access in the directory it resides in.

Device files only use the read and write bits to determine if a user can get data from a device or send data to it. The execute-bit is not used at all.
Symbolic links do not use the permission bits at all. Instead, the permission of the file pointed to is used. Most Unices just set all bits for verbosity but they serve no function.

Besides these permission bits there are 3 more that can be set for files and directories, namely the setuid, setgid and sticky bit, but these are beyond the scope of this course.

3.1.1 Lab 1: file permissions

This lab assumes your system has been prepared with a script that can be found at
http://crash.hamal.nl/setup_labs.sh

List the rights of the files in your home directory:

```
ls -l
```

Try to see what is in the file README:

```
cat README
```

Make the file readable

```
chmod a+r README
```

see the difference:

```
ls -l
```

try again:

```
cat README
```

4 Interacting with a Unix OS

After logging into a Unix system, you are presented with a prompt of your login shell. There are a few shells available, all with their own features and drawbacks. The most basic shell on most Unix systems is the “Bourne shell”, named after Stephen Bourne who wrote it (its path is generally **/bin/sh**). Though it lacks many features that make interacting with the system easy, it is the “greatest common denominator” on all Unix systems and thus the preferred shell for writing scripts. The original BSD versions provided the C Shell (**csh**) as the default login shell and only provided the Bourne shell for scripting compatibility. **csh** has more features than **sh** but differs enormously from **sh** in its syntax. In later years a rather advanced shell that uses the **csh** syntax was released.

This is **tcsh** (the "**t**" is from TENEX, an OS that inspired some of the features in **tcsh**). Because of its interactive features it soon became the favorite shell of many Unix users. Many commercial Unices provide the Korn shell (**ksh**) for interactive use. This shell is based on the Bourne syntax and many of its features are derived from **csh**. It is named after its writer David Korn. The GNU system (and thus GNU/Linux) provides the "Bourne-Again Shell (**bash**). This is an advanced shell that is backward compatible with the Bourne shell. Because of the popularity of GNU/Linux this is likely the most popular shell. A fairly recent shell to emerge which combines both syntactical as interactive features of **ksh**, **bash** and **tcsh** and some unique features of its own is the Z Shell (**zsh**). This is the preferred shell of the writer of this document but sadly still not known enough to be installed by default on most Unix systems. This is the default shell in a MacOS X terminal.

For the purpose of this course we will limit ourselves to the Bourne shell except where noted otherwise.

A shell command takes the form of:

```
commandname optional argument list
```

Arguments are optional, depending on the command and are separated by spaces. Most commands know a few special arguments consisting of a dash followed by a single character (like **-a**). These arguments are known as *options* because they tend to change the behavior of the program itself. A few programs recognize complete words as options (e.g. **find**) and some recognize both letters and words, where words have two dashes instead of one (especially GNU tools).

Every command has 3 *file descriptors* available. These are "standard input" (a.k.a. *stdin*), "standard output" (a.k.a. *stdout*) and "standard error" (a.k.a. *stderr*). They define where data is read from or where it is printed to. *Stdin* is generally attached to the keyboard of the TTY and both *stdout* and *stderr* are connected to its screen. Most programs that read their input from *stdin* keep expecting more input until the End of File (EOF) signal had been reached. If *stdin* is connected to the TTY's keyboard, this signal can be generated with the **<CTRL+D>** key combination.
Multiple commands can be entered on a single line by seperating them with semicolons (**;**). The commands will then be executed sequentially as if they were entered separately. There are hundreds of commands that are common on all Unices. We will look at a few of these that are used frequently. This will just be a boring summary.

man – Display the manual page. Most Unix systems have a very comprehensive set of documents. To know how to use a command, use **man commandname**. To search for a command based on a keyword, use **man -k keyword**.
echo – Display text on the screen. **echo** prints all its arguments to *stdout*. It is a "shell builtin" command, which means that no external program needs to be started for the command.
read – Read a line from *stdin* and assign it to one or more variables. This too is a shell builtin command.
ls – LiSt the directory content. Filenames that start with a dot are normally hidden for this command (and for *globbing* which will be dealt with later). To see all files including "hidden" files, the option **-a** must be used. The option **-l** causes a "long" listing including file owership, permission etc.
cp – CoPy files. This command requires at least two arguments. The last argument is the destination and all other arguments are source files. If the destination is a directory, the source file(s) are copied with the same name into the destination directory. If only two arguments are supplied and the destination is a file or doesn't exist yet, the source file is copied as the file with the destination's name.

mv – MoVe files. This is analogous to the **cp** command, but files are either moved to a new directory or renamed. This command cannot move files across file-systems since only the directory entry is changed, not the file (i.e., the inode) itself. Some versions of mv (like the GNU version) emulate a cross-file-system **mv** by copying the files to the destination and deleting the source files.
ln – LiNk files. Analogous to the **cp** command, this will create hard links to the source files. If the option **-s** is used, symbolic links are created instead.
rm – ReMove files (in fact: unlink files). This is used to remove files (not directories). A Unix OS doesn't make a habit of asking if you are sure you want to do something, so use this command with care! If the option **-r** is used with a directory as an argument, it will recursively remove the complete directory tree. Very dangerous!
pwd – Print Working Directory. This will print the current directory to *stdout*.
cd – Change Directory. Change the current directory to the argument or to the users home directory if no argument is given.
mkdir – MaKe DIRectory.
rmdir – ReMove DIRectory. This only works if the directory is empty. To remove a directory including all its content, use **rm -r**.
chmod – CHange file MODe. This changes the file permissions for user, group or other.
chgrp/**chown** – Change group or user ownership. **chown** can only be used by the superuser. **chgrp** can be used by ordinary users, but only to change ownership to a group that the user is a member of.
umask – Set file creation mask. This command takes an octal number as argument that defines which permission bits must be *cleared* when creating a new file.
cat – CATenate file(s). Without argument, this will copy *stdin* to *stdout*. When filenames are passed as arguments the contents of all these files are copied successively to *stdout*.
head – Print the first few lines of either *stdin* or the file(s) in the arguments to *stdout*. With the option **-n** the number of lines to print can be specified.
tail – Analogous to head, but print the last few lines.
cut – Extract specific columns from a text file.
grep – Search for lines with specified substrings (see the item on Regular Expressions, in the Scripting chapter).
sed – Stream Editor. Change the text from the input stream and write the changed text to *stdout* (see the Scripting chapter).
sort – sort a file or *stdin* based on its lines.
uniq – remove duplicated lines. With the proper options this can also print only unique or non-unique lines. **uniq** expects duplicated lines the input to be consecutive.
wc – Word Count. Count the number of characters, words and/or lines from *stdin* or files in the arguments.
more – Print the input to the screen, one page at a time.
date – Print or modify the system date and time. Notice that Unix keeps track of the time separately from the hardware clock and counts in seconds from 1 Jan. 1970 00:00 UTC. An *environment variable* (see next item) decides how the time is *represented.*
time – Measure the time it takes to run a program. This will execute all its arguments as entered and print the running time to *stderr*.
tar – tape archive. Create an (uncompressed) archive of a set of files to stdout, a file or a (tape) device.
compress/**uncompress** – (Un)compress a (single) file or *stdin*. Because of patents on the LZW algorithm that **compress** uses, the GNU project developed a patent-free compressor named **gzip**. This gained much popularity and is a “de facto” standard today.
xargs – transfer *stdin* to arguments for a command (see the item on Redirection and Piping)

tee – put a copy of the content of *stdout* in a file and leave another copy in *stdout* (see the item on Redirection and Piping)
expr – evaluate expressions (numeric, string and boolean)
awk – A scripting language (see the Scripting chapter).
find – Find files on that pass given constraints. This is a very powerful command and often used on Unix systems.

4.1 Variables, quoting and globbing

You can assign a (string) value to a variable and access that variable at a later time. Like many other aspects of the Unix OS, variable names are case sensitive but often uppercase names are used. Whitespace is not allowed in variable names. Consider the next example:

```
$ echo $HW

$ HW="Hello, World."
$ echo $HW
Hello, World.
$
```

It is important to omit whitespace around the assignment character when assigning a value to a variable. Accessing the variable is done by preceding the variable name with a dollar character (**$**). In most cases it is important to "export" the variable name so that its value will be present in subshells (more on that in the chapter on "forking"). The command to do this is "**export VARIABLE**".

A few variables are predefined after logging in. These are generally referred to as "environment variables". A few commonly used environment variables are:
$PATH – this is a colon-separated list of paths on the file-system where the shell will search for an external command to execute. Note that the current directory is not searched for a command unless it is part of the **$PATH** variable. In general is it discouraged to do include the current directory.
$MANPATH – like **$PATH** this is a colon-separated list of paths. These are searched for documentation when the man command is issued.
$LD_LIBRARY_PATH – This path is searched by the dynamic loader for shared libraries.
$USER – The user name of the logged-in user.
$HOME – the path to the users homepage.
$PS1 – The prompt string that indicates the shell is ready to receive a command. On most Unices this defaults to a **$** sign for ordinary users and a hash (**#**) for the superuser.
$PS2 – The prompt command to use on a second line when the issued command line continues (generally '**>**')
$PWD – the path of the current working directory.

There are also a few special variables available. They are generally used in shell scripts. These are:
$0 – The name of the issued command (without arguments).
$1 – The value of the first argument to the command.
$2 .. **$9** – Analogous to **$1**.
$# – The number of arguments passed to the command.
$* – All arguments given to the command.
$@** – The same as **$*, but the arguments that contain whitespace remain single arguments.

$? – The “exitcode” of the last command.
$$ – The “process id” of the current shell.
$! – The process id of the last “background” process.

A variable can be enclosed within curly braces to disambiguate the name.

```
$ echo $USER
rsw
$ echo $USERTEST

$ echo ${USER}TEST
rswTEST
$
```

The way variables are handled is affected by quoting. As mentioned earlier, command line arguments are separated by whitespace. That introduces the need for telling the shell when whitespace is not an argument separator but part of an argument. For this, you can use single (**'**) or double (**"**) quotes. The main difference between these two is that variable names are expanded in double quotes, but not in single ones. Consider this example:

```
$ HW="Hello, World."
$ echo "$HW"
Hello, World.
$ echo '$HW'
$HW
$
```

Variable expansion is done by the shell before evaluating the quotes, so if the variable contains whitespace in its value, it will still be passed on to the command as a single argument. Another way to pass whitespace verbatim to the command is by *escaping* it. We do this by preceding it with the backslash character (****). Escaping can be used on other characters as well. If a character has a special meaning (**$**, **|**, **&**, *****, **?** etc), preceding it with a backslash will pass it on verbatim. The following example shows that:

```
$ HW=Hello,\ World.
$ echo $HW
Hello, World.
$ echo \$HW
$HW
$ echo \\$HW
\Hello, World.
$
```

There is one more quote character: the back quote a.k.a. back tick (**`**). The purpose of this quote is command substitution. You can enter a command of which a part is another command in back quotes. The shell will first evaluate the back quoted command and substitute its output in the command line. In the following examples the “current” date is November 2, 2013:

```
$ ls
log.20131001  log.20131008  log.20131015  log.20131022  log.20131029
log.20131002  log.20131009  log.20131016  log.20131023  log.20131030
log.20131003  log.20131010  log.20131017  log.20131024  log.20131031
log.20131004  log.20131011  log.20131018  log.20131025  log.20131101
log.20131005  log.20131012  log.20131019  log.20131026  log.20131102
log.20131006  log.20131013  log.20131020  log.20131027
log.20131007  log.20131014  log.20131021  log.20131028
$ wc log.`date +%Y%m%d`
      1       8      54 log.20131102
$
```

With modern shells this can also be accomplished by enclosing the command between **$(** and **)**. This has the advantage of nesting the commands:

```
bash$ wc log.$(expr $(date +%Y%m%d) – 100)
    630    1616   40744 log.20131002
bash$
```

Another job of the shell is wild card expansion. This is known in Unix as *globbing* since this job was done in the earliest Unices by an external program named "glob". The primary glob characters are the asterisk (*****), the question mark (**?**) and the square brackets (**[]**). Globbing will work on the file-system and expand the glob characters to matching filenames. The asterisk will expand to *zero* or more characters, the question mark will expand to a single character and a number of characters within square brackets will expand to a single character that is mentioned in the brackets. You can also specify one or more ranges of characters in square brackets by using the dash (**-**) character. To expand files to the negation of the characters in the brackets, place an exclamation mark (**!**) as the first character in the brackets. Confused? Hopefully the next example will clarify it:

```
$ ls
bar  foo  foo_4  foo_45  foo_5  foo_A  foo_B  foo_a  foo_ab  foo_b
$ ls foo*
foo  foo_4  foo_45  foo_5  foo_A  foo_B  foo_a  foo_ab  foo_b
$ ls foo_?
foo_4  foo_5  foo_A  foo_B  foo_a  foo_b
$ ls foo_??
foo_45  foo_ab
$ ls foo_[abcde]
foo_a  foo_b
$ ls foo_[A-Z0-9]
foo_4  foo_5  foo_A  foo_B
$ ls foo_[!A-Z0-9]
foo_a  foo_b
$ ls f?o*b
foo_ab  foo_b
$
```

Note that like the **ls** command, globbing considers files that start with a dot to be hidden files:

```
$ ls -a
.  ..  .foo  foo  foo_4  foo_45  foo_5  foo_A  foo_B  foo_a  foo_ab
foo_b
$ echo *
foo foo_4 foo_45 foo_5 foo_A foo_B foo_a foo_ab foo_b
$
```

Modern shells also expand the tilde (**~**) to the user's home directory and **~foo** to the home directory of user *foo*. Quoting any glob character will pass it verbatim to the program and prevent the globbing to occur.

It is important to understand that the globbing is done by the shell, so the program that you start cannot verify if you used a glob character or if you entered all of the arguments yourself. In the first example, when we issued "**ls foo_??**" *two* arguments were passed to **ls**, not one. The big advantage of this system is that the programs can be simpler since they don't need to implement globbing code themselves and that the globbing is consistent for all programs. The disadvantage is that it is possible that globbing can cause the argument list to become too large, resulting in an error. This is solvable with the **xargs** command (more on that later).

4.1.1 Lab 2: File management, substitutions and globbing

Let's start with some basic file management. We will first create a directory to work in Therein we create yet another directory and file and remove them (replace the **X** in **studentX** with your student number).
Create a new directory to perfom the tests:

```
mkdir /tmp/studentX
cd /tmp/studentX
mkdir testdir
touch testfile
mv testfile testdir
rmdir testdir
```

Now we should get an error because the directory is not empty. Let's remove the file first

```
rm testdir/testfile
rmdir testdir
```

We will try out some substitutions. In this test we wil do nesting of variable substitution and command substitution (take notice of the backticks in both examples).

```
DT="date +%Y-%m-%d"
FN=`$DT`
touch "$FN" "$DT"
ls -l
```

As you can see, the variable substitution happens first and is evaluated along with the command substitution. The order is not reversable. Try this example (substitute **2013** with the current year):

```
Y="Single Letter "
Y2017="This Year"
echo "$Y2017"
echo "`date +%Y`"
echo "$Y`date +%Y`"
```

Like Lab 1, the next part assumes your system has been prepared with a script that can be found at **http://crash.hamal.nl/setup_labs.sh**

Go into the "**globme**" directory and list all files that end with a digit:

```
cd ~/globme
ls *[0-9]
```

List all files containing a digit:

```
ls *[0-9]*
```

List all hidden files:

```
ls .[!.]*
```

Why would it be incorrect to use "**ls .***"?

List all hidden files that contain the word "Alice":

```
grep -l Alice .[!.]*
```

Count the lines in all hidden files that contain the word "Alice" (note the back quotes):

```
wc -l `grep -l Alice .[!.]*`
```

List all files that are exactly five characters long:

```
ls ????? .????
```

ls has the "**-a**" options to show hidden files. Why doesn't "l**s -a ?????**" work?

4.2 Redirection and piping

As mentioned earlier, a command has 3 *file descriptors* available: *stdin*, *stdout* and *stderr*. These are numbered file descriptor 0, 1 and 2 respectively. For advanced use there are more available. Also mentioned earlier is that the file descriptors are connected to the TTY's keyboard (FD 0) and screen (FD1 and 2) by default, but this can be changed. To redirect an input stream, use the "less than" sign (**<**) and to redirect output, use the "greater than" sign (**>**). This sign is preceded by the file descriptor number. E.g., if you want to redirect the standard input from a file, you do:

```
command 0< /path/to/the/file
```

To redirect standard error to a file, use:

```
command 2> /path/to/outputfile
```

If the file descriptor number is omitted, either *stdin* (in case of input) or *stdout* (in case of output) is assumed. Sometimes it is required to redirect both *stdout* and *stderr* to the same output channel (e.g. redirect both to the same file). For that you have to "connect" *stderr* to the *stdout* channel and only redirect *stdout*. That is done like this:

```
command > /path/to/file 2> &1
```

In this case the file descriptor numbers are mandatory.

Output redirection normally overwrites existing files or creates a new one when needed, but there is a way to append output to an existing file. This is done by using two "greater than" signs instead of one:

```
command >> appended_file
```

The **<<** redirection also exists, this is called the *here document* and is mainly used in scripts. We will look at it in chapter 6.2.

Besides redirecting output to files or input from files, it is also possible to connect the output of one command to the input of another. This is called *piping* and uses the "pipe character" (**|**). This lies at the heart of the Unix philosophy, i.e. combine small tools to build the needed functionality.
Example:

```
command1 | command2
```

This will connect *stdout* of command1 to *stdin* of command2. Note that **command2** does not wait for **command1** to finish, **command2** starts immediately and eventually waits for input. Only *stdout* can be connected to the "left" side of the pipe, but you can connect *stderr* to *stdout* (**2>&1**) to have both piped to the next command.

Two commands that are used extensively with pipes are **xargs** and **tee**. **xargs** will transfer its *stdin* to arguments for the command that is specified in its own arguments. An example:

```
find /var/log -mtime -4 -print | xargs grep -l 'kernel error'
```

The first part here will print a list of filenames in the **/var/log** tree that are modified or created less than 4 days ago and **xargs** will transfer those filenames as arguments for the **grep** command, which will print only those filenames (**-l**) of the list that contain the text "kernel error" in their content.
tee is used if you want to save the output of a command to a file and still use the same output as piped input for another command. The syntax is:

```
command1 | tee /path/to/outputfile | command2
```

This idea is derived from physical plumbing where a T-piece is used in pipe-fitting.

If you want to concatenate *stdout* and/or *stderr* of multiple commands and pipe the combined result into the next command, you can group commands in two ways, with parentheses or curly braces. The difference is that commands in parentheses are executed in a sub-shell, while commands in curly braces are executed in the current shell. The following should clear it up a bit.

```
$ echo Foo ; echo Bar | wc
Foo
      1       1       4
$ { echo Foo ; echo Bar ;} | wc
      2       2       8
$ (echo Foo ; echo Bar) | wc
      2       2       8
$
```

As you can see, the space after the opening curly brace is mandatory, as is the terminating semicolon before the closing brace.

4.2.1 Lab 3: redirections and pipes

We're doing some simple redirects to see the result. Let's put the output of the "**date**" command into a file and check the content (first go to the directory of the last lab):

```
cd /tmp/studentX
date > testout.txt
cat testout.txt
```

Let's append something to the file:

```
echo "A command line can be versatile" >> testout.txt
cat testout.txt
```

What would happen with a single redirect?

```
echo "A command line can be versatile" > testout.txt
cat testout.txt
```

Let's count the characters in the file:

```
wc -c < testout.txt
```

But if that was the intent, we don't need the intermediate file, we can count the characters immediately:

```
echo "A command line can be versatile" | wc -c
```

Witness the difference between stdout and stderr:

```
dd if=testout.txt
dd if=testout.txt >/dev/null
dd if=testout.txt 2>/dev/null
```

4.3 Forks, jobs and processes

When an external command is issued on the command line, a sub-shell is started and the command is executed in that shell. The “parent” shell then waits for the “child” to finish. This process is called *forking*. This has a few consequences. Changes in the environment of the child process will not be visible in the parent. Also variables are only inherited in the child shell if they are *exported* with the **`export`** command. The following shows just that:

```
$ pwd
/home/rsw
$ VARPARENT=foo
$ VARBOTH=bar
$ export VARBOTH
$ echo $VARPARENT
foo
$ echo $VARBOTH
bar
$ echo $VARCHILD

$ sh
$ # Now we're in a subshell
$ echo $VARPARENT

$ echo $VARBOTH
bar
$ VARCHILD=baz ; export VARCHILD
$ VARBOTH=qux ; export VARBOTH
$ echo $VARCHILD $VARBOTH
baz qux
$ cd /tmp
$ exit
$ # Now we're back in the parent shell
$ pwd
/home/rsw
$ echo $VARBOTH
bar
$ echo $VARCHILD

$
```

It is possible to execute a script without forking a sub-shell. This is frequently used for setting variables from a script which should stay visible in the “parent” shell. This process is called *sourcing* a script. In Bourne-like shells, it is done with the “dot” command like this:

```
. scriptfile
```

Note that this is only allowed with shell scripts, not with binary executables. It is possible to start a program (whether script or binary) in the context of the current shell, but after that script or program finishes a signal is sent to the parent of your *current* shell and your current shell exits. If it was your login shell, this will cause a logout. To accomplish that, use

```
exec program program_argument_list
```

Every process on a Unix system has a *process id* (PID). With the command **`ps`** you can get a lot of information on the various processes, like their PID, the PID of their parent process, the user who

started the process, the TTY it's connected to etc. With the **kill** command you can send a signal to a process. Some signals (like SIGINT, the Interrupt signal) are handled by the process itself (most programs will exit cleanly if they receive SIGINT) while others (e.g. SIGKILL) are handled by the kernel. With the **kill** command you can send any signal to any process. Of course a process will only act on the signal if you have the proper permissions. If you interact with a program via the shell you can send a few signals to it via key combinations. The **<CTRL+C>** keys send a SIGINT to the process and the **<CRTL+Z>** keys send a SIGSTOP which normally suspends a process (see the next item on jobs).

A program can be started in the background. That means that the program starts and the shell prompt is returned so that another command can be issued. Starting a program in the background is done by appending an ampersand (**&**) to the end of the command line:

```
commandname arguments &
```

A running program can be suspended by sending it a SIGSTOP signal. This can be done by pressing **<CTRL+Z>** on the program's TTY or issuing "**kill -STOP *PID***" (where ***PID*** is he program's process id) from a shell. The **jobs** command will list all suspended and background processes started by the current shell. The numbers you see in the listing are not process ids but *job ids*. You can access the jobs by preceding the job id with a percent character (**%**). You can call one of the jobs to the foreground (your interactive session) by issuing the command **fg %jobid**. If the job id is omitted, the current job is assumed (denoted in the **jobs** listing with a plus (**+**) sign). Similarly, you can continue running a suspended job in the background by using the **bg** command. **jobs**, **fg** and **bg** are shell builtin commands. When you have active jobs (either in running or in suspended state) the shell will prevent you from logging out. Some shells may still log out but terminate the jobs while doing so. To start a program in the background that will keep running after you logout, use the **nohup** command. This will start the command without a TTY but send all output to a file named "**nohup.out**".

4.4 Scheduling

Sometimes the need exists to start a program non-interactively on a specified time. This can be accomplished with **at** or, if it should be done repetitively, with **cron**. **at** expects the command and command arguments in its standard input. The time and date to execute the command are specified in its arguments. An example might be:

```
echo "find /tmp -mtime +30 | xargs rm -f" | at 20:08 tomorrow
```

This will delete all files in **/tmp** tomorrow at 8:08 pm which are *then* older than 30 days.
If a job needs to run regularly, it can be scheduled with the **cron** daemon. **cron** reads its configuration from a file named **crontab** which is somewhere in the **/var** tree on most Unices. This file can be viewed or edited with the **crontab** command and has the following syntax:

```
a b c d e commandname argument-list
```

The moment of execution of the command is specified in the first 5 fields. All of these arguments can either be a numerical value as described below or an asterisk as value (meaning *all* values are valid). It is possible to put multiple values in a field separated by commas. The moment of execution is the moment that all arguments are true (except for fields "c" and "e", the command runs if one of those is true). Specifying 5 asterisks will execute the command every minute. The explanation of the first 5 fields is:

a – the minute value of the time of day (can be 0-59)
b – the hour value of the time of day (0-23)
c – the day of the month (1-31)
d – the month (1-12)
e – the day of the week (0-6, 0 is Sunday).
So the example:

```
5 * * 3,6 2 echo Cron worked on `/bin/date` >> /tmp/myfile
```

will append text to **/tmp/myfile** 5 minutes past every hour on every Tuesday in March and June. Both **cron** and **at** lack a connection to a TTY. If output is generated by the command it is sent by email to the user. **at** inherits the exported environment variables, but **cron** has a very limited environment. Usually it is advisable to include complete paths to the programs in the **crontab** entry.

4.5 Shell initialization

After login, the shell will source a few files if they are present. With these file you can set up new defaults for various environment variables, define often used functions (see par. 6.1) etc. These can be set system-wide by the system administrator, or by the user on a per-user basis.
The Bourne and Korn shells will source the file **/etc/profile** on starting a login shell. If the user has a **.profile** in her homedirectory, that file will be sourced next. The Bourne-Again shell will also source these files, but also **/etc/bashrc** and **$HOME/.bashrc** on non-login interactive shells. **$HOME/.profile** can be replaced with **$HOME/.bash-profile**. This will prevent it from being sourced by **sh** or **ksh**, so bash-specific extensions can be used. See the manpage for the various shells for what files are sourced at shell startup.

5 Networking

In the early days of Unix users were connected to the system via physical connections (mostly via serial lines). There is no fundamental difference between terminals connected with a serial line and the way a monitor and keyboard are connected to a personal computer. To overcome the proximity constraint the serial line could be extended via two modems and the POTS telephone network. In those days one Unix computer could connect to another via a serial line and share files via a set of tools, collective named *UUCP* (Unix to Unix CoPy). When TCP/IP networking became available, tools were developed to enable users to interact with the system via this network. The most common of these tools were **telnet**, **ftp** and the "rsh" tools (**rlogin**, **rsh** and **rcp**). **telnet** and **rlogin** are used for logging in on a remote system and interacting with it. **ftp** and **rcp** are used for file sharing between two hosts on a network and **rsh** is used for executing a command on a remote host and have the results display on the local host. These tools were sufficient at the time, but these days they are almost obsolete and users have mostly moved to **ssh**, both for its enhanced security and its ease of use. **ssh** is in fact mainly a secure replacement for **rlogin** and **rsh**, while the secure equivalent of **rcp** is named **scp**. In the latter verions of the ssh suite a secure replacement for **ftp** came available named **sftp. ftp** is still widely used, but generally not for authenticated file sharing. From one system a user issues:

```
telnet remote_host
```

and is presented with a login prompt of the remote system.

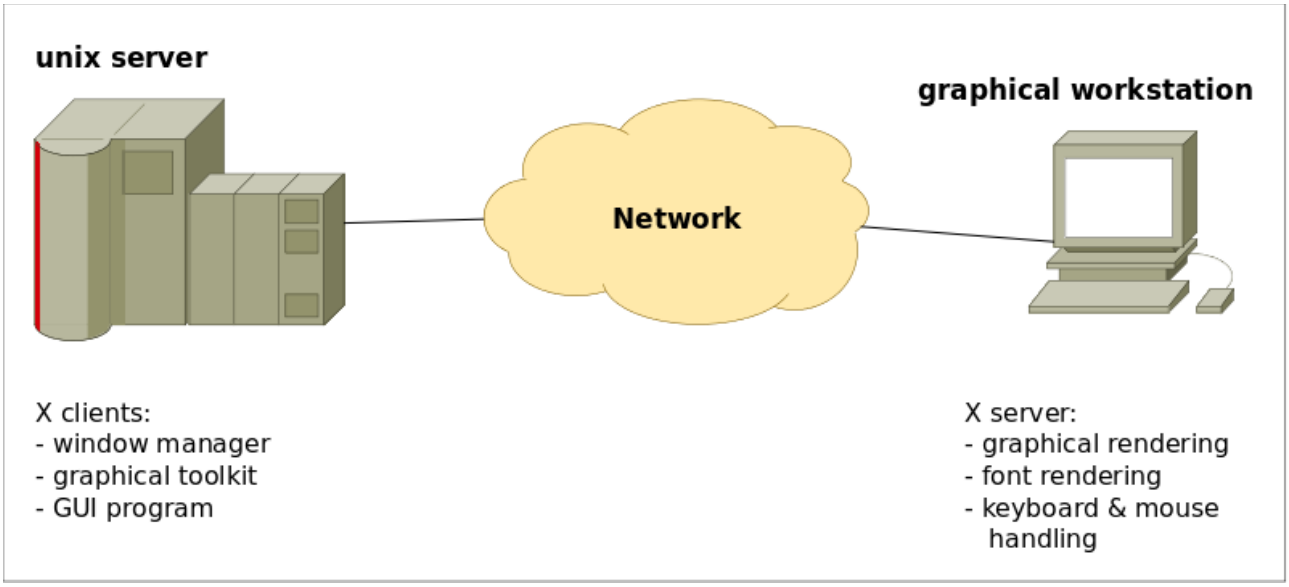
It is important to understand that there is no fundamental difference between a user who interacts with a system after logging in with **telnet** or **ssh**, and a user who interacts with a system via a

physical connection (serial line or PC monitor and keyboard). Both can freely start any program they like. That differs from a “service” that is provided and accessible via the network, e.g. a webserver which enables a CGI program to be executed. In the latter case the program is executed on the server, but the user is constrained by the webserver in what program to run and how. With shell access those constraints are absent. The TTY that is assigned to a user who logs in via the network is determined dynamically. For this reason it is referred to as a “pseudo TTY”. The main difference between the ssh suite and the other tools is that unlike the latter, the ssh suite uses an asymmetric cryptographic system to encrypt all data (the same way as SSL web encryption works in the browser).

5.1 The graphical user interface

You might wonder what a graphical user interface has to do with networking. On Unix networking is an intrinsic part of the GUI. Since Unix is a multi-user OS it is often the case that a user is connected to a Unix system via the network, sometimes from the other side of the world. The Unix system has to be able to properly display graphical programs to all these users. The GUI system that is commonly used on a Unix system is called “the X Windowing System” (sometimes shorted as X-windows or X). It differentiates between the “*X server*”, i.e. the program that is capable of displaying graphical content, handling the mouse and keyboard and rendering fonts, and the “*X client*”, i.e. the program that requires graphical output. The terms “server” and “client” are a bit confusing here, since most people expect some big piece of iron when they hear the term “server” and surely don't expect a workstation to carry that name, but the name makes sense. The workstation *serves* the ability to display graphics. In fact, most big Unix systems that host hundreds or more concurrent users are very capable of running graphical applications but lack graphical hardware themselves.

Schematically it looks like the following:



The Unix system uses an environment variable to know where the X server resides. This variable is **`$DISPLAY`** and takes the form of “**`hostname:a.b`**” where **a** and **b** are numbers. **`hostname`** is either a name or IP address of the X server, **a** is the *display* number and **b** is the *screen* number. The display number is mandatory. Most workstations have only a single X server running, so generally the display number will be 0. Still, it is possible to run multiple X servers on a single workstation.

For the Unix server these will be separate instances. The screen number is optional. If omitted (along with its preceding dot) a screen number of 0 is assumed. Screen numbers are used to distinguish multiple screens in a “multi-headed” workstation.

Since the X windowing system is based on network connections, there needs to be some permission system to decide what X clients can access the X server. There are two mechanisms for managing access: **`xhost`** and **`xauth`**. **`xhost`** is used to allow or disallow access to the X server, based on the *host* that connects. This implies that all users on that host have write access to the screen and read access from the keyboard and mouse. Using “**`xhost +`**” allows access to all users on all hosts on the network. This is not the preferred option to chose. **`xauth`** uses what is called a “*magic cookie*” for the permission system. From a shell in the X environment on the workstation the user lists a “cookie” (a 128 bit value that is generated on server startup) and enters that with the displayname on the system that needs access to the X server. That allows only that user on that system access to the X server. This is a lot better, but still not the best system, both from the perspective of usability as from that of security. As stated earlier, **`ssh`** is chosen partly for its ease of use. This is one such area. **`ssh`** can create an X tunnel, i.e. that it will create a “pseudo displayname” on the Unix system that runs the X clients and add proper access to that displayname via **`xauth`**. Clients that connect to that pseudo display will access the real X server via a *tunnel* that is embedded in the encrypted connection between Unix host and X workstation. This increases the security, especially on the network between X clients and X server. It also automates the **`xauth`** setup which is an improvement from the point of usability.

6 Shell scripting

Multiple shell commands can be put into a file and executed by means of that file. Such a file is called a shell script. In fact the ability to create a file with the a collection of shell commands introduces nothing new that cannot be done on the command line. Still, it is deemed important enough to write a chapter about it. The reason is that certain aspects of the shell, like flow control, are not frequently utilized on the command line but frequently are in scripts.

Like most other scripting languages in the Unix environment, the hash character (**#**) is treated as a comment indicator. Anything following a hash up to the end of line is silently ignored – except for a single case. As stated before, the way Unix determines if a file is executable is by the execute-bit in the permission bits. This is equally true for scripts, so there should be a way for the OS to differentiate between scripts written in Bourne or C shell, awk, perl and a myriad of other scripting languages. This is done by putting a *shebang* in the very first line of the script. This has the following syntax:

```
#!/path/to/shell optional_arguments
```

The name “shebang” is probably derived from “shell bang”. The exclamation mark is called a “*bang*” in certain Unix contexts (e.g. UUCP). So a Bourne shell script will have the text “**`#!/bin/sh`**” as the first line. That way the OS can determine which interpreter to start for the script.

6.1 Shell functions

We have seen before that curly braces can be used to group a number of commands for combined redirection. This construct can also be used to create shell functions. A name can be assigned to a

group of shell commands in braces. These commands are not executed until they are referenced by the assigned name. Example:

```
usage() { echo $0 'start | stop | restart' ; }
# some more commands
usage
```

6.2 Here-document

In the item on redirection on page 19 it was mentioned that it is possible to use the **<<** construct for input redirection. This form of redirecting is called the *here-document*. It will use all the next lines as input to the command until the literal word specified after the **<<** is found at the start of a line. Keep in mind that whitespace preceding a command is generally discarded and that this is used for indenting shell scripts for verbosity, the here-document is sensitive to leading whitespace. Example:

```
$ cat << FOO
> Hello, World!
>    This is part of a here-document
>    FOO
> FOO
Hello, World!
   This is part of a here-document
   FOO
$
```

6.3 Flow control

For being able to utilize different flows in the script, the shell must be able to assign boolean (true of false) values. This is accomplished with the *exitcode*. When a program exits it returns a value to the shell. If this value is zero, the boolean value is considered *true*, and if the value is non-zero the boolean value is considered *false*. This is different from e.g. perl and C, where 0 is considered false and non-zero true.

The Bourne shell recognizes the following flow constructs: **`if`**, **`case`**, **`for`**, **`while`** and **`until`**.

An if statement has the following syntax:

```
if first_command
then
    #do something
elif second_command
then
    #do something
...
else
    #do something
fi
```

At an **`if`** or **`elif`** statement the command is executed and the return value is evaluated. Multiple **`elif`**s are allowed, but only a single **`else`** is. Both **`elif`** and **`else`** are optional. If no evaluation succeeds and no **`else`** statement is included, nothing in the “body” of the statement is executed.

An example:

```
if test "$ANSWER" = "Yes"
then
    echo "Thank you."
elif test "$ANSWER" = "No"
then
    echo "I hope you'll reconsider soon."
else
    echo "Please answer Yes or No."
fi
```

More on the test command in the end of this chapter.

There is a simplified mechanism for alternation available which uses the **&&** and **||** constructs:

```
command1 && command2
```

This will execute command 1 and if the exitcode is zero, it will execute command 2.

```
command1 || command 2
```

This will execute command 1 and if the exitcode is non-zero, it will execute command 2.
They can also be combined:

```
command1 && command2 || command3
```

Execute either command2 or command3 depending on the exitcode of command1.
Example:

```
grep foo $HOME/bar > /dev/null 2>&1 && echo "Foo is listed" \
     || echo "Nothing found"
```

The case statement only evaluates variable values. The syntax is:

```
case string in
     value-list1)
          #do something
     ;;
     value-list2)
          #do something
     ;;
     ...
esac
```

It is useless to evaluate a literal string, so in general a variable will be placed in the position. The value-lists are one or more values separated by the pipe character. If any of these values match the string, the commands up to the next **;;** are executed. If multiple values match the string, only the first matching entry is executed. Globbing rules apply to the values when matching with the string.

Example:

```
case $1 in
    start|stop)
        /usr/sbin/sshd $1
    ;;
    restart)
        $0 stop
        $0 start
    ;;
    reload)
        pkill -HUP sshd
    ;;
    *)
        echo "Usage: $0 start|stop|restart|reload"
        exit 1
    ;;
esac
```

The **`while`** statement will execute one or more statements repetitively (this is called a loop) as long as the condition is true. The condition is a command and its exit value is evaluated. The syntax is:

```
while command
do
     #loop body
done
```

It is clear that either in the loop body or externally something should happen to have the command return a non-zero value eventually or else the loop will keep executing forever. The check on the command can also be inverted by using "**`until`**" instead of "**`while`**".

During execution of the loop body it is possible to disregard the rest of the statements and either exit the loop or start the next incarnation immediately. You can exit the loop with the **`break`** command and start the next instance of the loop with the **`continue`** command.

The **`for`** statement will execute one or more statements repetitively while assigning different values to a variable during each incarnation. The syntax is:

```
for variable in value-list
do
     #loop body
done
```

The value-list contains whitespace-separated values and with each incarnation the variable will be assigned the next value of this list. The **`break`** and **`continue`** statements work in the **`for`** loop as well.
Example:

```
for name in foo bar quux*
do
    test -d $name && tar cf $HOME/backup/$name.tar $name
done
```

A command that is used very often in **if** and **while** statements is the **test** command. This command is a shell builtin command that can do various comparative operations on strings, numbers and files. The command can be written down in two ways, as "**test expression**" or as "**[expression]**". It is possible to make boolean combinations of various expressions including negation and setting preferences. **test** will exit with a zero value on success and a non-zero value on failure. Some tests that are supported are string equality, comparative values of numbers, file types etc. Some examples:
Test string equality:

```
[ "$VAR" = foo ]
```

Test if string is empty:

```
[ -z "$VAR" ]
```

Do a numeric comparison:

```
[ "$VAR" -lt 12 ]
```

Is **foo** a directory?

```
[ -d foo ]
```

See the *manpage* for a complete list. Take notice of quoting of the variables. Omitting that might cause syntax errors.

6.3.1 Lab 4: Write a small shell script

Use the "**nano**" editor to create a file named **/tmp/studentX/myscript**. The script should output different texts depending on the name with that is used to run the script with. If the script is run using the name "**myscript**", output the text "**Hello, World**". If the same script is run using using the name "**secondname**", output the text "**Hello, Universe**". Lastly, if the script is run using any other name, output the text "**Please run me with another name**".

You can either use an if statement or a case statement to evaluate which name is used. The full name of the script is placed in the "**$0**" variable. If the script is finished, make it executable with

```
chmod a+x /tmp/studentX/myscript
```

Also create two alternate names for the script by creating symbolic links to it:

```
ln -s /tmp/studentX/myscript /tmp/studentX/secondname
ln -s /tmp/studentX/myscript /tmp/studentX/thirdname
```

Run the script with all three names and see if it outputs different texts.

6.4 Regular expressions

Regular expressions (usually shorted as *regexps*) are constructs that are used extensively in many programs in the Unix environment, such as **grep**, **perl**, **sed**, **awk** and **vi**. Formally, regexps constitute the language of strings that can be expressed by a *state transition diagram* (STD). This enables powerful data-structures that can parse the text efficiently. Although many tools support regexps, their syntax differs a bit. The basic syntax is as follows.

- A string is a concatenation of characters and is recognized as itself.
- A dot is a placeholder for any character.
- If a string of characters is enclosed in square brackets (**[]**) a single character from that string is recognized. If the closing bracket needs to be recognized, it should be placed as the first character in the string. Ranges of characters can be specified with a dash (e.g.

[A-Za-z0-9]: this will recognize a single alphanumeric character). The string can be negated by preceding it with a caret (e.g. **[^0-9]** denotes a single non-numeric character.
- An asterisk (*****) denotes *zero* or more repetitions of the previous character.
- Starting the regexp with a caret (**^**) binds it to the start of the line. Ending it with a dollar sign (**$**) will bind it to the end of the line. Binding can also be done with **\<** and **\>** to bind the regexp to the beginning and end of a word respectively.
- Separating two regexps with an excaped pipe symbol (**\|**) will recognize any of the regexps. In extended regexps escaping the pipe symbol is not necessary.
- Part of a regexp may be grouped within escaped parentheses ("**\(**" and "**\)**") and referenced with **\n** where **n** is the number that corresponds to the **n**th grouping. Like with the pipe symbol, escaping should not be done when using extended regexps. E.g. the (basic) regexp "**\(abc\)\(def\)xxx\2\1**" will recognize the string "**abcdefxxxdefabc**.".
- In extended regexps (such as in **perl** and **egrep**, the repetitions of a character (or group of characters) can be quantified.
  - **{n}** – the character is repeated exactly **n** times.
  - **{,n}** – the character is repeated at most **n** times.
  - **{n,}** – the character is repeated at least **n** times.
  - **{n,m}** – the character is repeated at least **n** and at most **m** times.

  It is also possible to use a plus sign (**+**) to specify at least one repetition of the character or the question mark (**?**) for zero or one instance of the character.

6.5 sed and awk

Before the heydays of **perl**, most jobs of parsing and modifying text automatically was done with a combination of **sed** and **awk**. Though **perl** and other scripting languages like **python** have generally taken over that part, **sed** and **awk** are still used, mostly for the simpler and quick-and-dirty jobs.

6.5.1 sed

sed is a stream editor. It will read *stdin* or one or more files and send the modified text to *stdout*. **sed** takes an optional range specifier and a command or multiple commands enclosed in curly braces. Lines that are unaffected by either the range specifier or the command are printed unchanged to *stdout*. Take for instance an email in the Unix "*mbox*" format. This contains the headers (Date:, Subject: etc.), followed by an empty line, followed by the mail body. If we would only be interested in the body and we have the complete mail text in *stdin*, we could strip the headers with

```
sed -e '1,/^$/d'
```

Here the range is "**1,/^$/**" which specifies the range from the first line to the first occurrence of the regular expression "**^$**", which is an empty line. The command is "**d**", which deletes the lines from the stream.

If the range is omitted, all lines are subjected to the command. The range can be a single number or regexp or it can be two numbers and/or regexps separated by a comma. Using a single range element indicates that only the specified line will be subjected to the command. Two range elements indicate a range from the first line number or regexp occurrence to the last, including both specified lines. A single dollar sign denotes the last line. A few commands require only a single range element. In the range field, a regexp must be delimited by slashes (**/**).

If multiple commands are given surrounded by curly braces (**{}**) each command and the closing brace must be on separate lines. Some of the most common commands are:

- Substitute. The syntax is "**s/regexp/replacement/flags**". Unlike the regexp in the range, the slash can be substituted by any character as long as all three characters are identical. In the substitution references to groups (**\n** where **n** is a number) are allowed. The ampersand (**&**) denotes the complete recognized string. Flags are optional and are used e.g. for replacing all occurrences on the line (**g**), or do a case insensitive match (**i**, GNU extension).
- Delete. The syntax is "**/regexp/d**". This will delete entire lines that match the regexp. The regexp is optional and when omitted, the command will delete all lines.
- Append/Insert. The syntax is **a*text*** or **i*text***. A single range parameter is mandatory and the text is inserted either after (**a**) or before (**i**) the specified line.

6.5.2 awk

awk is a programming language that is especially useful for parsing text and acting upon that. It is named after its designers Al Aho, Peter Weinberger and Brian Kernighan. The language was extended some time after its invocation, and to distinguish between the old and new syntax, the new one was named **nawk** (new awk). The GNU project created their own free version which is compatible with **nawk** and named it **gawk**. In the next paragraphs **awk** refers to the **nawk** syntax. **awk** works on one or more files specified on the command line, or on *stdin* if no files are specified. The structure of an **awk** program is:

```
pattern { statements block }
pattern { statements block }
...
```

The pattern is optional. Without a pattern all lines from the input are subjected to the corresponding statements. Two patterns are special: **BEGIN** and **END**. All statement blocks with a **BEGIN** pattern are executed before any lines are read from input and all blocks that are preceded by the **END** pattern are executed after all input lines have been read. Other patterns may be any of:

- **/pattern/** – a regular expression. In awk the extended regexp syntax is used (like in **egrep**).
- a relational expression (e.g. **a < b**, **$4 ~ /foo/**, etc)
- A boolean construct of patterns with **&&** (boolean and), **||** (boolean or) and **!** (boolean not). Parentheses alter the evaluation order.
- **pattern ? pattern : pattern** – alternative pattern evaluation, the syntax compares to that of the C programming language.
- **pattern1, pattern2** – specify a range of input lines from the line containing **pattern1** to the line containing **pattern2** , inclusive.

The input line is divided in multiple "*fields*", separated by whitespace (unless the variable **FS** is redefined). These are accessed with the field variables **$n** where **n** is a number. **$0** indicates the whole line. Variables are just alphanumeric names starting with a letter and are referenced like that (comparable to the C programming language). **awk** recognizes string and numeric variables (without explicit declarations) and arrays of variables (arrays in **awk** are *associative*, which means that the array index does not need to be numeric and even if it is numeric it is treated as a string. Arrays are noted with square brackets like **arrayname[index]**.

The statement block can consist of multiple statements separated by semicolons (**;**). A statement can consist of an *action statement* (like **print**), or a *flow statement* (**if**, **for**, **while**, etc.) which itself has one or more action statements of its own. Multiple action statements that belong to a flow statement can be combined in a block of its own, surrounded by braces. The syntax of the flow statements is again very much like C. For example, the next program calculates the first 10 Fibonacci numbers:

```
awk 'BEGIN {    cnt=0
                a=0
                b=1
                while (cnt < 10)
                {    cnt++
                     c=a+b
                     a=b
                     b=c
                     print "Fib(" cnt ") is "c
                }
          }'
```

awk is often used for simple parsing jobs in shell scripts like print only the file sizes in a directory:

```
ls -l | awk '{print $5}'
```

or sum the filesizes:

```
ls -l | awk '{sum += $5} END {print sum}'
```

or sum the filesizes of only executable files:

```
ls -l | awk '$1 ~ /^-.*x/ {sum += $5} END {print sum}'
```

7 Miscellaneous

7.1 Editors

A lot of a Unix system considers text files. Most of the tools mentioned here expect plain text input and/or produce it. An important tool in the OS is a text editor with which the user can create or modify text files. The earliest Unices used **ed** as the standard editor. This is a line-based editor where users can modify the text based on commands that apply to the current line or a range of lines. It is nearly identical to **sed** with the exception of using it interactively.

These days the standard editor on about every Unix is **vi**, written by one of the original BSD developers named Bill Joy. This is derived from a descendant from **ed** named **ex**, but can switch between “visual” editing and “line” editing. In its line mode it is very comparable to ed. Though it has a steep learning curve, it is extremely powerful and efficient which makes it the editor of choice for most Unix administrators and consultants. **vi**'s visual editing knows 2 modes of operation, *insert* mode and *command* mode. From command mode you get into insert mode with the commands **i**, **I**, **a**, **A**, **o**, **O**, **s**, **S**, **c** or **C**. These differ in where the cursor is placed and whether text is deleted in the process. To get back to command mode, the **<Esc>** key should be pressed. In command mode the cursor can be moved around the text, blocks of text can be cut and/or pasted,

etc. Also from command mode, an ex command can be issued by typing a colon (**:**) followed by the command. All searches in **vi** are based on (basic) regular expressions.

There is a large number of alternatives available for editing text, but most of these are not installed by default on a Unix system. The best known is **emacs**, which was intended to be an integral part of the GNU system. **emacs** is a lot more than just a text editor, it can read mail, it can do file management and access remote files as if they were local, it is extendable by *LISP* programs, etc. **emacs** is sometimes regarded as a shell of its own. There are also many "flamewars" on the Internet between the Unix administrators of the "vi-camp" and the "emacs-camp". Most modern shells (**tcsh**, **bash**, **zsh**) use key bindings from **emacs** for command editing.
Another popular editor is **pico**. This is the editor that is distributed as part of the mail and news reader **pine**. The look and feel is mostly comparable to "Wordstar" that was used on the DOS platform in the eighties. Because of some license restrictions, a clone of **pico** was created and named **nano**.

When we talk about editing text files, we should note one aspect where Unix differs from other operating systems, i.e. in the line ending of text files. This is mostly visible when comparing texts made on Unix with texts made on a Windows system. A DOS or Windows text editor will create lines that end with 2 control characters, the "carriage return" (CR) followed by the "line feed" (LF). This is derived from how a mechanical typewriter works. In Unix, lines end with only the LF. CR has an *ASCII* value of 13 and LF has a value of 10. When viewing a file created on Windows with certain programs in a Unix environment, the CR characters are displayed as "**^M**":

```
Hello Reader,^M
This demonstrates how^M
line ends differ.^M
```

When a file is created on Unix and viewed with certain programs in DOS or Windows, you get a "staircase effect" or the LF is printed as a special character and everything gets printed on the same line. There are tools to transfer from DOS to Unix line ends (**dos2unix**, **fromdos**) and the other way around (**unix2dos**, **todos**) but it can also be done with **sed**:

```
sed -e 's/\r$//' < dosfile > unixfile
sed -e 's/$/\r/' < unixfile > dosfile
```

7.2 Programming

As previously mentioned, Unix is a big toolbox with a lot of small tools that can be combined to create the needed functionality. Sometimes though, a specific tool may be lacking. For those purposes almost every modern Unix has a C compiler and the essential C libraries to create your own binary or compile a program from another environment for the current one:

```
$ cat hw.c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("%s\n", "Hello, World!");
    return 0;
}

$ cc hw.c
$ ./a.out
Hello, World!
$
```

For programs that are slightly more complex than this, the **make** program can be used. Though **make** is generally used for development purposes, it is in fact a kind of scripting.
The general syntax is:

```
target: prerequisites
     command
     ...
```

The target is the argument that is given to the **make** command. It will execute all commands in the given target block. All command entries *must* be preceded by the literal TAB character. If the prerequisite files don't exist. **make** will look for a target with the name of the file and execute those commands. Calling **make** without an argument will cause it to search for the target **all:**. If we take our **hw.c** file above as an example, make could be used like this:

```
$ cat Makefile
all: hw.o
     cc -o hw hw.o

hw.o: hw.c
     cc -c hw.c
$ make
cc -c hw.c
cc -o hw hw.o
$ ./hw
Hello, World!
$
```

Appendix A: Manpage Syntax

Manpages are divided in different sections. Often the number of the section is added in parentheses after the subject to indicate which section is meant, like **passwd(1)** or **passwd(5)**. In the first case, the **passwd** entry in section 1 of the manpages is indicated, i.e. the entry for the **passwd** command. The entry in section 5 deals with the format of the file **/etc/passwd**. For a complete listing of the sections and what kind of manpage to expect there, see the manpage for the **man** command. Some environments, most notably **perl**, include an addition to the section to indicate the environment, e.g. **Time::ParseDate(3pm)**. A manpage may look (in part) like the following:

```
MAN(1)                    Manual pager utils                    MAN(1)

NAME
       man - an interface to the on-line reference manuals

SYNOPSIS
       man  [-c|-w|-tZ]  [-H[browser]] [-T[device]] [-adhu7V] [-i|-I]
       [-m system[,...]] [-L locale] [-p string] [-C file] [-M  path]
       [-P  pager]  [-r  prompt]  [-S list] [-e extension] [[section]
       page ...] ...
       man -l [-7] [-tZ] [-H[browser]] [-T[device]] [-p  string]  [-P
       pager] [-r prompt] file ...
       man -k [apropos options] regexp ...
       man -f [whatis options] page ...

DESCRIPTION
       man  is the system's manual pager. Each page argument given to
       man is normally the name of a program,  utility  or  function.
```

The SYNOPSIS part will give the syntax of the command with all possible options. When commands are very closely related, they are sometimes documented with the same manpage where the SYNOPSIS will show the various versions (see e.g. **printf(3)**). Optional arguments are enclosed in square brackets and mutually exclusive arguments are seperated by pipe symbols. An ellipsis (**...**) indicates possible repetitions of the last argument. The sections that follow give in-depth descriptions of the commands and options and may also refer to related files and other manpages.

Appendix B: Common Unix commands

command	builtin	description
`.`	y	execute a script without forking and return (source)
`:`	y	return a zero value (same as true)
`[`	y	test conditions (man test)
`at`		execute a command at a later time
`awk`		text parsing language
`basename`		strip path componenr from a file
`bc`		CLI calculator with infix notation
`bg`	y	continue running a process in the background
`cal`		print a calendar
`cat`		catenate standard input or file(s) to standard output
`cc`		compile a C program
`cd`	y	change directory
`chgrp`		change group ownership
`chmod`		change file permission mode
`chown`		change user ownership (administrator only)
`cmp`		compare two files
`compress`		compress files with the LZW algorithm
`cp`		copy files
`cpio`		create an archive of a list of files
`crontab`		show or set scheduled tasks
`cut`		manage columns of text
`date`		show or set the date and time
`dc`		CLI calculator with postfix notation
`dd`		copy and convert data stream
`df`		show available space on file-systems
`diff`		print difference between two textfiles
`dirname`		print only path component of a file
`du`		calculate disk usage of a files or directories
`dump`		make a file-system based backup
`echo`	y	print arguments
`ed`		CLI text editor

command	builtin	description
egrep		search text based extended regular expressions
eval	y	perform globbing and variable expansion, then execute line
exec	y	pass control of the shell to a program
exit	y	terminate the current shell
export	y	make variables available to subshells
expr		evaluate expressions
false		return a non-zero value
fg	y	bring suspended or background job to foreground
fgrep		search text based on fixed substrings
file		determine file content type
find		search files based on constraints
finger		view login information of a (remote) user
fmt		reformat text
ftp		Copy files to and from a network connected computer
gcc		GNU C compiler
grep		search text based on regular expressions
gunzip		unpack a gzip-compressed file
gzip		compress a file with the Lempel-Ziv algorithm
jobs	y	show backgrounded and suspended jobs
join		join two text files based on a common field
kill	y	send a signal to a process or job
ld		combine object files to a single binary
ln		create hard or soft links to files
lpr		print a file to a printer
ls		show directory content
m4		create text output based on a macro language
mail		send or read electronic mail
make		execute commands based on targets and prerequisites
man		show manual page
mesg		control write access to your terminal
mkdir		create a directory
mknod		create device file
more		show text one page at a time
mount		attach a file system to the directory tree
mv		rename or move filenames

command	builtin	description
nawk		same as awk, new syntax
newgrp	y	change default group
nice		start a process with a specified priority
nohup		prevent process hangup after shell exit
nroff		typesetting system for documents
od		octal (and hex and character) dump of a binary file
passwd		change login password
ping		test TCP/IP connectivity with remote host
pr		format text for printing
ps		list processes and their status
pwd	y	print the current directory
read	y	read a line from input and assign it to variables
renice		change the priority of a process
restor		restore files from a backup with dump
rm		remove files
rmdir		remove empty directories
roff		typesetting system for documents
scp		copy files via encrypted network connections
sed		stream editor
set	y	set shell mode and/or assign values to positional variables
shift	y	remove the first (few) positional variables from argument list
sleep	y	do nothing for specified number of seconds
sort		sort lines in a file
split		split a file into multiple parts
ssh		execute commands on remote host via an encrypted connection
strip		remove debugging code from binary executable
stty		set or show tty parameters
su		switch user
sum		calculate checksum
sync		write cache to disks
tail		show last few lines of a text file
tar		create archive of files to file or device
tee		copy standard output to a file
telnet		login to a remote host
test	y	test conditions

command	builtin	description
time		calculate the time that a program runs
touch		renew timestamp of a file
tr		translate or delete text characters
trap	y	execute specified command if signal is received
troff		typesetting system for documents
true		return a zero value
type	y	show path of command according to the shell
tty		show TTY identifier
ulimit	y	show or modify resource limits
umask	y	create mask for permission bits on new files and directories
umount		detach a file-system from the directory tree
uname		print system information
uncompress		uncompress a file (not suitable for gzip-ed files)
uniq		remove duplicate lines
unset	y	remove the value from a shell variable
vi		edit a text in visual mode
wait	y	wait for a process and return its exit value
wall		send a message to all users
wc		count characters, words and lines in a text
which		show path of command according to $PATH variable
who		print logged-in users
write		send a message to another user
xargs		transfer standard input to command line arguments
yes		keep repeating a text until the process is killed
zcat		uncompress files or standard input to standard output

Appendix C: Exercises

The exercises presented here suppose that the reader has already familiarized herself with basic file management like copying, renaming, linking and deleting files and directories. The exercises combine many parts of the material and relevant man pages will have to be examined for the specific options. To set up the lab environment, do the following as an ordinary user on a Linux system:

```
wget http://crash.hamal.nl/exercises.sh -O /tmp/exercises.sh
sh /tmp/exercises.sh
```

Exercise 1:
What is the syntax of the substitute command to use with **sed** to replace all consecutive underscore characters (_) with the text "Mrs. Johnson" in the file "**letter.txt**"?

Exercise 2:
There is a text file named "**secret.txt**" in the directory "**Deep Secret**". What is its content? What is the content of the other file in the same directory?

Exercise 3:
How do you delete the file with the name "**-rf [a-z]***"?

Exercise 4:
If the binary for the **ls** command is unusable, how could you list the content of a directory?

Exercise 5:
What is the command line to delete all files in the "**sizetest**" directory tree where the files exceed 700 kB in size? What is the command line to move all the remaining files in the same directory tree that exceed 200kB in size to the directory "**bigfiles**"?

Exercise 6:
Write a script that will "unlink" all files in its arguments that are hard links, i.e. if the file has multiple links, copy it, delete the original and rename the copy to the original name. For all files with single names, print a message that it was already unlinked. Beware of limited disk space! Use files in the "**dircmp0**" directory for this exercise.

Exercise 7:
Write a script that expects two directory names as arguments. Then calculate the sum of the file sizes of all filenames that are present in the first directory but not in the second (hint: look at the "**-u**" option in the manpage of **uniq**). Use the directories "**dircmp0**" and "**dircmp1**" for this exercise. Note, for summing sizes using **awk**, see the example on page 32.

Solutions to these exercises are available at
`http://crash.hamal.nl/`