Linux Fundamentals

Paul Cobbaut

Linux Fundamentals

Paul Cobbaut

lt-0.970.169

Published Sun 30 Nov 2008 10:09:50 PM CET

Abstract

This book is meant to be used in an instructor-led training. For self-study, the idea is to read this book next to a working Linux computer so you can immediately do every subject, even every command.

This book is aimed towards novice Linux system administrators (and might be interesting and useful for home users that want to know a bit more about their Linux system). However, this book is not meant as an introduction to Linux desktop applications like text editors, browsers, mail clients, multimedia or office applications.

More information and free .pdf available at **http://www.linux-training.be** .

Feel free to contact the authors:

- Paul Cobbaut: paul.cobbaut@gmail.com, http://www.linkedin.com/in/cobbaut

Contributors to the Linux Training project are:

- Serge van Ginderachter: serge@ginsys.be, docbook xml and pdf build scripts; svn hosting
- Hendrik De Vloed: hendrik.devloed@ugent.be, buildheader.pl script

We'd also like to thank our reviewers:

- Wouter Verhelst: wouter@grep.be, http://grep.be
- Geert Goossens: goossens.geert@gmail.com, http://www.linkedin.com/in/geertgoossens
- Elie De Brauwer: elie@de-brauwer.be, http://www.de-brauwer.be
- Christophe Vandeplas: christophe@vandeplas.com, http://christophe.vandeplas.com

Table of Contents

List of Tables

Chapter 1. Introduction to Unix and Linux

1.1. Unix History

1.1.1. AT&T Bell Labs

In 1969 **Dennis Ritchie** and **Ken Thompson** wrote **UNICS** (Uniplexed Information and Computing System) at Bell Labs. Together with **Douglas McIlroy** they are seen as the creators of Unix. The name Unix is a play on the Multics Operating System for large mainframe computers. Unics (later renamed to Unix) was written for mini computers like the DEC PDP-series. In 1973 they decided to write Unix in C (instead of assembler), to make it portable to other computers. Unix was made available to universities, companies and the US government, including the full source code. This meant that every C programmer could make changes. By 1978 about 600 computers were running Unix.

Table 1.1. Early Unix Timeline

<table>
<tr><th>1969-1977</th><th>1978-1980</th><th>1981</th><th>1982</th></tr>
<tr><td rowspan="3">UNIX Time Sharing System</td><td rowspan="2">BSD</td><td rowspan="2">4.1BSD</td><td>4.1BSD</td></tr>
<tr><td>SunOS 1.0</td></tr>
<tr><td colspan="2">Unix</td><td>Unix System III</td></tr>
</table>

1.1.2. The Unix Wars

The unity and openness that existed in the Unix world until 1977 was long gone by the end of the eighties. Different vendors of distinct versions of Unix tried to set the standard. Sun and AT&T joined the **X/Open** group to unify Unix. Other vendors joined the Open Software Foundation or **OSF**. These struggles were not good for Unix, allowing for new operating system families like OS/2, Novell Netware and Microsoft Windows NT to take big chunks of server market share in the early nineties. The table below shows the evolution of a united Unix into several Unixes in the eighties.

Table 1.2. Eighties Unix Timeline

<table>
<tr><th>1983</th><th>1984</th><th>1985</th><th>1986</th><th>1987</th><th>1988</th><th>1989</th><th>1990</th><th>1991</th><th>1992</th></tr>
<tr><td colspan="3" rowspan="2">4.1BSD</td><td colspan="5">4.3BSD</td><td colspan="2">BSD Net/2</td></tr>
<tr><td colspan="2">4.3BSD</td><td colspan="6">NeXTSTEP</td></tr>
<tr><td colspan="3">SunOS1.0</td><td colspan="2">SunOS3.2</td><td colspan="3" rowspan="2">SystemVr4</td><td colspan="2">Solaris</td></tr>
<tr><td colspan="5">System V</td><td colspan="2">UnixWare</td></tr>
<tr><td colspan="3">System V</td><td colspan="7">AIX</td></tr>
<tr><td>III + V</td><td colspan="9">HP-UX</td></tr>
</table>

1.1.3. University of California, Berkeley

Students of Berkeley were happy to join in the development of Bell Labs Unix, but were not so happy with the restrictive licensing. Unix was open source software, but it still required purchase of a license. So during the eighties, they rewrote all the Unix tools, until they had a complete Unix-like operating system. By 1991, the **BSD** (Berkeley Software Distribution) branch of Unix was completely separate from the Bell Labs Unix. **NetBSD**, **FreeBSD** and **OpenBSD** are three current Unix-like operating systems derived from the 1991 **BSD Net/2** codebase. Sun Solaris, Microsoft Windows NT and Apple Mac OS X all used source code from BSD. The table below shows operating systems still in use today that are in a way derived from the 1978-1981 BSD codebase.

Table 1.3. Current BSD Timeline

<table>
<tr><th>1991</th><th>1992</th><th>1993</th><th>1994</th><th>1995</th><th>1996</th><th>1997</th><th>1998</th><th>1999</th><th>2000-2008</th></tr>
<tr><td colspan="2" rowspan="3">BSD Net/2</td><td colspan="8">FreeBSD</td></tr>
<tr><td colspan="8">NetBSD</td></tr>
<tr><td colspan="3">NetBSD</td><td colspan="6">OpenBSD</td></tr>
<tr><td colspan="8">NeXTSTEP</td><td colspan="2">Mac OS X</td></tr>
<tr><td colspan="10">Solaris</td></tr>
</table>

1.1.4. GNU's not Unix

Largely because of unhappiness with the restrictive licensing on existing Unix implementations, **Richard Stallman** initiated the **GNU Project** in 1983. The GNU project aims to create free software. Development of the GNU operating system started, aiming to create a complete Unix-like branch, seperate from the two other (BSD and Bell Labs). Today the GNU compiler **gcc** and most other GNU utilities (like **bash**) are among the most popular on many Unix-like systems. The official kernel of this project is **GNU/Hurd**, but you can hardly call that kernel a finished product.

1.1.5. Linux

Where **GNU/Hurd** failed, the Linux kernel succeeded! In 1991 a Finnish student named **Linus Torvalds** started writing his own operating system for his intel 80386 computer. In January 1992, Linus decided to release Linux under the GNU GPL. Thanks to this, thousands of developers are now working on the Linux kernel. Linus Torvalds is in charge of the kernel developers.

Contrary to popular believe, they are not all volunteers. Today big companies like Red Hat, Novell, IBM, Intel, SGI, Oracle, Montavista, Google, HP, NetApp, Cisco, Fujitsu, Broadcom and others are actively paying developers to work on the Linux kernel. According to the Linux Foundation "over 3700 individual developers from over 200 different companies have contributed to the kernel between 2005 and april 2008". 1057 developers from 186 different companies contributed code to make kernel version 2.6.23 into 2.6.24.

1.2. Licensing

1.2.1. Proprietary

Some flavors of Unix, like HP-UX, IBM AIX and Sun Solaris 9 are delivered after purchase in binary form. You are not authorized to install or use these without paying a license to the owner. You are not authorized to distribute these copies to other people, and you are not authorized to look at or change the closed source code of the operating system. This software is usually protected by copyright, patents and extensive software licensing.

1.2.2. BSD

BSD style licenses are close to the public domain. They essentially state that you can copy the software, but you have to leave the copyright notice that refers to BSD. This license gives a lot of freedom, but offers few protection to someone copying and selling your work.

1.2.3. GNU General Public License (GPL)

More and more software is being released under the **GPL** (in 2006 Java was released under the GPL). The goal of the GPL is to guarantee that free software stays free. Everyone can work together on GPL software, knowing that the software will be freely available to everyone. The GPL can protect software, even in court.

Free as in **freedom of speech**, not to be confused with free as in not having to pay for your free beer. In other words, or even better, in other languages free software

translates to **vrije software** (Dutch) or **Logiciel Libre** (French). Whereas the free from free beer translates to gratis.

Briefly explained, the GPL allows you to copy software, the GPL allows you to distribute (sell or give away) that software, and the GPL grants you the right to read and change the source code. But the person receiving or buying the software from you has the same rights. And also, should you decide to distribute modified versions of GPL software, then you are obligated to put the same license on the modifications (and provide the source code of your modifications).

1.2.4. Others...

There are many other licenses on software. You should read and understand them before using any software.

1.3. Current Distributions

1.3.1. What is a distribution ?

Unix comes in many flavors, usually called **distributions**. A distribution (or in short distro) is a collection of software packages, distributed on CD, online or pre-installed on computers. All the software in a distribution is tested and integrates nicely into a whole. Software is maintained (patched) by the distributor, and is managed by an **integrated package manager**. Many distro's have a central **repository of approved software**. Installing software from outside the distro can sometimes be cumbersome and may void your warranty on the system.

1.3.2. Linux Distributions

There are hundreds of Linux distributions, just take a look at the distrowatch.com website. For many years, Red Hat, Suse and Mandrake were considered the big three for end users. Red Hat is still the biggest commercial Linux vendor, but in 2008, the most popular Linux distribution for home users is **Ubuntu** from Canonical.

1.3.2.1. Linux distribution detection

Depending on the distribution used, there are distinct files that contain the distribution version.

The **/etc/redhat-release** file contains the Red Hat version on most of the Red Hat and Red Hat derived systems. Debian and Ubuntu systems contain **/etc/debian-version**. Note that Ubuntu was originally derived from Debian.

```
paul@RHELv4u4:~$ cat /etc/redhat-release
```

```
Red Hat Enterprise Linux AS release 4 (Nahant Update 4)

serge@venusia:~$ cat /etc/debian_version
lenny/sid
```

The **/etc/lsb-release** file can be found on distributions that follow the Linux Standard Base. Other variations to these files are **/etc/slackware-version**, **/etc/SuSE-release**, **/etc/gentoo-release** and **/etc/mandrake-release**.

```
serge@venusia:~$ cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=8.04
DISTRIB_CODENAME=hardy
DISTRIB_DESCRIPTION="Ubuntu 8.04.1"
```

1.3.2.2. Red Hat

Redhat exists as a company since 1993. They distribute **Red Hat Enterprise Linux** (RHEL) to companies and manage the **Fedora** project. RHEL is probably the most popular Linux-based distro on servers. Fedora is a very popular and user friendly Linux-based distro, aimed towards home users. The company makes a profit of around one hundred million dollars a year, selling support contracts. Red Hat contributes a lot to the Linux kernel and other free software projects.

1.3.2.2.1. Red Hat Linux

Red Hat Linux was distributed from 1994 until 2003. It was one of the oldest common Linux distributions. Red Hat Linux was the first distro to use the **rpm** package format. Many other distro's are originally derived from Red Hat Linux. The company **Red Hat, Inc.** decided to split Red Hat Linux into **Fedora** and **Red Hat Enterprise Linux**.

1.3.2.2.2. Fedora

Fedora is sponsored by Red Hat, and is aimed toward home users. There is no official support from Red Hat. Every six to eight months, there is a new version of Fedora. Fedora usually has more recent versions of kernel and applications than RHEL. Fedora 9 was released May 2008.

1.3.2.2.3. Red Hat Enterprise Linux 4

Since 2005 Red Hat distributes four different RHEL4 variants. **RHEL AS** is for mission-critical computer systems. **RHEL ES** is for small to mid-range servers. **RHEL WS** is for technical power user desktops and critical design. **Red Hat Desktop** is for multiple deployments of single user desktops. Red Hat does not give an explanation for the meaning of AS, ES and WS, but it might be Advanced Server, Entry-level Server and Workstation.

1.3.2.2.4. Red Hat Enterprise Linux 5

Red Hat Enterprise Linux version 5 is available since March 2007. One of the notable new features is the inclusion of **Xen**. Xen is a free virtual machine application that allows NetBSD and Linux to serve as host for guest OS'ses. Beyond just virtualization, RHEL 5 also has better SELinux support, clustering, network storage and smartcard integration.

1.3.2.2.5. CentOS and Unbreakable Linux

Both **CentOS** and Oracle's **Unbreakable Linux** are directly derived from RHEL, but all references to Red Hat trademarks are removed. Companies are allowed to do this (GPL), and are hoping to make a profit selling support (without having the cost to maintain and develop their own distribution). Red Hat is not really worried about this, since they develop a lot on Linux, and thus can offer much better support. The Oracle offer however is still very recent, let's wait and see how many organizations will buy a complete solution from Oracle.

1.3.2.3. Ubuntu

Ubuntu is a rather new distribution, based on **Debian** and funded by South African developer and billionaire astronaut **Mark Shuttleworth**. Ubuntu is giving away free (as in beer and speech) CD's with **Ubuntu, Linux for Human Beings**. Many people consider Ubuntu to be the most user friendly Linux distribution. The company behind Ubuntu is **Canonical**, they aim to make a profit of selling support soon. Ubuntu is probably the most popular Unix-like distribution on personal desktops.

Image copied from **xkcd.com**.

1.3.2.4. Novell Suse

A couple of years ago, **Novell** bought the German company **Suse**. They are seen as the main competitor to Red Hat with their SLES (Suse Linux Enterprise Server) and SLED (Suse Linux Enterprise Desktop) versions of Suse Linux. Similar to Fedora,

Novell hosts the **OpenSUSE** project as a testbed for upcoming SLED and SLES releases.

Novell has signed a very controversial deal with Microsoft. Some high profile open source developers have left the company because of this agreement, and many people from the open source community are actively advocating to abandon Novell completely.

1.3.2.5. Debian

Debian is one of the most secure Linux distro's. It is known to be stable and reliable. The Debian people also have a strong focus towards freedom. You will not find patented technologies or non-free software in the standard Debian repositories. A lot of distributions (Ubuntu, Knoppix, ...) are derived from the Debian codebase. Debian has **aptitude**, which is considered the best package management system.

1.3.2.6. Mandriva

Mandriva is the unification of the Brazilian distro Conectiva with the French distro Mandrake. They are considered a user friendly distro, with support from the French government.

1.3.3. BSD Distributions

1.3.3.1. FreeBSD

FreeBSD is a complete operating system. The kernel and all of the utilities are held in the same source code tree. FreeBSD runs on many architectures and is considered to be reliable and robust. Millions of websites are running on FreeBSD, including some big like yahoo.com, apache.org, sony.co.jp, netcraft, php.net, freebsd.org and (until last year) ftp.cdrom.com. Apple's MacOSX contains the FreeBSD virtual file system, network stack and more.

1.3.3.2. NetBSD

NetBSD development started around the same time (1993) as FreeBSD. NetBSD aims for maximum portability and thus runs on many architectures. NetBSD is often used in embedded devices.

1.3.3.3. OpenBSD

Co-founder **Theo De Raadt** from NetBSD founded the **OpenBSD** project in 1994. OpenBSD aims for maximum security. The past ten years, only two vulnerabilities

were found in the default install of OpenBSD. All source code is thoroughly checked. OpenBSD runs on sixteen different architectures and is commonly used for firewalls and IDS. The OpenBSD people also bring us **OpenSSH**.

1.3.4. Major Vendors of Unix

We should at least mention IBM's **AIX**, Sun's **Solaris** and Hewlett-Packards **HP-UX**, all are based on the original Unix from Bell Labs (Unix System V). Sun's **SunOS**, HP's **Tru64** (originally from DEC) and Apple's **MacOSX** are more derived from the BSD branch. But most Unixes today may contain source code and implementations from both original Unix-branches.

1.3.5. Solaris

1.3.5.1. Solaris 8 and Solaris 9

All **Sun Solaris** releases before Solaris 10 are proprietary binary only, just like IBM AIX and HP-UX.

1.3.5.2. Solaris 10

Solaris 10 is the officialy supported Sun distribution. It is a free (as in beer) download. Sun releases binary patches and updates. Sun would like a community built around the solaris kernel, similar to the Linux community. Sun released the Solaris kernel under the CDDL, a license similar to the GPL, hoping this will happen.

1.3.5.3. Nevada and Solaris Express

Nevada is the codename for the next release of Solaris (Solaris 11). It is currently under development by Sun and is based on the OpenSolaris code. Solaris Express Community Edition is an official free binary release including open source OpenSolaris and some closed source technologies, updated twice a month without any support from Sun. Solaris Express Developer Edition is the same, but with some support, thorough testing before release, and released twice a year.

1.3.5.4. OpenSolaris, Belenix and Nexenta

OpenSolaris is een open source development project (yes, it is only source code). Future versions of the Solaris operating system are based on this source code. The **Belenix** LiveCD is based on OpenSolaris. Another famous opensolaris based distro is **Nexenta**. Nexenta (www.gnusolaris.org) looks like **Ubuntu** and feels like **Debian**. The goal of this **GNU/Solaris** project is to have the best Linux desktop (Ubuntu) including the **aptitude** package manager running on a Sun Solaris kernel.

1.4. Certification

1.4.1. LPI: Linux Professional Institute

1.4.1.1. LPIC Level 1

This is the junior level certification. You need to pass exams 101 and 102 to achieve **LPIC 1 certification**. To pass level one, you will need Linux command line, user management, backup and restore, installation, networking and basic system administration skills.

1.4.1.2. LPIC Level 2

This is the advanced level certification. You need to be LPIC 1 certified and pass exams 201 and 202 to achieve **LPIC 2 certification**. To pass level two, you will need to be able to administer medium sized Linux networks, including Samba, mail, news, proxy, firewall, web and ftp servers.

1.4.1.3. LPIC Level 3

This is the senior level certification. It contains one core exam (301) which tests advanced skills mainly about ldap. To achieve this level you also need LPIC Level 2 and pass a specialty exam (302 or 303). Exam 302 mainly focuses on Samba, 303 is about advanced security. More info on http://www.lpi.org.

1.4.1.4. Ubuntu

When you are LPIC Level 1 certified, you can take a LPI Ubuntu exam (199) and become Ubuntu certified.

1.4.2. Red Hat Certified Engineer

The big difference with most certs is that there are no multiple choice questions for **RHCE**. Red Hat Certified Engineers have taken a live exam consisting of two parts. First they have to troubleshoot and maintain an existing but broken setup (scoring at least 80 percent), second they have to install and configure a machine (scoring at least 70 percent).

1.4.3. MySQL

There are two tracks for MySQL certification; Certified MySQL 5.0 Developer (CMDEV) and Certified MySQL 5.0 DBA (CMDBA). The **CMDEV** is focused at

database application developers, the **CMDBA** is for database administrators. Both tracks require two exams each. The MySQL cluster DBA certification requires CMDBA certification and passing the CMCDBA exam.

1.4.4. Novell CLP/CLE

To become a **Novell Certified Linux Professional**, you have to take a live practicum. This is a VNC session to a set of real SLES servers. You have to perform several tasks and are free to choose your method (commandline or YaST or ...). No multiple choice involved.

1.4.5. Sun Solaris

Sun uses the classical formula of multiple choice exams for certification. Passing two exams for an operating system gets you the Solaris Certified Administrator for Solaris X title.

1.4.6. Other certifications

There are many other less known certs like EC council's Certified Ethical Hacker, CompTIA's Linux+ and Sair's Linux GNU.

1.5. Where to find help ?

1.5.1. Manual Pages

Most Unix tools and commands have pretty good man pages. Type **man** followed by a command (for which you want help) and start reading. Ah, and press **q** to quit the manpage.

```
paul@laika:~$ man whois
Reformatting whois(1), please wait...
paul@laika:~$
```

Manpages can be useful when you are switching a lot between different flavors of unix, to find those little differences in commands. Very often manpages also describe configuration files and daemons.

```
paul@laika:~$ man syslog.conf
Reformatting syslog.conf(5), please wait...
paul@laika:~$ man syslogd
Reformatting syslogd(8), please wait...
```

The **man -k** command (same as **apropos**) will show you a list of manpages containing your searchstring.

```
paul@laika:~$ man -k syslog
lm-syslog-setup (8)  - configure laptop mode to switch syslog.conf ...
logger (1)           - a shell command interface to the syslog(3) ...
syslog-facility (8)  - Setup and remove LOCALx facility for sysklogd
syslog.conf (5)      - syslogd(8) configuration file
syslogd (8)          - Linux system logging utilities.
syslogd-listfiles (8) - list system logfiles
paul@laika:~$
```

By now you will have noticed the numbers between the round brackets. **man man** will explain to you that these are section numbers. If you want to know more, RTFM (Read The Fantastic Manual). *Unfortunately, manual pages do not have the answer to everything...*

```
paul@laika:~$ man woman
No manual entry for woman
```

1.5.2. Red Hat Manuals online

Red Hat has a lot of info online at http://www.redhat.com/docs/manuals/ in both pdf and html format. *Unfortunately, the information there is not always up to date.*

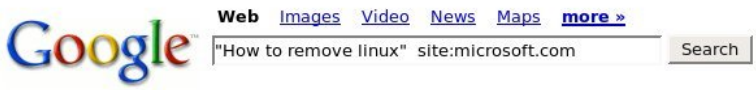
1.5.3. Searching the internet with Google

Google is a powerful tool to find help about Unix, or anything else. Here are some tricks.

Look for phrases instead of single words.

Search only pages from the .be TLD (or substitute .be for any other Top Level Domain). You can also use "country:be" to search only pages from Belgium (based on ip rather than TLD).

Search for pages inside one domain



Search for pages **not** containing some words.

1.5.4. Wikipedia

Wikipedia is a web-based, free-content encyclopedia. Its growth the past two years has been astonishing. You have a good chance of finding a clear explanation by typing your search term behind **http://en.wikipedia.org/wiki/** like this example shows.

1.5.5. The Linux Documentation Project

On **www.tldp.org** you will find a lot of documentation, faqs, howtos and man pages about Linux and many other programs running on Linux.

1.5.6. This book

This book is available in .pdf format. Download it at http://www.linux-training.be .

1.6. Discovering the classroom

It is time now to take a look at what we have in this classroom. Students should be able to log on to one or more (virtual) Linux computers and test connectivity to each other and to the internet.

Chapter 2. First Steps

2.1. Working with directories

To explore the Linux filetree, you will need some tools. Here's a small overview of the most common commands to work with directories. These commands are available on any Linux system.

2.1.1. pwd

The **you are here** sign can be displayed with the **pwd** command (Print Working Directory). Go ahead, try it: open a commandline interface (like gnome-terminal, konsole, xterm or a tty) and type **pwd**. The tool displays your **current directory**.

```
paul@laika:~$ pwd
/home/paul
```

2.1.2. cd

You can change your current directory with the **cd** command (Change Directory).

```
paul@laika$ cd /etc
paul@laika$ pwd
/etc
paul@laika$ cd /bin
paul@laika$ pwd
/bin
paul@laika$ cd /home/paul/
paul@laika$ pwd
/home/paul
```

2.1.2.1. cd ~

You can pull off a trick with cd. Just typing **cd** without a target directory, will put you in your home directory. Typing **cd ~** has the same effect.

```
paul@laika$ cd /etc
paul@laika$ pwd
/etc
paul@laika$ cd
paul@laika$ pwd
/home/paul
paul@laika$ cd ~
paul@laika$ pwd
/home/paul
```

2.1.2.2. cd ..

To go to the **parent directory** (the one just above your current directory in the directory tree), type **cd ..** .

```
paul@laika$ pwd
/usr/share/games
paul@laika$ cd ..
paul@laika$ pwd
/usr/share
paul@laika$ cd ..
paul@laika$ cd ..
paul@laika$ pwd
/
```

*To stay in the current directory, type **cd .** ;-)*

2.1.2.3. cd -

Another useful shortcut with cd is to just type **cd -** to go to the previous directory.

```
paul@laika$ pwd
/home/paul
paul@laika$ cd /etc
paul@laika$ pwd
/etc
paul@laika$ cd -
/home/paul
paul@laika$ cd -
/etc
```

2.1.2.4. absolute and relative paths

You should be aware of **absolute and relative paths** in the filetree. When you type a path starting with a slash, then the root of the filetree is assumed. If you don't start your path with a slash, then the current directory is the assumed starting point.

The screenshot below first shows the current directory (/home/paul). From within this directory, you have to type **cd /home** instead of **cd home** to go to the /home directory.

```
paul@laika$ pwd
/home/paul
paul@laika$ cd home
bash: cd: home: No such file or directory
paul@laika$ cd /home
paul@laika$ pwd
/home
```

When inside /home, you have to type **cd paul** instead of **cd /paul** to enter the subdirectory paul of the current directory /home.

```
paul@laika$ pwd
/home
paul@laika$ cd /paul
bash: cd: /paul: No such file or directory
paul@laika$ cd paul
paul@laika$ pwd
/home/paul
```

In case your current directory is the root directory, then both **cd /home** and **cd home** will get you in the /home directory.

```
paul@laika$ cd /
paul@laika$ pwd
/
paul@laika$ cd home
paul@laika$ pwd
/home
paul@laika$ cd /
paul@laika$ pwd
/
paul@laika$ cd /home
paul@laika$ pwd
/home
```

This was the last screenshot with pwd statements. From now on, the current directory will often be displayed in the prompt. We will explain later in this book, how the shell variable $PS1 can be configured to do this.

2.1.3. ls

You can list the contents of a directory with **ls**.

```
paul@pasha:~$ ls
allfiles.txt  dmesg.txt  httpd.conf  stuff  summer.txt
paul@pasha:~$
```

2.1.3.1. ls -a

A frequently used option with ls is **-a** to show all files. All files means including the **hidden files**. When a filename on a Unix file system starts with a dot, it is considered a hidden file, and it doesn't show up in regular file listings.

```
paul@pasha:~$ ls
allfiles.txt  dmesg.txt  httpd.conf  stuff  summer.txt
```

```
paul@pasha:~$ ls -a
.   allfiles.txt   .bash_profile  dmesg.txt   .lesshst  stuff
..  .bash_history  .bashrc        httpd.conf  .ssh      summer.txt
paul@pasha:~$
```

2.1.3.2. ls -l

Many times you will be using options with ls to display the contents of the directory in different formats, or to display different parts of the directory. Just typing ls gives you a list of files in the directory. Typing **ls -l** (that is a letter L, not the number 1) gives you a long listing (more information on the contents).

```
paul@pasha:~$ ls -l
total 23992
-rw-r--r-- 1 paul paul 24506857 2006-03-30 22:53 allfiles.txt
-rw-r--r-- 1 paul paul    14744 2006-09-27 11:45 dmesg.txt
-rw-r--r-- 1 paul paul     8189 2006-03-31 14:01 httpd.conf
drwxr-xr-x 2 paul paul     4096 2007-01-08 12:22 stuff
-rw-r--r-- 1 paul paul        0 2006-03-30 22:45 summer.txt
```

2.1.3.3. ls -lh

Another frequently used ls option is **-h**. It shows the numbers (file sizes) in a more human readable format. Also shown below is some variation in the way you can give the options to ls. We will explain the details of the output later in this book.

```
paul@pasha:~$ ls -l -h
total 24M
-rw-r--r-- 1 paul paul  24M 2006-03-30 22:53 allfiles.txt
-rw-r--r-- 1 paul paul  15K 2006-09-27 11:45 dmesg.txt
-rw-r--r-- 1 paul paul 8.0K 2006-03-31 14:01 httpd.conf
drwxr-xr-x 2 paul paul 4.0K 2007-01-08 12:22 stuff
-rw-r--r-- 1 paul paul    0 2006-03-30 22:45 summer.txt
paul@pasha:~$ ls -lh
total 24M
-rw-r--r-- 1 paul paul  24M 2006-03-30 22:53 allfiles.txt
-rw-r--r-- 1 paul paul  15K 2006-09-27 11:45 dmesg.txt
-rw-r--r-- 1 paul paul 8.0K 2006-03-31 14:01 httpd.conf
drwxr-xr-x 2 paul paul 4.0K 2007-01-08 12:22 stuff
-rw-r--r-- 1 paul paul    0 2006-03-30 22:45 summer.txt
paul@pasha:~$ ls -hl
total 24M
-rw-r--r-- 1 paul paul  24M 2006-03-30 22:53 allfiles.txt
-rw-r--r-- 1 paul paul  15K 2006-09-27 11:45 dmesg.txt
-rw-r--r-- 1 paul paul 8.0K 2006-03-31 14:01 httpd.conf
drwxr-xr-x 2 paul paul 4.0K 2007-01-08 12:22 stuff
-rw-r--r-- 1 paul paul    0 2006-03-30 22:45 summer.txt
paul@pasha:~$ ls -h -l
total 24M
-rw-r--r-- 1 paul paul  24M 2006-03-30 22:53 allfiles.txt
-rw-r--r-- 1 paul paul  15K 2006-09-27 11:45 dmesg.txt
-rw-r--r-- 1 paul paul 8.0K 2006-03-31 14:01 httpd.conf
drwxr-xr-x 2 paul paul 4.0K 2007-01-08 12:22 stuff
-rw-r--r-- 1 paul paul    0 2006-03-30 22:45 summer.txt
```

2.1.4. mkdir

Walking around the Unix filetree is fun, but it is even more fun to create your own directories with **mkdir**. You have to give at least one parameter to **mkdir**, the name of the new directory to be created. Think before you type a leading / .

```
paul@laika:~$ mkdir MyDir
paul@laika:~$ cd MyDir
paul@laika:~/MyDir$ ls -al
total 8
drwxr-xr-x  2 paul paul 4096 2007-01-10 21:13 .
drwxr-xr-x 39 paul paul 4096 2007-01-10 21:13 ..
paul@laika:~/MyDir$ mkdir stuff
paul@laika:~/MyDir$ mkdir otherstuff
paul@laika:~/MyDir$ ls -l
total 8
drwxr-xr-x 2 paul paul 4096 2007-01-10 21:14 otherstuff
drwxr-xr-x 2 paul paul 4096 2007-01-10 21:14 stuff
paul@laika:~/MyDir$
```

2.1.4.1. mkdir -p

When given the option **-p**, then mkdir will create parent directories as needed.

```
paul@laika:~$ mkdir -p MyDir2/MySubdir2/ThreeDeep
paul@laika:~$ ls MyDir2
MySubdir2
paul@laika:~$ ls MyDir2/MySubdir2
ThreeDeep
paul@laika:~$ ls MyDir2/MySubdir2/ThreeDeep/
```

2.1.5. rmdir

When a directory is empty, you can use **rmdir** to remove the directory.

```
paul@laika:~/MyDir$ rmdir otherstuff
paul@laika:~/MyDir$ ls
stuff
paul@laika:~/MyDir$ cd ..
paul@laika:~$ rmdir MyDir
rmdir: MyDir/: Directory not empty
paul@laika:~$ rmdir MyDir/stuff
paul@laika:~$ rmdir MyDir
```

2.1.5.1. rmdir -p

And similar to the mkdir -p option, you can also use rmdir to recursively remove directories.

```
paul@laika:~$ mkdir -p dir/subdir/subdir2
```

```
paul@laika:~$ rmdir -p dir/subdir/subdir2
paul@laika:~$
```

2.2. Practice: Working with directories

1. Display your current directory.

2. Change to the /etc directory.

3. Now change to your home directory using only three key presses.

4. Change to the /boot/grub directory using only eleven key presses.

5. Go to the parent directory of the current directory.

6. Go to the root directory.

7. List the contents of the root directory.

8. List a long listing of the root directory.

9. Stay where you are, and list the contents of /etc.

10. Stay where you are, and list the contents of /bin and /sbin.

11. Stay where you are, and list the contents of ~.

12. List all the files (including hidden files) in your homedirectory.

13. List the files in /boot in a human readable format.

14. Create a directory testdir in your homedirectory.

15. Change to the /etc directory, stay here and create a directory newdir in your homedirectory.

16. Create in one command the directories ~/dir1/dir2/dir3 (dir3 is a subdirectory from dir2, and dir2 is a subdirectory from dir1).

17. Remove the directory testdir.

18. If time permits (or if you are waiting for other students to finish this practice), use and understand pushd and popd. Use the man page of bash to find information about pushd, popd and dirs.

2.3. Solution: Working with directories

1. Display your current directory.

```
pwd
```

2. Change to the /etc directory.

```
cd /etc
```

3. Now change to your home directory using only three key presses.

```
cd (and the enter key)
```

4. Change to the /boot/grub directory using only eleven key presses.

```
cd /boot/grub (use the tab key)
```

5. Go to the parent directory of the current directory.

```
cd .. (with space between cd and ..)
```

6. Go to the root directory.

```
cd /
```

7. List the contents of the root directory.

```
ls
```

8. List a long listing of the root directory.

```
ls -l
```

9. Stay where you are, and list the contents of /etc.

```
ls /etc
```

10. Stay where you are, and list the contents of /bin and /sbin.

```
ls /bin /sbin
```

11. Stay where you are, and list the contents of ~.

```
ls ~
```

12. List all the files (including hidden files) in your homedirectory.

```
ls -al ~
```

13. List the files in /boot in a human readable format.

```
ls -lh /boot
```

14. Create a directory testdir in your homedirectory.

```
mkdir ~/testdir
```

15. Change to the /etc directory, stay here and create a directory newdir in your homedirectory.

```
cd /etc ; mkdir ~/newdir
```

16. Create in one command the directories ~/dir1/dir2/dir3 (dir3 is a subdirectory from dir2, and dir2 is a subdirectory from dir1).

```
mkdir -p ~/dir1/dir2/dir3
```

17. Remove the directory testdir.

```
rmdir testdir
```

18. If time permits (or if you are waiting for other students to finish this practice), use and understand pushd and popd. Use the man page of bash to find information about pushd, popd and dirs.

```
man bash
```

The Bash shell has two built-in commands called **pushd** and **popd**. Both commands work with a common stack of previous directories. Pushd adds a directory to the stack and changes to a new current directory, popd removes a directory from the stack and sets the current directory.

```
paul@laika:/etc$ cd /bin
paul@laika:/bin$ pushd /lib
/lib /bin
paul@laika:/lib$ pushd /proc
/proc /lib /bin
paul@laika:/proc$
paul@laika:/proc$ popd
/lib /bin
paul@laika:/lib$
paul@laika:/lib$
paul@laika:/lib$ popd
/bin
paul@laika:/bin$
```

2.4. Working with files

2.4.1. file

The **file** utility determines the file type. Linux does not use extensions to determine the file type. Your editor does not care whether a file ends in .TXT or .DOC. As a system administrator, you should use the **file** command to determine the file type. First some examples on a typical Linux system.

```
paul@laika:~$ file pic33.png
pic33.png: PNG image data, 3840 x 1200, 8-bit/color RGBA, non-interlaced
paul@laika:~$ file /etc/passwd
/etc/passwd: ASCII text
paul@laika:~$ file HelloWorld.c
HelloWorld.c: ASCII C program text
```

Here's another example of the file utility. It shows the different type of binaries on different architectures.

```
# Solaris 9 on Intel
bash-2.05$ file /bin/date
/bin/date:      ELF 32-bit LSB executable 80386 Version 1, dynamically \
linked, stripped

# Ubuntu Linux on AMD64
paul@laika:~$ file /bin/date
/bin/date: ELF 64-bit LSB executable, AMD x86-64, version 1 (SYSV), for\
 GNU/Linux 2.6.0, dynamically linked (uses shared libs), for GNU/Linux \
2.6.0, stripped

# Debian Sarge on SPARC
paul@pasha:~$ file /bin/date
/bin/date: ELF 32-bit MSB executable, SPARC, version 1 (SYSV), for GNU/\
Linux 2.4.1, dynamically linked (uses shared libs), for GNU/Linux 2.4.1\
, stripped
```

The file command uses a magic file that contains patterns to recognize filetypes. The magic file is located in **/usr/share/file/magic**. Type **man 5 magic** for more information.

2.4.2. touch

One easy way to create a file is with **touch**. (We will see many other ways for creating files later in this book.)

```
paul@laika:~/test$ touch file1
paul@laika:~/test$ touch file2
paul@laika:~/test$ touch file555
paul@laika:~/test$ ls -l
total 0
-rw-r--r-- 1 paul paul 0 2007-01-10 21:40 file1
-rw-r--r-- 1 paul paul 0 2007-01-10 21:40 file2
-rw-r--r-- 1 paul paul 0 2007-01-10 21:40 file555
```

2.4.2.1. touch -t

Of course, touch can do more than just create files. Can you find out what by looking at the next screenshot ? If not, check the manual of touch.

```
paul@laika:~/test$ touch -t 200505050000 SinkoDeMayo
paul@laika:~/test$ touch -t 130207111630 BigBattle
paul@laika:~/test$ ls -l
total 0
-rw-r--r-- 1 paul paul 0 1302-07-11 16:30 BigBattle
-rw-r--r-- 1 paul paul 0 2005-05-05 00:00 SinkoDeMayo
```

2.4.3. rm

When you no longer need a file, use **rm** to remove it. Unlike some graphical user interfaces, the command line in general does not have a *waste bin* or *trashcan* to recover files. When you use rm to remove a file, the file is gone. So be careful before removing files!

```
paul@laika:~/test$ ls
BigBattle  SinkoDeMayo
paul@laika:~/test$ rm BigBattle
paul@laika:~/test$ ls
SinkoDeMayo
```

2.4.3.1. rm -i

To prevent yourself from accidentally removing a file, you can type **rm -i**.

```
paul@laika:~/Linux$ touch brel.txt
paul@laika:~/Linux$ rm -i brel.txt
rm: remove regular empty file `brel.txt'? y
paul@laika:~/Linux$
```

2.4.3.2. rm -rf

By default, rm will not remove non-empty directories. However rm accepts several options that will allow you to remove any directory. The **rm -rf** statement is famous because it will erase anything (providing that you have the permissions to do so). When you are logged on as root, be very careful with **rm -rf** (the **f** means **force** and the **r** means **recursive**), because being root implies that permissions don't apply to you, so you can literally erase your entire system by accident.

```
paul@laika:~$ ls test
SinkoDeMayo
paul@laika:~$ rm test
rm: cannot remove `test': Is a directory
paul@laika:~$ rm -rf test
paul@laika:~$ ls test
ls: test: No such file or directory
```

2.4.4. cp

To copy a file, use **cp** with a source and a target argument. If the target is a directory, then the sourcefiles are copied in that target directory.

```
paul@laika:~/test$ touch FileA
paul@laika:~/test$ ls
```

```
FileA
paul@laika:~/test$ cp FileA FileB
paul@laika:~/test$ ls
FileA  FileB
paul@laika:~/test$ mkdir MyDir
paul@laika:~/test$ ls
FileA  FileB  MyDir
paul@laika:~/test$ cp FileA MyDir/
paul@laika:~/test$ ls MyDir/
FileA
```

2.4.4.1. cp -r

To copy complete directories, use **cp -r** (the **-r** option forces **recursive** copying of all files in all subdirectories).

```
paul@laika:~/test$ ls
FileA  FileB  MyDir
paul@laika:~/test$ ls MyDir/
FileA
paul@laika:~/test$ cp -r MyDir MyDirB
paul@laika:~/test$ ls
FileA  FileB  MyDir  MyDirB
paul@laika:~/test$ ls MyDirB
FileA
```

2.4.4.2. cp multiple files to directory

You can also use cp to copy multiple file into a directory. In that case, the last argument (aka the target) must be a directory.

```
cp file1 file2 dir1/file3 dir1/file55 dir2
```

2.4.4.3. cp -i

To prevent cp from overwriting existing files, use the -i (for interactive) option.

```
paul@laika:~/test$ cp fire water
paul@laika:~/test$ cp -i fire water
cp: overwrite `water'? no
paul@laika:~/test$
```

2.4.4.4. cp -p

To preserve permissions and time stamps from source files, use **cp -p**.

```
paul@laika:~/perms$ cp file* cp
paul@laika:~/perms$ cp -p file* cpp
paul@laika:~/perms$ ll *
```

```
-rwx------ 1 paul paul    0 2008-08-25 13:26 file33
-rwxr-x--- 1 paul paul    0 2008-08-25 13:26 file42

cp:
total 0
-rwx------ 1 paul paul 0 2008-08-25 13:34 file33
-rwxr-x--- 1 paul paul 0 2008-08-25 13:34 file42

cpp:
total 0
-rwx------ 1 paul paul 0 2008-08-25 13:26 file33
-rwxr-x--- 1 paul paul 0 2008-08-25 13:26 file42
```

2.4.5. mv

Use **mv** to rename a file, or to move the file to another directory.

```
paul@laika:~/test$ touch file100
paul@laika:~/test$ ls
file100
paul@laika:~/test$ mv file100 ABC.txt
paul@laika:~/test$ ls
ABC.txt
paul@laika:~/test$
```

When you need to rename only one file, then **mv** is the prefered command to use.

2.4.6. rename

The **rename** command can also be used, but it has a more complex syntax to enable renaming of many files at once. Below two examples, the first switches all occurrences of txt in png for all filenames ending in .txt. The second example switches all occurrences of uppercase ABC in lowercase abc for all filenames ending in .png . The following syntax will work on debian and ubuntu (prior to Ubuntu 7.10).

```
paul@laika:~/test$ ls
123.txt  ABC.txt
paul@laika:~/test$ rename 's/txt/png/' *.txt
paul@laika:~/test$ ls
123.png  ABC.png
paul@laika:~/test$ rename 's/ABC/abc/' *.png
paul@laika:~/test$ ls
123.png  abc.png
paul@laika:~/test$
```

On Red Hat Enterprise Linux (and many other Linux distro's like Ubuntu 8.04), the syntax of rename is a bit different. The first example below renames all *.conf files, replacing any occurrence of conf with bak. The second example renames all(*) files, replacing one with ONE.

```
[paul@RHEL4a test]$ ls
```

```
one.conf  two.conf
[paul@RHEL4a test]$ rename conf bak *.conf
[paul@RHEL4a test]$ ls
one.bak  two.bak
[paul@RHEL4a test]$ rename one ONE *
[paul@RHEL4a test]$ ls
ONE.bak  two.bak
[paul@RHEL4a test]$
```

2.5. Practice: Working with files

1. List the files in the /bin directory

2. Display the type of file of /bin/cat, /etc/passwd and /usr/bin/passwd.

3a. Download wolf.jpg and book.pdf from http://www.linux-training.be (wget http://www.linux-training.be/studentfiles/wolf.jpg)

3b. Display the type of file of wolf.jpg and book.pdf

3c. Rename wolf.jpg to wolf.pdf (use mv).

3d. Display the type of file of wolf.pdf and book.pdf.

4. Create a directory ~/touched and enter it.

5. Create the files today.txt and yesterday.txt in touched.

6. Change the date on yesterday.txt to match yesterday's date.

7. Copy yesterday.txt to copy.yesterday.txt

8. Rename copy.yesterday.txt to kim

9. Create a directory called ~/testbackup and copy all files from ~/touched in it.

10. Use one command to remove the directory ~/testbackup and all files in it.

11. Create a directory ~/etcbackup and copy all *.conf files from /etc in it. Did you include all subdirectories of /etc ?

12. Use rename to rename all *.bak files to *.backup . (if you have more than one distro available, try it on all!)

2.6. Solution: Working with files

1. List the files in the /bin directory

```
ls /bin
```

2. Display the type of file of /bin/cat, /etc/passwd and /usr/bin/passwd.

```
file /bin/cat /etc/passwd /usr/bin/passwd
```

3a. Download wolf.jpg and book.pdf from http://www.linux-training.be (wget http://www.linux-training.be/studentfiles/wolf.jpg)

```
wget http://www.linux-training.be/studentfiles/wolf.jpg
```

```
wget http://www.linux-training.be/studentfiles/book.pdf
```

3b. Display the type of file of wolf.jpg and book.pdf

```
file wolf.jpg wolf.pdf
```

3c. Rename wolf.jpg to wolf.pdf (use mv).

```
mv wolf.jpg wolf.pdf
```

3d. Display the type of file of wolf.pdf and book.pdf.

```
file wolf.pdf book.pdf
```

4. Create a directory ~/touched and enter it.

```
mkdir ~/touched ; cd ~/touched
```

5. Create the files today.txt and yesterday.txt in touched.

```
touch today.txt yesterday.txt
```

6. Change the date on yesterday.txt to match yesterday's date.

```
touch -t 200810251405 yesterday (substitute 20081025 with yesterdays date)
```

7. Copy yesterday.txt to copy.yesterday.txt

```
cp yesterday.txt copy.yesterday.txt
```

8. Rename copy.yesterday.txt to kim

```
mv copy.yesterday.txt kim
```

9. Create a directory called ~/testbackup and copy all files from ~/touched in it.

```
mkdir ~/testbackup ; cp -r ~/touched ~/testbackup/
```

10. Use one command to remove the directory ~/testbackup and all files in it.

```
rm -rf ~/testbackup
```

11. Create a directory ~/etcbackup and copy all *.conf files from /etc in it. Did you include all subdirectories of /etc ?

```
cp -r /etc/*.conf ~/etcbackup
```

```
Only *.conf files that are directly in /etc/ are copied.
```

12. Use rename to rename all *.bak files to *.backup . (if you have more than one distro available, try it on all!)

```
On RHEL: touch 1.bak 2.bak ; rename bak backup *.bak
```

```
On Debian: touch 1.bak 2.bak ; rename 's/bak/backup/' *.bak
```

2.7. File contents

2.7.1. head

You can use **head** to display the first ten lines of a file.

```
paul@laika:~$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
paul@laika:~$
```

The head command can also display the first n lines of a file.

```
paul@laika:~$ head -4 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
```

Head can also display the first n bytes.

```
paul@laika:~$ head -c4 /etc/passwd
rootpaul@laika:~$
```

2.7.2. tail

Similar to head, the **tail** command will display the last ten lines of a file.

```
paul@laika:~$ tail /etc/services
vboxd           20012/udp
binkp           24554/tcp             # binkp fidonet protocol
asp             27374/tcp             # Address Search Protocol
```

```
asp             27374/udp
csync2          30865/tcp             # cluster synchronization tool
dircproxy       57000/tcp             # Detachable IRC Proxy
tfido           60177/tcp             # fidonet EMSI over telnet
fido            60179/tcp             # fidonet EMSI over TCP

# Local services
paul@laika:~$
```

The tail command has other useful options, we will use some of them during this course.

2.7.3. cat

The **cat** command is one of the most universal tools. All it does is copying standard input to standard output, but in combination with the shell, this can be very powerful and diverse. Some examples will give a glimpse of the possibilities. The first example is simple, you can use cat to display a file on the screen. If the file is longer than the screen, it will scroll by until the end.

```
paul@laika:~$ cat /etc/resolv.conf
nameserver 194.7.1.4
paul@laika:~$
```

2.7.3.1. concatenate

cat is short for **concatenate**. One of the basic uses of cat is to concatenate files into a bigger (or complete) file.

```
paul@laika:~$ echo one > part1
paul@laika:~$ echo two > part2
paul@laika:~$ echo three > part3
paul@laika:~$ cat part1 part2 part3
one
two
three
paul@laika:~$
```

2.7.3.2. create files

You can use cat to create files with one or more lines of text. Just type the command as is shown in the screenshot below. Then type one or more lines, finish each line with the enter key. After the last line, type and hold the Control (Ctrl) key and press d. The **Ctrl d** key combination will send an EOF (End of File) to the running process, this will end the cat command.

```
paul@laika:~/test$ cat > winter.txt
It is very cold today!
```

```
paul@laika:~/test$ cat winter.txt
It is very cold today!
paul@laika:~/test$
```

You can actually choose this end marker for cat with << as is shown in this screenshot.

```
paul@laika:~/test$ cat > hot.txt <<stop
> It is hot today!
> Yes it is summer.
> stop
paul@laika:~/test$ cat hot.txt
It is hot today!
Yes it is summer.
paul@laika:~/test$
```

2.7.3.3. copy files

In the third example you will see that cat can be used to copy files. We will explain in detail what happens here in the bash shell chapter.

```
paul@laika:~/test$ cat winter.txt
It is very cold today!
paul@laika:~/test$ cat winter.txt > cold.txt
paul@laika:~/test$ cat cold.txt
It is very cold today!
paul@laika:~/test$
```

2.7.4. tac

Just one example will show you the purpose of **tac** (as the opposite of cat).

```
paul@laika:~/test$ cat count
one
two
three
four
paul@laika:~/test$ tac count
four
three
two
one
paul@laika:~/test$
```

2.7.5. more and less

The **more** command is useful for displaying files that take up more than one screen. More will allow you to see the contents of the file page by page. You can use the spacebar to see the next page, or q to quit more. Some people prefere the **less** command instead of more.

2.7.6. strings

With the **strings** command you can display readable ascii strings found in (binary) files. This example locates the ls binary, and then displays readable strings in the binary file (output is truncated).

```
paul@laika:~$ which ls
/bin/ls
paul@laika:~$ strings /bin/ls
/lib/ld-linux.so.2
librt.so.1
__gmon_start__
_Jv_RegisterClasses
clock_gettime
libacl.so.1
...
```

2.8. Practice: File contents

1. Display the first 12 lines of /etc/X11/xorg.conf.

2. Display the last line of /etc/passwd.

3. Use cat to create a file named count.txt that looks like this:

```
One
Two
Three
Four
Five
```

4. Use cp to make a backup of this file to cnt.txt.

5. Use cat to make a backup of this file to catcnt.txt

6. Display catcnt.txt, but with all lines in reverse order (the last line first).

7. Use more to display /var/log/messages.

8. Display the readable character strings from the /usr/bin/passwd command.

9. Use ls to find the biggest file in /etc.

2.9. Solution: File contents

1. Display the first 12 lines of /etc/X11/xorg.conf.

```
head -12 /etc/X11/xorg.conf
```

2. Display the last line of /etc/passwd.

```
tail -1 /etc/passwd
```

3. Use cat to create a file named count.txt that looks like this:

```
One
Two
Three
Four
Five
```

```
cat > count.txt
```

4. Use cp to make a backup of this file to cnt.txt.

```
cp count.txt cnt.txt
```

5. Use cat to make a backup of this file to catcnt.txt

```
cat count.txt > catcnt.txt
```

6. Display catcnt.txt, but with all lines in reverse order (the last line first).

```
tac catcnt.txt
```

7. Use more to display /var/log/messages.

```
more /var/log/messages
```

8. Display the readable character strings from the /usr/bin/passwd command.

```
strings /usr/bin/passwd
```

9. Use ls to find the biggest file in /etc.

```
cd ; ls -lrS /etc
```

Chapter 3. The Linux File system Tree

3.1. About files on Linux

3.1.1. case sensitive

Linux is **case sensitive**, this means that FILE1 is different from file1, and /etc/hosts is different from /etc/Hosts (the latter one does not exist on a typical Linux computer).

This screenshot points to the difference between two files, one with uppercase W, the other with lowercase w.

```
paul@laika:~/Linux$ ls
winter.txt  Winter.txt
paul@laika:~/Linux$ cat winter.txt
It is cold.
paul@laika:~/Linux$ cat Winter.txt
It is very cold!
```

3.1.2. everything is a file

A directory is a special kind of file, but it is still a file. Even a terminal window or a hard disk are represented somewhere in de file system hierarchy by a file. It will become clear troughout this course that everything on Linux is a file.

3.1.3. root directory

All Linux systems have a directory structure that starts at the **root directory**. The root directory is represented by a slash, like this: / . Everything that exists on your linux system can be found below this root directory. Let's take a brief look at the contents of the root directory.

```
[paul@RHELv4u3 ~]$ ls /
bin   dev  home  media  mnt  proc  sbin     srv  tftpboot  usr
boot  etc  lib   misc   opt  root  selinux  sys  tmp       var
[paul@RHELv4u3 ~]$
```

3.1.4. man hier (FileSystem Hierarchy)

There are some differences between Linux distributions. For help about your machine, enter **man hier** to find information about the file system hierarchy. This manual will explain the directory structure on your computer.

3.1.5. Filesystem Hierarchy Standard

Red Hat Enterprise Linux, Fedora, Novell SLES/SLED, OpenSUSE and others (even Sun Solaris) all aim to follow the **Filesystem Hierarchy Standard** (FHS). Maybe the FHS will make more Unix file system trees unite in the future. The **FHS** is available online at http://www.pathname.com/fhs/.

3.2. Filesystem Hierarchy

On http://www.pathname.com/fhs/ we read "The filesystem hierarchy standard has been designed to be used by Unix distribution developers, package developers, and system implementors. However, it is primarily intended to be a reference and is not a tutorial on how to manage a Unix filesystem or directory hierarchy." Below we will discuss a couple of root directories. *For a complete reference, you'll have to check with every developer and system administrator in the world ;-)*

3.2.1. /bin binaries

The **/bin** directory contains binaries for use by all users. According to the FHS /bin/date should exist, and /bin should contain /bin/cat. You will find a bin subdirectory in many other directories. Binaries are sometimes called **executables**. In the screenshot below you see a lot of common unix commands like cat, cp, cpio, date, dd, echo, grep and so on. A lot of these will be covered in this book.

```
paul@laika:~$ ls /bin
archdetect       egrep             mt               setupcon
autopartition    false             mt-gnu           sh
bash             fgconsole         mv               sh.distrib
bunzip2          fgrep             nano             sleep
bzcat            fuser             nc               stralign
bzcmp            fusermount        nc.traditional   stty
bzdiff           get_mountoptions  netcat           su
bzegrep          grep              netstat          sync
bzexe            gunzip            ntfs-3g          sysfs
bzfgrep          gzexe             ntfs-3g.probe    tailf
bzgrep           gzip              parted_devices   tar
bzip2            hostname          parted_server    tempfile
bzip2recover     hw-detect         partman          touch
bzless           ip                partman-commit   true
bzmore           kbd_mode          perform_recipe   ulockmgr
cat              kill              pidof            umount
...
```

3.2.2. /boot static files to boot the system

The **/boot** directory contains all files needed to boot the computer. These files don't change very often. On Linux systems you typically find the **/boot/grub** directory here. This /boot/grub contains **/boot/grub/menu.lst** (the grub configuration file is often linked to **/boot/grub/grub.conf**), which defines the bootmenu that is being displayed before the kernel starts.

3.2.3. /dev device files

Device files in **/dev** appear to be ordinary files, but are not located on the harddisk. The /dev directory is populated with files when the kernel is recognizing hardware.

3.2.3.1. Common physical devices

Common hardware such as hard disk devices are represented by device files in **/dev**. Below a screenshot of SATA device files on a laptop and then IDE attached drives on a desktop. (The detailed meaning of these devices will be discussed later.)

```
#
# SATA or SCSI
#
paul@laika:~$ ls /dev/sd*
/dev/sda  /dev/sda1  /dev/sda2  /dev/sda3  /dev/sdb  /dev/sdb1  /dev/sdb2

#
# IDE or ATAPI
#
paul@barry:~$ ls /dev/hd*
/dev/hda  /dev/hda1  /dev/hda2  /dev/hdb  /dev/hdb1  /dev/hdb2  /dev/hdc
```

Besides representing physical hardware, some device files are special. These special devices can be very useful.

3.2.3.2. /dev/tty and /dev/pts

For example **/dev/tty1** represents a terminal or console attached to the system. (Don't break your head on the exact terminology of 'terminal' or 'console', what we mean here is a commandline interface.) When typing commands in a terminal that is part of a graphical interface like Gnome or KDE, then your terminal will be represented as **/dev/pts/1** (1 can be another number).

3.2.3.3. /dev/null

On Linux you will find special devices like **/dev/null** which can be considered a black hole, it has unlimited storage, but nothing can be retrieved from it. Technically speaking, anything given to /dev/null will be discarded. /dev/null can be useful to discard unwanted output from commands. */dev/null is not a good location to store all your backups ;-).*

3.2.4. /etc Configuration Files

All of the machine-specific configuration files should be located in **/etc**. Many times the name of a configuration files is the same as the application or daemon or protocol with .conf added as an extension. But there is much more to be found in /etc.

```
paul@laika:~$ ls /etc/*.conf
/etc/adduser.conf        /etc/ld.so.conf       /etc/scrollkeeper.conf
/etc/brltty.conf         /etc/lftp.conf        /etc/sysctl.conf
/etc/ccertificates.conf  /etc/libao.conf       /etc/syslog.conf
/etc/cvs-cron.conf       /etc/logrotate.conf   /etc/ucf.conf
/etc/ddclient.conf       /etc/ltrace.conf      /etc/uniconf.conf
/etc/debconf.conf        /etc/mke2fs.conf      /etc/updatedb.conf
/etc/deluser.conf        /etc/netscsid.conf    /etc/usplash.conf
/etc/fdmount.conf        /etc/nsswitch.conf    /etc/uswsusp.conf
/etc/hdparm.conf         /etc/pam.conf         /etc/vnc.conf
/etc/host.conf           /etc/pnm2ppa.conf     /etc/wodim.conf
/etc/inetd.conf          /etc/povray.conf      /etc/wvdial.conf
/etc/kernel-img.conf     /etc/resolv.conf
paul@laika:~$
```

3.2.4.1. /etc/X11/

The graphical display (aka **X Window System** or just **X**) is driven by software from the X.org foundation. The configuration file for your graphical display is **/etc/X11/xorg.conf**.

3.2.4.2. /etc/filesystems

When mounting a file system without specifying explicitly the file system, then **mount** will first probe **/etc/filesystems**. Mount will skip lines with the **nodev** directive, and should this file end with a single * on the last line, then mount will continue probing **/proc/filesystems**.

```
paul@RHELv4u4:~$ cat /etc/filesystems
ext3
ext2
nodev proc
nodev devpts
iso9660
vfat
hfs
paul@RHELv4u4:~$
```

3.2.4.3. /etc/skel/

The **skeleton** directory **/etc/skel** is copied to the home directory of a newly created user. It usually contains hidden files like a **.bashrc** script.

3.2.4.4. /etc/sysconfig/

This directory, which is not mentioned in the FHS, contains a lot of Red Hat Enterprise Linux configuration files. We will discuss some of them in greater detail. The screenshot below is the **/etc/sysconfig** from RHELv4u4 with everything installed.

```
paul@RHELv4u4:~$ ls /etc/sysconfig/
apmd          firstboot     irda              network      saslauthd
apm-scripts   grub          irqbalance        networking   selinux
authconfig    hidd          keyboard          ntpd         spamassassin
autofs        httpd         kudzu             openib.conf  squid
bluetooth     hwconf        lm_sensors        pand         syslog
clock         i18n          mouse             pcmcia       sys-config-sec
console       init          mouse.B           pgsql        sys-config-users
crond         installinfo   named             prelink      sys-logviewer
desktop       ipmi          netdump           rawdevices   tux
diskdump      iptables      netdump_id_dsa    rhn          vncservers
dund          iptables-cfg  netdump_id_dsa.p  samba        xinetd
paul@RHELv4u4:~$
```

The file **/etc/sysconfig/firstboot** tells the Red Hat Setup Agent to not run at boot time. If you want to run the Red Hat Setup Agent at the next reboot, then simply remove this file, and run **chkconfig --level 5 firstboot on**. The Red Hat Setup Agent allows you to install the latest updates, create a user account, join the Red Hat Network and more. It will then create the /etc/sysconfig/firstboot file again.

```
paul@RHELv4u4:~$ cat /etc/sysconfig/firstboot
RUN_FIRSTBOOT=NO
```

The file **/etc/sysconfig/harddisks** contains some parameters to tune the hard disks. The file explains itself.

You can see hardware detected by **kudzu** in **/etc/sysconfig/hwconf**. Kudzu is software from Red Hat for automatic discovery and configuration of hardware.

The keyboard type and table are set in the **/etc/sysconfig/keyboard** file. For more console keyboard information, check the manual pages of **keymaps(5)**, **dumpkeys(1)**, **loadkeys(1)** and the directory **/lib/kbd/keymaps/**.

```
root@RHELv4u4:/etc/sysconfig# cat keyboard
KEYBOARDTYPE="pc"
KEYTABLE="us"
```

We will discuss the networking files in this directory in the networking chapter.

3.2.5. /home sweet home

You will find a lot of locations with an extensive hierarchy of personal or project data under **/home**. It is common practice (but not mandatory) to name the users home directory after their username in the format /home/$USERNAME. Like in this example:

```
paul@pasha:~$ ls /home
geert  guillaume  maria  paul  tom
```

Besides giving every user (or every project or group) a location to store personal files, the home directory of a user also serves as a location to store the user profile. A typical Unix user profile contains a bunch of hidden files (files who's filename starts with a dot). The hidden files of the Unix user profile contain settings specific for that user.

```
paul@pasha:~$ ls -d /home/paul/.*
/home/paul/.              /home/paul/.bash_profile  /home/paul/.ssh
/home/paul/..             /home/paul/.bashrc        /home/paul/.viminfo
/home/paul/.bash_history  /home/paul/.lesshst       /home/paul/.Xauthority
```

3.2.6. /initrd

This empty directory is used as a mount point by Red Hat Enterprise Linux during boot time. Removing it causes a kernel panic during the next boot.

3.2.7. /lib shared libraries

Binaries, like those found in /bin, often use shared libraries located in **/lib**. Below a partial screenshot of the contents of /lib.

```
paul@laika:~$ ls /lib/libc*
/lib/libc-2.5.so     /lib/libcfont.so.0.0.0  /lib/libcom_err.so.2.1
/lib/libcap.so.1     /lib/libcidn-2.5.so     /lib/libconsole.so.0
/lib/libcap.so.1.10  /lib/libcidn.so.1       /lib/libconsole.so.0.0.0
/lib/libcfont.so.0   /lib/libcom_err.so.2    /lib/libcrypt-2.5.so
```

3.2.7.1. /lib/modules

Typically, the kernel loads kernel modules from **/lib/modules**.

3.2.7.2. /lib32 and /lib64

We are now (the year 2007) in a transition between 32-bit and 64-bit systems. So you might encounter directories named **/lib32** and **/lib64**, to clarify the register size used at compilation time of the libraries. My current 64-bit laptop has some older 32-bit binaries and libraries for compatibility with legacy applications. The screenshot uses the **file** utility to point out the difference.

```
paul@laika:~$ file /lib32/libc-2.5.so
/lib32/libc-2.5.so: ELF 32-bit LSB shared object, Intel 80386, \
version 1 (SYSV), for GNU/Linux 2.6.0, stripped
paul@laika:~$ file /lib64/libcap.so.1.10
/lib64/libcap.so.1.10: ELF 64-bit LSB shared object, AMD x86-64, \
version 1 (SYSV), stripped
```

The ELF **Executable and Linkable Format** is used in almost every Unix-like operating system since System V.

3.2.8. /media for Removable Media

The **/media** directory serves as a mount point for removable media, meaning devices such as CD-ROM's, digital cameras and various usb-attached devices. Since **/media** is rather new in the Unix world, you could very well encounter systems running without this directory. Solaris 9 does not have it, Solaris 10 does.

```
paul@laika:~$ ls /media/
cdrom  cdrom0  usbdisk
```

3.2.9. /mnt standard mount point

Older Unixes (and Linuxes) used to mount all kind of stuff under /mnt/something/. According to the FHS, **/mnt** should only be used to temporarily mount something. But you will most likely witness a lot of systems with more than one directory underneath /mnt used as a mountpoint for various local and remote filesystems.

3.2.10. /opt Optional software

Most of my systems today have an empty **/opt** directory. It is considered outdated, but you might find some systems with add-on software installed in /opt. If that is the case, the package should install all its files in the typical bin, lib, etc subdirectories in /opt/$packagename/. If for example the package is called wp, then it installs in /opt/wp, putting binaries in /opt/wp/bin and manpages in /opt/wp/man. Most of the default software which comes along with the distribution, will not be installed in /opt.

3.2.11. /proc conversation with the kernel

/proc is another special directory, appearing to be ordinary files, but not taking up diskspace. It is actually a view on the kernel, or better on what the kernel sees, and a means to talk to the kernel directly. **/proc** is a proc filesystem.

```
paul@RHELv4u4:~$ mount -t proc
none on /proc type proc (rw)
```

When listing the /proc directory, you will see a lot of numbers (on any Unix), and some interesting files (on Linux)

```
mul@laika:~$ ls /proc
1      2339   4724  5418  6587  7201       cmdline      mounts
```

```
10175  2523   4729  5421  6596  7204       cpuinfo      mtrr
10211  2783   4741  5658  6599  7206       crypto       net
10239  2975   4873  5661  6638  7214       devices      pagetypeinfo
141    29775  4874  5665  6652  7216       diskstats    partitions
15045  29792  4878  5927  6719  7218       dma          sched_debug
1519   2997   4879  6     6736  7223       driver       scsi
1548   3      4881  6032  6737  7224       execdomains  self
1551   30228  4882  6033  6755  7227       fb           slabinfo
1554   3069   5     6145  6762  7260       filesystems  stat
1557   31422  5073  6298  6774  7267       fs           swaps
1606   3149   5147  6414  6816  7275       ide          sys
180    31507  5203  6418  6991  7282       interrupts   sysrq-trigger
181    3189   5206  6419  6993  7298       iomem        sysvipc
182    3193   5228  6420  6996  7319       ioports      timer_list
18898  3246   5272  6421  7157  7330       irq          timer_stats
19799  3248   5291  6422  7163  7345       kallsyms     tty
19803  3253   5294  6423  7164  7513       kcore        uptime
19804  3372   5356  6424  7171  7525       key-users    version
1987   4      5370  6425  7175  7529       kmsg         version_signature
1989   42     5379  6426  7188  9964       loadavg      vmcore
2      45     5380  6430  7189  acpi       locks        vmnet
20845  4542   5412  6450  7191  asound     meminfo      vmstat
221    46     5414  6551  7192  buddyinfo  misc         zoneinfo
2338   4704   5416  6568  7199  bus        modules
```

Let's investigate the file properties inside /proc. Looking at the date and time will display the current date and time, meaning the files are constantly updated (A view on the kernel).

```
paul@RHELv4u4:~$ date
Mon Jan 29 18:06:32 EST 2007
paul@RHELv4u4:~$ ls -al /proc/cpuinfo
-r--r--r--  1 root root 0 Jan 29 18:06 /proc/cpuinfo
paul@RHELv4u4:~$
paul@RHELv4u4:~$  ...time passes...
paul@RHELv4u4:~$
paul@RHELv4u4:~$ date
Mon Jan 29 18:10:00 EST 2007
paul@RHELv4u4:~$ ls -al /proc/cpuinfo
-r--r--r--  1 root root 0 Jan 29 18:10 /proc/cpuinfo
```

Most files in /proc are 0 bytes, yet they contain data, sometimes a lot of data. You can see this by executing cat on files like **/proc/cpuinfo**, which contains information on the CPU.

```
paul@RHELv4u4:~$ file /proc/cpuinfo
/proc/cpuinfo: empty
paul@RHELv4u4:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 15
model           : 43
model name      : AMD Athlon(tm) 64 X2 Dual Core Processor 4600+
stepping        : 1
cpu MHz         : 2398.628
cache size      : 512 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
```

```
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 1
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge...
bogomips        : 4803.54
```

Just for fun, here is /proc/cpuinfo on a Sun Sunblade 1000...

```
paul@pasha:~$ cat /proc/cpuinfo
cpu : TI UltraSparc III (Cheetah)
fpu : UltraSparc III integrated FPU
promlib : Version 3 Revision 2
prom : 4.2.2
type : sun4u
ncpus probed : 2
ncpus active : 2
Cpu0Bogo : 498.68
Cpu0ClkTck : 000000002cb41780
Cpu1Bogo : 498.68
Cpu1ClkTck : 000000002cb41780
MMU Type : Cheetah
State:
CPU0: online
CPU1: online
```

... and on a Sony Playstation 3.

```
[root@ps3 tmp]# uname -a
Linux ps3 2.6.20-rc5 #58 SMP Thu Jan 18 13:35:01 CET 2007 ppc64 ppc64
ppc64 GNU/Linux
[root@ps3 tmp]# cat /proc/cpuinfo
processor       : 0
cpu             : Cell Broadband Engine, altivec supported
clock           : 3192.000000MHz
revision        : 5.1 (pvr 0070 0501)

processor       : 1
cpu             : Cell Broadband Engine, altivec supported
clock           : 3192.000000MHz
revision        : 5.1 (pvr 0070 0501)

timebase        : 79800000
platform        : PS3
machine         : PS3
```

Most of the files in /proc are read only, some require root privileges. But some files are writable, a lot of files in **/proc/sys** are writable. Let's discuss some of the files in /proc.

3.2.11.1. /proc/cmdline

The parameters that were passed to the kernel at boot time are in **/proc/cmdline**.

```
paul@RHELv4u4:~$ cat /proc/cmdline
ro root=/dev/VolGroup00/LogVol00 rhgb quiet
```

3.2.11.2. /proc/filesystems

The **/proc/filesystems** file displays a list of supported file systems. When you mount a file system without explicitly defining one, then mount will first try to probe **/etc/filesystems** and then probe **/proc/filesystems** for all the filesystems in there without the **nodev** label. If /etc/filesystems ends with a line containing nothing but a *, then both files are probed.

```
paul@RHELv4u4:~$ cat /proc/filesystems
nodev   sysfs
nodev   rootfs
nodev   bdev
nodev   proc
nodev   sockfs
nodev   binfmt_misc
nodev   usbfs
nodev   usbdevfs
nodev   futexfs
nodev   tmpfs
nodev   pipefs
nodev   eventpollfs
nodev   devpts
        ext2
nodev   ramfs
nodev   hugetlbfs
        iso9660
nodev   relayfs
nodev   mqueue
nodev   selinuxfs
        ext3
nodev   rpc_pipefs
nodev   vmware-hgfs
nodev   autofs
paul@RHELv4u4:~$
```

3.2.11.3. /proc/interrupts

On the x86 architecture, **/proc/interrupts** displays the interrupts.

```
paul@RHELv4u4:~$ cat /proc/interrupts
           CPU0
  0:   13876877    IO-APIC-edge  timer
  1:         15    IO-APIC-edge  i8042
  8:          1    IO-APIC-edge  rtc
  9:          0   IO-APIC-level  acpi
 12:         67    IO-APIC-edge  i8042
 14:        128    IO-APIC-edge  ide0
 15:     124320    IO-APIC-edge  ide1
169:     111993   IO-APIC-level  ioc0
177:       2428   IO-APIC-level  eth0
NMI:          0
LOC:   13878037
```

```
ERR:          0
MIS:          0
paul@RHELv4u4:~$
```

On a machine with two CPU's, the file looks like this.

```
paul@laika:~$ cat /proc/interrupts
           CPU0       CPU1
  0:     860013          0  IO-APIC-edge     timer
  1:       4533          0  IO-APIC-edge     i8042
  7:          0          0  IO-APIC-edge     parport0
  8:    6588227          0  IO-APIC-edge     rtc
 10:       2314          0  IO-APIC-fasteoi  acpi
 12:        133          0  IO-APIC-edge     i8042
 14:          0          0  IO-APIC-edge     libata
 15:      72269          0  IO-APIC-edge     libata
 18:          1          0  IO-APIC-fasteoi  yenta
 19:     115036          0  IO-APIC-fasteoi  eth0
 20:     126871          0  IO-APIC-fasteoi  libata, ohci1394
 21:      30204          0  IO-APIC-fasteoi  ehci_hcd:usb1, uhci_hcd:usb2
 22:       1334          0  IO-APIC-fasteoi  saa7133[0], saa7133[0]
 24:     234739          0  IO-APIC-fasteoi  nvidia
NMI:         72         42
LOC:     860000     859994
ERR:          0
paul@laika:~$
```

3.2.11.4. /proc/kcore

The physical memory is represented in **/proc/kcore**. Do not try to cat this file, instead use a debugger. The size of /proc/kcore is the same as your physical memory, plus four bytes.

```
paul@laika:~$ ls -lh /proc/kcore
-r-------- 1 root root 2.0G 2007-01-30 08:57 /proc/kcore
paul@laika:~$
```

3.2.11.5. /proc/mdstat

You can obtain RAID information from the kernel by displaying **/proc/mdstat**. With a RAID configured, it looks like this.

```
paul@RHELv4u2:~$ cat /proc/mdstat
Personalities : [raid5]
md0 : active raid5 sdd1[2] sdc1[1] sdb1[0]
      2088192 blocks level 5, 64k chunk, algorithm 2 [3/3] [UUU]

unused devices: <none>
paul@RHELv4u2:~$
```

When there is no RAID present, the following is displayed.

```
paul@RHELv4u4:~$ cat /proc/mdstat
Personalities :
unused devices: <none>
paul@RHELv4u4:~$
```

3.2.11.6. /proc/meminfo

You will rarely want to look at **/proc/meminfo**...

```
paul@RHELv4u4:~$ cat /proc/meminfo
MemTotal:       255864 kB
MemFree:          5336 kB
Buffers:         42396 kB
Cached:         159912 kB
SwapCached:          0 kB
Active:         104184 kB
Inactive:       119724 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       255864 kB
LowFree:          5336 kB
SwapTotal:     1048568 kB
SwapFree:      1048568 kB
Dirty:              40 kB
Writeback:           0 kB
Mapped:          33644 kB
Slab:            21956 kB
CommitLimit:   1176500 kB
Committed_AS:    82984 kB
PageTables:        960 kB
VmallocTotal:   761848 kB
VmallocUsed:      2588 kB
VmallocChunk:   759096 kB
HugePages_Total:     0
HugePages_Free:      0
Hugepagesize:     4096 kB
```

...since the **free** command displays the same information in a more user friendly output.

```
paul@RHELv4u4:~$ free -om
             total       used       free     shared    buffers     cached
Mem:           249        244          5          0         41        156
Swap:         1023          0       1023
paul@RHELv4u4:~$
```

3.2.11.7. /proc/modules

/proc/modules lists all modules loaded by the kernel. The output would be too long to display here, so lets **grep** for a few. First vm (from Vmware), which tells us that vmmon and vmnet are both loaded. You can display the same information with **lsmod**.

```
paul@laika:~$ cat /proc/modules | grep vm
vmnet 36896 13 - Live 0xffffffff88b21000 (P)
vmmon 194540 0 - Live 0xffffffff88af0000 (P)
paul@laika:~$ lsmod | grep vm
vmnet                   36896  13
vmmon                  194540  0
paul@laika:~$
```

Some modules depend on others. In the following example, you can see that the nfsd module is used by exportfs, lockd and sunrpc.

```
paul@laika:~$ cat /proc/modules | grep nfsd
nfsd 267432 17 - Live 0xffffffff88a40000
exportfs 7808 1 nfsd, Live 0xffffffff88a3d000
lockd 73520 3 nfs,nfsd, Live 0xffffffff88a2a000
sunrpc 185032 12 nfs,nfsd,lockd, Live 0xffffffff889fb000
paul@laika:~$ lsmod | grep nfsd
nfsd                   267432  17
exportfs                 7808  1 nfsd
lockd                   73520  3 nfs,nfsd
sunrpc                 185032  12 nfs,nfsd,lockd
paul@laika:~$
```

3.2.11.8. /proc/mounts

Like the **mount** command and the **/etc/mtab** file, **/proc/mounts** lists all the mounted file systems. But /proc/mounts displays what the kernel sees, so it is always up to date and correct. You see the device, mount point, file system, read-only or read-write and two zero's.

```
paul@RHELv4u4:~$ cat /proc/mounts
rootfs / rootfs rw 0 0
/proc /proc proc rw,nodiratime 0 0
none /dev tmpfs rw 0 0
/dev/root / ext3 rw 0 0
none /dev tmpfs rw 0 0
none /selinux selinuxfs rw 0 0
/proc /proc proc rw,nodiratime 0 0
/proc/bus/usb /proc/bus/usb usbfs rw 0 0
/sys /sys sysfs rw 0 0
none /dev/pts devpts rw 0 0
/dev/sda1 /boot ext3 rw 0 0
none /dev/shm tmpfs rw 0 0
none /proc/sys/fs/binfmt_misc binfmt_misc rw 0 0
sunrpc /var/lib/nfs/rpc_pipefs rpc_pipefs rw 0 0
paul@RHELv4u4:~$
```

3.2.11.9. /proc/partitions

The **/proc/partitions** file contains a table with major and minor number of partitioned devices, their number of blocks and the device name in **/dev**. Verify with **/proc/devices** to link the major number to the proper device.

```
paul@RHELv4u4:~$ cat /proc/partitions
major minor  #blocks  name

   3     0     524288 hda
   3    64     734003 hdb
   8     0    8388608 sda
   8     1     104391 sda1
   8     2    8281507 sda2
   8    16    1048576 sdb
   8    32    1048576 sdc
   8    48    1048576 sdd
 253     0    7176192 dm-0
 253     1    1048576 dm-1
paul@RHELv4u4:~$
```

3.2.11.10. /proc/swaps

You can find information about **swap partition(s)** in **/proc/swaps**.

```
paul@RHELv4u4:~$ cat /proc/swaps
Filename                        Type        Size    Used    Priority
/dev/mapper/VolGroup00-LogVol01    partition   1048568 0       -1
paul@RHELv4u4:~$
```

3.2.12. /root the superuser's home

On many systems, **/root** is the default location for the root user's personal data and profile. If it does not exist by default, then some administrators create it.

3.2.13. /sbin system binaries

Similar to /bin, but mainly for booting and for tools to configure the system. A lot of the system binaries will require root privileges for certain tasks. You will also find a **/sbin** subdirectory in other directories.

3.2.14. /srv served by your system

You may find **/srv** to be empty on many systems, but not for long. The FHS suggests locating cvs, rsync, ftp and www data to this location. The FHS also approves administrative naming in /srv, like /srv/project55/ftp and /srv/sales/www. Red Hat plans to move some data that is currently located in /var to /srv.

3.2.15. /sys Linux 2.6 hot plugging

The **/sys** directory is created for the Linux 2.6 kernel. Since 2.6, Linux uses **sysfs** to support **usb** and **IEEE 1394** (aka **FireWire**) hot plug devices. See the manual pages

of udev(8) (the successor of **devfs**) and hotplug(8) for more info (Or visit http://linux-hotplug.sourceforge.net/).

```
paul@RHELv4u4:~$ ls /sys/*
/sys/block:
dm-0 fd0 hdb md0  ram1  ram11 ram13 ram15 ram3 ram5 ram7 ram9
dm-1 hda hdc ram0 ram10 ram12 ram14 ram2  ram4 ram6 ram8 sda

/sys/bus:
i2c  ide  pci  platform  pnp  scsi  serio  usb

/sys/class:
firmware i2c-adapter input misc netlink printer scsi_device tty
graphics i2c-dev     mem   net  pci_bus raw     scsi_host   usb

/sys/devices:
pci0000:00  platform  system

/sys/firmware:
acpi

/sys/module:
ac       dm_mirror    ext3      ip_conntrack    ipt_state  md5
autofs4  dm_mod       floppy    iptable_filter  ipv6       mii
battery  dm_snapshot  i2c_core  ip_tables       jbd        mptbase
button   dm_zero      i2c_dev   ipt_REJECT      lp         mptfc

/sys/power:
state
paul@RHELv4u4:~$
```

3.2.16. /tmp for temporary files

When applications (or Users) need to store temporary data, they should use **/tmp**. /tmp might take up diskspace, then again, it might also not (as in being mounted inside RAM memory). In any case, files in /tmp can be cleared by the operating system. Never use /tmp to store data that you want to archive.

3.2.17. /usr Unix System Resources

Although **/usr** is pronounced like user, never forget that it stands for Unix System Resources. The /usr hierarchy should contain **sharable, read only** data. Some people even choose to mount /usr as read only. This can be done from its own partition, or from a read only NFS share.

3.2.18. /var variable data

Data that is unpredictable in size, such as log files (**/var/log**), print spool directories (**/var/spool**) and various caches (**/var/cache**) should be located in **/var**. But /var is much more than that, it contains Process ID files in **/var/run** and temporary files that survive a reboot in **/var/tmp**. There will be more examples of /var usage further in this book.

3.2.18.1. /var/lib/rpm

Red Hat Enterprise Linux keeps files pertaining to **RPM** in **/var/lib/rpm/**.

3.2.18.2. /var/spool/up2date

The **Red Hat Update Agent** uses files in **/var/spool/up2date**. This location is also used when files are downloaded from the **Red Hat Network**.

3.2.19. Practice: file system tree

1. Does the file /bin/cat exist ? What about /bin/dd and /bin/echo. What is the type of these files ?

2. What is the size of the Linux kernel file(s) (vmlinu*) in /boot ?

3. Create a directory ~/test. Then issue the following commands:

```
cd ~/test
```

```
dd if=/dev/zero of=zeroes.txt count=1 bs=100
```

```
od zeroes.txt
```

dd will copy one times (count=1) a block of size 100 bytes (bs=100) from the file /dev/zero to ~/test/zeroes.txt. Can you describe the functionality of /dev/zero ?

4. Now issue the following command:

```
dd if=/dev/random of=random.txt count=1 bs=100 ; od random.txt
```

dd will copy one times (count=1) a block of size 100 bytes (bs=100) from the file /dev/random to ~/test/random.txt. Can you describe the functionality of /dev/random ?

5. Issue the following two commands, and look at the first character of each output line.

```
ls -l /dev/sd* /dev/hd*
```

```
ls -l /dev/tty* /dev/input/mou*
```

The first ls will show block(b) devices, the second ls shows character(c) devices. Can you tell the difference between block and character devices ?

6. Use cat to display /etc/hosts and /etc/resolv.conf. What is your idea about the purpose of these files ?

7. Are there any files in /etc/skel/ ? Check also for hidden files.

8. Display /proc/cpuinfo. On what architecture is your Linux running ?

9. Display /proc/interrupts. What is the size of this file ? Where is this file stored ?

10. Can you enter the /root directory ? Are there (hidden) files ?

11. Are ifconfig, fdisk, parted, shutdown and grub-install present in /sbin ? Why are these binaries in /sbin and not in /bin ?

12. Is /var/log a file or a directory ? What about /var/spool ?

13. Open two command prompts (Ctrl-Shift-T in gnome-terminal) or terminals (Ctrl-Alt-F1, Ctrl-Alt-F2, ...) and issue the **who am i** in both. Then try to echo a word from one terminal to the other.

14. Read the man page of **random** and explain the difference between **/dev/random** and **/dev/urandom**.

3.2.20. Solutions: file system tree

1. Does the file /bin/cat exist ? What about /bin/dd and /bin/echo. What is the type of these files ?

```
ls /bin/cat ; file /bin/cat
```

```
ls /bin/dd ; file /bin/dd
```

```
ls /bin/echo ; file /bin/echo
```

2. What is the size of the Linux kernel file(s) (vmlinu*) in /boot ?

```
ls -lh /boot/vm*
```

3. Create a directory ~/test. Then issue the following commands:

```
cd ~/test
```

```
dd if=/dev/zero of=zeroes.txt count=1 bs=100
```

```
od zeroes.txt
```

dd will copy one times (count=1) a block of size 100 bytes (bs=100) from the file /dev/zero to ~/test/zeroes.txt. Can you describe the functionality of /dev/zero ?

/dev/zero is a Linux special device. It can be considered a source of zeroes. You cannot send something to /dev/zero, but you can read zeroes from it.

4. Now issue the following command:

```
dd if=/dev/random of=random.txt count=1 bs=100 ; od random.txt
```

dd will copy one times (count=1) a block of size 100 bytes (bs=100) from the file /dev/random to ~/test/random.txt. Can you describe the functionality of /dev/random ?

/dev/random acts as a **random number generator** on your Linux machine.

5. Issue the following two commands, and look at the first character of each output line.

```
ls -l /dev/sd* /dev/hd*

ls -l /dev/tty* /dev/input/mou*
```

The first ls will show block(b) devices, the second ls shows character(c) devices. Can you tell the difference between block and character devices ?

Block devices are always written to (or read from) in blocks. For hard disks, blocks of 512 bytes are common. Character devices act as a stream of characters (or bytes). Mouse and keyboard are typical character devices.

6. Use cat to display /etc/hosts and /etc/resolv.conf. What is your idea about the purpose of these files ?

/etc/hosts contains hostnames with their ip-address

/etc/resolv.conf should contain the ip-address of a DNS name server.

7. Are there any files in /etc/skel/ ? Check also for hidden files.

```
Issue "ls -al /etc/skel/". Yes, there should be hidden files there.
```

8. Display /proc/cpuinfo. On what architecture is your Linux running ?

```
The file should contain at least one line with Intel or other cpu.
```

9. Display /proc/interrupts. What is the size of this file ? Where is this file stored ?

The size is zero, yet the file contains data. It is not stored anywhere because /proc is a virtual file system that allows you to talk with the kernel. (If you answered "stored in RAM-memory, that is also correct...).

10. Can you enter the /root directory ? Are there (hidden) files ?

```
Try "cd /root". Yes there are (hidden) files there.
```

11. Are ifconfig, fdisk, parted, shutdown and grub-install present in /sbin ? Why are these binaries in /sbin and not in /bin ?

```
Because those files are only meant for system administrators.
```

12. Is /var/log a file or a directory ? What about /var/spool ?

```
Both are directories.
```

13. Open two command prompts (Ctrl-Shift-T in gnome-terminal) or terminals (Ctrl-Alt-F1, Ctrl-Alt-F2, ...) and issue the **who am i** in both. Then try to echo a word from one terminal to the other.

```
tty-terminal: echo Hello > /dev/tty1

pts-terminal: echo Hello > /dev/pts/1
```

14. Read the man page of **random** and explain the difference between **/dev/random** and **/dev/urandom**.

```
man 4 random
```

Chapter 4. Introduction to the shell

4.1. about shells

4.1.1. several shells

The command line interface used on most Linux systems is **bash**, which stands for **Bourne again shell**. Bash incorporates features from **sh** (the original Bourne shell), **csh** (the C shell) and **ksh** (the Korn shell). Ubuntu recently started including the **dash** (Debian ash) shell.

This chapter will explain general features of a shell using mainly the **/bin/bash** shell. Important differences in **/bin/ksh** will be mentioned seperately.

4.1.2. type

To find out whether a command given to the shell will be executed as an **external command** or as a **builtin command**, use the **type** command.

```
paul@laika:~$ type cd
cd is a shell builtin
paul@laika:~$ type cat
cat is /bin/cat
```

You can also use this shell builtin to show you whether the command is aliased or not.

```
paul@laika:~$ type ls
ls is aliased to `ls --color=auto'
```

Careful, some commands exist both as a shell builtin and as an external command.

```
paul@laika:~$ type -a echo
echo is a shell builtin
echo is /bin/echo
paul@laika:~$
```

4.1.3. which

The which command will look for binaries in the **PATH** environment variable. (Variables are explained later). In the screenshot below, it looks like cd is built-in, and ls cp rm mv mkdir pwd and which are external commands.

```
[root@RHEL4b ~]# which cp ls mv rm cd mkdir pwd which
/bin/cp
```

```
/bin/ls
/bin/mv
/bin/rm
/usr/bin/which: no cd in (/usr/kerberos/sbin:/usr/kerberos/bin:...
/bin/mkdir
/bin/pwd
/usr/bin/which
[root@RHEL4b ~]#
```

4.1.4. alias

4.1.4.1. create an alias

The shell will allow you to create aliases. Aliases can be used to create an easier to remember alias for an existing command.

```
[paul@RHELv4u3 ~]$ cat count.txt
one
two
three
[paul@RHELv4u3 ~]$ alias dog=tac
[paul@RHELv4u3 ~]$ dog count.txt
three
two
one
```

4.1.4.2. abbreviate commands

An **alias** can also be useful to abbreviate an existing command.

```
paul@laika:~$ alias ll='ls -lh --color=auto'
paul@laika:~$ alias c='clear'
paul@laika:~$
```

4.1.4.3. default options

Aliases can be used to supply commands with default options. The example below shows how to make the -i option default when typing rm.

```
[paul@RHELv4u3 ~]$ rm -i winter.txt
rm: remove regular file `winter.txt'? no
[paul@RHELv4u3 ~]$ rm winter.txt
[paul@RHELv4u3 ~]$ ls winter.txt
ls: winter.txt: No such file or directory
[paul@RHELv4u3 ~]$ touch winter.txt
[paul@RHELv4u3 ~]$ alias rm='rm -i'
[paul@RHELv4u3 ~]$ rm winter.txt
rm: remove regular empty file `winter.txt'? no
[paul@RHELv4u3 ~]$
```

Some distributions enable default aliases to protect users from accidentaly erasing files ('rm -i', 'mv -i', 'cp -i')

4.1.4.4. viewing aliases

You can give multiple aliases as arguments to the **alias** command to get a small list. Not providing any argument will give you a complete list of current aliases.

```
paul@laika:~$ alias c ll
alias c='clear'
alias ll='ls -lh --color=auto'
```

4.1.4.5. unalias

You can undo an alias with the **unalias** command.

```
[paul@RHEL4b ~]$ which rm
/bin/rm
[paul@RHEL4b ~]$ alias rm='rm -i'
[paul@RHEL4b ~]$ which rm
alias rm='rm -i'
        /bin/rm
[paul@RHEL4b ~]$ unalias rm
[paul@RHEL4b ~]$ which rm
/bin/rm
[paul@RHEL4b ~]$
```

4.1.5. echo

This book frequently uses the **echo** command to demonstrate shell features. The echo command echoes the input that it receives.

```
paul@laika:~$ echo Burtonville
Burtonville
paul@laika:~$ echo Smurfs are blue
Smurfs are blue
```

4.1.6. shell expansion

The shell is very important, because every command on your Linux system is processed and changed by the shell. After you type the command, but before the command is executed the shell might change your command line! The manual page of a typical shell contains more than one hundred pages.

One of the primary features of a shell is to perform a **command line scan**. When you enter a command on the shell's command prompt, and press the enter key, then the

shell will start scanning that line. While scanning the line, the shell might make a lot of changes to the command line you typed. This process is called **shell expansion**. After the shell has finished scanning and changing that line, the line will be executed. Shell expansion is influenced by the following topics : control operators, white space removal, filename generation, variables, escaping, embedding and shell aliases. All these topics are discussed in the next sections.

4.1.7. internal and external commands

Not all commands are external to the shell, some are built-in.

External commands are programs that have their own binary and reside somewhere on the system in a directory. Many external commands are located in /bin or /sbin. **Builtin commands** are functions inside the shell program. When an external command exists with the same name as a built-in command, then the builtin will take priority. You will need to provide the full path to execute the external command.

4.1.8. displaying shell expansion

You can display the shell expansion with **set -x**, and stop displaying it with **set +x**. You might want to use this further on in this course, or when in doubt about what exactly the shell is doing with your command.

```
[paul@RHELv4u3 ~]$ set -x
++ echo -ne '\033]0;paul@RHELv4u3:~\007'
[paul@RHELv4u3 ~]$ echo $USER
+ echo paul
paul
++ echo -ne '\033]0;paul@RHELv4u3:~\007'
[paul@RHELv4u3 ~]$ echo \$USER
+ echo '$USER'
$USER
++ echo -ne '\033]0;paul@RHELv4u3:~\007'
[paul@RHELv4u3 ~]$ set +x
+ set +x
[paul@RHELv4u3 ~]$ echo $USER
paul
```

4.2. Practice: about shells

1. Is tac a shell builtin command ?

2. Is there an existing alias for rm ?

3. Read the man page of rm, make sure you understand the -i option of rm. Create and rm a file to test the -i option.

4. Execute: **alias rm='rm -i'** . Test your alias with a testfile. Does this work as expected ?

5. List all current aliases.

6. Create an alias called 'city' that echoes your hometown.

7. Use your alias to test that it works.

8. Remove your city alias.

9. What is the location of the cat and the passwd commands ?

10. Explain the difference between the following commands:

```
echo
```

```
/bin/echo
```

The 'echo' will be interpreted by the shell as the builtin echo command. The '/bin/echo' will make the shell execute the echo binary located in the /bin directory.

11. Explain the difference between the following commands:

```
echo Hello
```

```
echo -n Hello
```

The -n option of the echo command will prevent echo from echoing a trailing newline. 'echo Hello' will echo six characters in total, 'echo -n hello' only echoes five characters.

4.3. Solution: about shells

1. Is tac a shell builtin command ?

```
type tac
```

2. Is there an existing alias for rm ?

```
alias rm
```

3. Read the man page of rm, make sure you understand the -i option of rm. Create and rm a file to test the -i option.

```
man rm
```

```
touch testfile
```

```
rm -i testfile
```

4. Execute: **alias rm='rm -i'** . Test your alias with a testfile. Does this work as expected ?

```
touch testfile
```

```
rm testfile (should ask for confirmation!)
```

5. List all current aliases.

```
alias
```

6. Create an alias called 'city' that echoes your hometown.

```
alias city='echo Antwerpen'
```

7. Use your alias to test that it works.

```
city (it should display Antwerpen)
```

8. Remove your city alias.

```
unalias city
```

9. What is the location of the cat and the passwd commands ?

```
which cat (probably /bin/cat)
```

```
which passwd (probably /usr/bin/passwd)
```

10. Explain the difference between the following commands:

```
echo
```

```
/bin/echo
```

The **echo** will be interpreted by the shell as the builtin echo command. The **/bin/echo** will make the shell execute the echo binary located in the /bin directory.

11. Explain the difference between the following commands:

```
echo Hello
```

```
echo -n Hello
```

The -n option of the echo command will prevent echo from echoing a trailing newline. **echo Hello** will echo six characters in total, **echo -n hello** only echoes five characters.

4.4. control operators

4.4.1. ; semicolon

You can put two or more commands on the same line, separated by a semicolon **;**. The scan will then go until each semicolon, and the lines will be executed sequentially, with the shell waiting for each command to end before starting the next one.

```
[paul@RHELv4u3 ~]$ echo Hello
Hello
[paul@RHELv4u3 ~]$ echo World
World
[paul@RHELv4u3 ~]$ echo Hello ; echo World
Hello
World
```

```
[paul@RHELv4u3 ~]$
```

4.4.2. & ampersand

When on the other hand you end a line with an ampersand **&**, then the shell will not wait for the command to finish. You will get your shell prompt back, and the command is executed in background. You will get a message when it has finished executing in background.

```
[paul@RHELv4u3 ~]$ sleep 20 &
[1] 7925
[paul@RHELv4u3 ~]$
...wait 20 seconds...
[paul@RHELv4u3 ~]$
[1]+  Done                    sleep 20
```

The technical explanation of what happens in this case is explained in the chapter about **processes**.

4.4.3. && double ampersand

You can control execution of commands with **&&** denoting a logical AND. With && the second command is only executed when the first one succeeds (returns a zero exit status).

```
paul@barry:~$ echo first && echo second ; echo third
first
second
third
paul@barry:~$ zecho first && echo second ; echo third
-bash: zecho: command not found
third
paul@barry:~$
```

Another example of the same **logical AND** principle.

```
[paul@RHELv4u3 ~]$ cd gen && ls
file1  file3  File55  fileab  FileAB   fileabc
file2  File4  FileA   Fileab  fileab2
[paul@RHELv4u3 gen]$ cd gen && ls
-bash: cd: gen: No such file or directory
[paul@RHELv4u3 gen]$
```

4.4.4. || double vertical bar

The reverse is true for || . Meaning the second command is only executed when the first command fails (or in other words: returns a non-zero exit status).

```
paul@barry:~$ echo first || echo second ; echo third
first
third
paul@barry:~$ zecho first || echo second ; echo third
-bash: zecho: command not found
second
third
paul@barry:~$
```

Another example of the same **logical OR** principle.

```
[paul@RHELv4u3 ~]$ cd gen || ls
[paul@RHELv4u3 gen]$ cd gen || ls
-bash: cd: gen: No such file or directory
file1  file3  File55  fileab  FileAB   fileabc
file2  File4  FileA   Fileab  fileab2
[paul@RHELv4u3 gen]$
```

4.4.5. Combining && and ||

You can use the logical AND and OR to echo whether a command worked or not.

```
paul@laika:~/test$ rm file1 && echo It worked! || echo It failed!
It worked!
paul@laika:~/test$ rm file1 && echo It worked! || echo It failed!
rm: cannot remove `file1': No such file or directory
It failed!
paul@laika:~/test$
```

4.4.6. # pound sign

Anything written after a pound sign (#) is ignored by the shell. This is useful to write **shell comment**, but has no influence on the command execution or shell expansion.

```
paul@barry:~$ mkdir test    # we create a directory
paul@barry:~$ cd test       #### we enter the directory
paul@barry:~/test$ ls       # is it empty ?
paul@barry:~/test$
```

4.4.7. \ escaping special characters

When you want to use one of the shell control characters, but without the shell interpreting them, then you can **escape** them with a backslash \.

```
[paul@RHELv4u3 ~]$ echo hello \; world
hello ; world
[paul@RHELv4u3 ~]$ echo hello\ \ \ world
hello   world
```

```
[paul@RHELv4u3 ~]$ echo escaping \\\ \#\ \&\ \"\ \'
escaping \ # & " '
[paul@RHELv4u3 ~]$ echo escaping \\\?\*\"\'
escaping \?*"'
```

4.4.8. end of line backslash

Lines ending in a backslash are continued on the next line. The shell is not interpreting the newline character and will wait with shell expansion and execution of the command line until a newline without backslash is encountered.

```
[paul@RHEL4b ~]$ echo This command line \
> is split in three \
> parts
This command line is split in three parts
[paul@RHEL4b ~]$
```

4.5. Practice: control operators

0. All these questions can be answered by one command line!!

1. When you type 'passwd', which file is executed ?

2. What kind of file is that ?

3. Execute the pwd command twice. (remember 0.)

4. Execute ls after cd /etc, but only if cd /etc did not error.

5. Execute cd /etc after cd etc, but only if cd etc fails.

6. Execute sleep 10, what is this command doing ?

7. Execute sleep 200 in background (do not wait for it to finish).

8. Use echo to display "Hello World with strange' characters \ * [} ~ \\ ." (including all quotes)

9. Use one echo command to display three words on three lines.

4.6. Solution: control operators

0. Each question can be answered by one command line!

1. When you type 'passwd', which file is executed ?

```
which passwd
```

2. What kind of file is that ?

```
file $(which passwd)
```

3. Execute the pwd command twice. (remember 0.)

```
pwd ; pwd
```

4. Execute ls after cd /etc, but only if cd /etc did not error.

```
cd /etc && ls
```

5. Execute cd /etc after cd etc, but only if cd etc fails.

```
cd etc || cd /etc
```

6. Execute sleep 10, what is this command doing ?

```
pausing for ten seconds
```

7. Execute sleep 200 in background (do not wait for it to finish).

```
sleep 200 &
```

8. Use echo to display "Hello World with strange' characters \ * [} ~ \\ ." (including all quotes)

```
echo \"Hello World with strange\' characters \\ \* \[ \} \~ \\\\ \. \"
```

```
echo \""Hello World with strange' characters \ * [ } ~ \\ . "\"
```

9. Use one echo command to display three words on three lines.

```
echo -e "one \ntwo \nthree"
```

4.7. shell variables

4.7.1. $ dollar sign

Another important character interpreted by the shell is the dollar sign **$**. The shell will look for an **environment variable** named like the string behind the dollar sign and replace it with the value of the variable (or with nothing if the variable does not exist).

An example of the $USER variable. The example shows that shell variables are case sensitive!

```
[paul@RHELv4u3 ~]$ echo Hello $USER
Hello paul
[paul@RHELv4u3 ~]$ echo Hello $user
Hello
```

4.7.2. common variables

Some more examples using $HOSTNAME, $USER, $UID, $SHELL and $HOME.

```
[paul@RHELv4u3 ~]$ echo This is the $SHELL shell
This is the /bin/bash shell
[paul@RHELv4u3 ~]$ echo This is $SHELL on computer $HOSTNAME
This is /bin/bash on computer RHELv4u3.localdomain
[paul@RHELv4u3 ~]$ echo The userid of $USER is $UID
The userid of paul is 500
[paul@RHELv4u3 ~]$ echo My homedir is $HOME
My homedir is /home/paul
```

4.7.3. $? dollar question mark

The exit code of the previous command is stored in the shell variable $?. Actually **$?** is a shell parameter and not a variable, you cannot assign a value to $?.

```
paul@laika:~/test$ touch file1 ; echo $?
0
paul@laika:~/test$ rm file1 ; echo $?
0
paul@laika:~/test$ rm file1 ; echo $?
rm: cannot remove `file1': No such file or directory
1
```

4.7.4. unbound variables

The example below tries to display the value of the $MyVar variable, but it fails because the variable does not exist. By default the shell will display nothing when a variable is unbound (does not exist).

```
[paul@RHELv4u3 gen]$ echo $MyVar

[paul@RHELv4u3 gen]$
```

There is however the **nounset** shell attribute that you can use to generate an error when a variable does not exist.

```
paul@laika:~$ set -u
paul@laika:~$ echo $Myvar
bash: Myvar: unbound variable
paul@laika:~$ set +u
paul@laika:~$ echo $Myvar

paul@laika:~$
```

In the bash shell **set -u** is identical to **set -o nounset** and likewise **set +u** is identical to **set +o nounset**.

4.7.5. creating and setting variables

The example creates the variable $MyVar and sets its value.

```
[paul@RHELv4u3 gen]$ MyVar=555
[paul@RHELv4u3 gen]$ echo $MyVar
555
[paul@RHELv4u3 gen]$
```

4.7.6. set

You can use the **set** command to display a list of environment variables. On Ubuntu and Debian systems, the set command will end the list of shell variables with a list of shell functions, use **set | more** to see the variables then.

4.7.7. unset

Use the **unset** command to remove a variable from your shell environment.

```
[paul@RHEL4b ~]$ MyVar=8472
[paul@RHEL4b ~]$ echo $MyVar;unset MyVar;echo $MyVar
8472

[paul@RHEL4b ~]$
```

4.7.8. env

The **env** command can also be useful for other neat things, like starting a clean shell (a shell without any inherited environment). The **env -i** command clears the environment for the subshell. Notice that bash will set the $SHELL variable on startup.

```
[paul@RHEL4b ~]$ bash -c 'echo $SHELL $HOME $USER'
/bin/bash /home/paul paul
[paul@RHEL4b ~]$ env -i bash -c 'echo $SHELL $HOME $USER'
/bin/bash
[paul@RHEL4b ~]$
```

You can also use the env tool to set the LANG variable (or any other) for an instance of bash with one command. The example below uses this to show the influence of the LANG variable on file globbing.

```
[paul@RHEL4b test]$ env LANG=C bash -c 'ls File[a-z]'
Filea  Fileb
[paul@RHEL4b test]$ env LANG=en_US.UTF-8 bash -c 'ls File[a-z]'
Filea  FileA  Fileb  FileB
[paul@RHEL4b test]$
```

4.7.9. exporting variables

You can export shell variables to other shells with the **export** command. This will export the variable to child shells.

```
[paul@RHEL4b ~]$ var3=three
[paul@RHEL4b ~]$ var4=four
[paul@RHEL4b ~]$ export var4
[paul@RHEL4b ~]$ echo $var3 $var4
three four
[paul@RHEL4b ~]$ bash
[paul@RHEL4b ~]$ echo $var3 $var4
four
```

But it will not export to the parent shell (previous screenshot continued).

```
[paul@RHEL4b ~]$ export var5=five
[paul@RHEL4b ~]$ echo $var3 $var4 $var5
four five
[paul@RHEL4b ~]$ exit
exit
[paul@RHEL4b ~]$ echo $var3 $var4 $var5
three four
[paul@RHEL4b ~]$
```

4.7.10. delineate variables

Until now, we have seen that bash interpretes a variable starting from a dollar sign, until the first occurence of a non-alphanumerical character that is not an underscore. In some situations, this can be a problem. This issue can be resolved with curly braces like in this example.

```
[paul@RHEL4b ~]$ prefix=Super
[paul@RHEL4b ~]$ echo Hello $prefixman and $prefixgirl
Hello  and
[paul@RHEL4b ~]$ echo Hello ${prefix}man and ${prefix}girl
Hello Superman and Supergirl
[paul@RHEL4b ~]$
```

4.7.11. quotes and variables

Notice that double quotes still allow the parsing of variables, whereas single quotes prevent this.

```
[paul@RHELv4u3 ~]$ MyVar=555
[paul@RHELv4u3 ~]$ echo $MyVar
555
```

```
[paul@RHELv4u3 ~]$ echo "$MyVar"
555
[paul@RHELv4u3 ~]$ echo '$MyVar'
$MyVar
```

The bash shell will replace variables with their value in double quoted lines, but not in single quoted lines.

```
paul@laika:~$ city=Burtonville
paul@laika:~$ echo "We are in $city today."
We are in Burtonville today.
paul@laika:~$ echo 'We are in $city today.'
We are in $city today.
```

4.8. Practice: shell variables

1. Use echo to display Hello followed by your username. (use a bash variable!)

2. Copy the value of $LANG to $MyLANG.

3. List all current shell variables.

4. Create a variable MyVar with a value of 1201.

5. Do the env and set commands display your variable ?

6. Destroy your variable.

7. Find the list of shell options in the man page of bash. What is the difference between "set -u" and "set -o nounset" ?

8. Create two variables, and export one of them.

9. Display the exported variable in an interactive child shell.

10. Create a variable, give it the value 'Dumb', create another variable with value 'do'. Use echo and the two variables to echo Dumbledore.

11. Activate **nounset** in your shell. Test that it shows an error message when using non-existing variables.

12. Deactivate nounset.

4.9. Solution: shell variables

1. Use echo to display Hello followed by your username. (use a bash variable!)

```
echo Hello $USER
```

2. Copy the value of $LANG to $MyLANG.

```
MyLANG=$LANG
```

3. List all current shell variables.

```
set
```

4. Create a variable MyVar with a value of 1201.

```
MyVar=1201
```

5. Do the env and set commands display your variable ?

```
env | grep 1201 ; echo -- ; set | grep 1201
```

You will notice that **set** displays all variables, whereas **env** does not.

6. Destroy your variable.

```
unset MyVar
```

7. Find the list of shell options in the man page of bash. What is the difference between "set -u" and "set -o nounset" ?

read the manual of bash (man bash), search for nounset -- both mean the same thing.

8. Create two variables, and export one of them.

```
var1=1; export var2=2
```

9. Display the exported variable in an interactive child shell.

```
bash
```

```
echo $var2
```

10. Create a variable, give it the value 'Dumb', create another variable with value 'do'. Use echo and the two variables to echo Dumbledore.

```
varx=Dumb; vary=do
```

```
echo ${varx}le${vary}re
```

11. Activate **nounset** in your shell. Test that it shows an error message when using non-existing variables.

```
set -u
```

```
set -o nounset
```

Both these lines have the same effect (read the manual of bash, search for nounset).

12. Deactivate nounset.

```
set +u
```

```
set +o nounset
```

4.10. white space and quoting

4.10.1. white space removal

Before execution, the shell looks at the command line. Parts that are seperated by one or more consecutive **white spaces** (or tabs) are considered separate arguments. The first argument is the command to be executed, the other arguments are given to the command as arguments.

This explains why the following four different command lines are the same after **shell expansion**.

```
[paul@RHELv4u3 ~]$ echo Hello World
Hello World
[paul@RHELv4u3 ~]$ echo Hello   World
Hello World
[paul@RHELv4u3 ~]$ echo   Hello   World
Hello World
[paul@RHELv4u3 ~]$    echo      Hello      World
Hello World
[paul@RHELv4u3 ~]$
```

The echo command will separate the different arguments it recieves from the shell by a white space.

4.10.2. single quotes

You can prevent the removal of white spaces by quoting the spaces.

```
[paul@RHEL4b ~]$ echo 'A line with      single    quotes'
A line with      single    quotes
[paul@RHEL4b ~]$
```

4.10.3. double quotes

You can also prevent the removal of white spaces by double quoting the spaces.

```
[paul@RHEL4b ~]$ echo "A line with      double    quotes"
A line with      double    quotes
[paul@RHEL4b ~]$
```

The only difference between single and double quotes is the parsing of shell variables. You can already see what happens in this screenshot.

```
paul@laika:~$ echo 'My user is $USER'
My user is $USER
```

```
paul@laika:~$ echo "My user is $USER"
My user is paul
```

4.10.4. echo and quotes

Quoted lines can include special escaped characters recognized by the **echo** command (when using **echo -e**). The screenshot below shows how to use escaped n for a newline and escaped t for a tab (usually eight white spaces).

```
[paul@RHEL4b ~]$ echo -e "A line with \na newline"
A line with
a newline
[paul@RHEL4b ~]$ echo -e 'A line with \na newline'
A line with
a newline
[paul@RHEL4b ~]$ echo -e "A line with \ta tab"
A line with     a tab
[paul@RHEL4b ~]$ echo -e 'A line with \ta tab'
A line with     a tab
[paul@RHEL4b ~]$
```

The echo command can generate more than white spaces, tabs and newlines. Look in the man page for a list of options (and don't forget that echo can be both builtin and external).

4.10.5. shell embedding

Shells can be embedded on the command line, or in other words the command line scan can spawn new processes, containing a fork of the current shell. You can use variables to prove that new shells are created. In the screenshot below, the variable $var1 only exists in the (temporary) sub shell.

```
[paul@RHELv4u3 gen]$ echo $var1

[paul@RHELv4u3 gen]$ echo $(var1=5;echo $var1)
5
[paul@RHELv4u3 gen]$ echo $var1

[paul@RHELv4u3 gen]$
```

You can embed a shell in an **embedded shell**, this is called **nested embedding** of shells.

```
$ P=Parent;
$ echo $P$C$G - $(C=Child;echo $P$C$G - ;echo $(G=Grand;echo $P$C$G))
Parent - ParentChild - ParentChildGrand
```

Single embedding can be useful to avoid changing your current directory. The screenshot below uses back ticks instead of dollar-bracket to embed.

```
[paul@RHELv4u3 ~]$ echo `cd /etc; ls -d * | grep pass`
passwd passwd- passwd.OLD
[paul@RHELv4u3 ~]$
```

4.10.6. backticks and single quotes

Placing the embedding between **backticks** uses one character less than the dollar and parenthesis combo. Be careful however, backticks are often confused with single quotes. The technical difference between ' and ` is significant! You can not use backticks to nest embedded shells.

```
[paul@RHELv4u3 gen]$ echo `var1=5;echo $var1`
5
[paul@RHELv4u3 gen]$ echo 'var1=5;echo $var1'
var1=5;echo $var1
[paul@RHELv4u3 gen]$
```

4.11. Practice: white space and quoting

1. Display **A B C** with two spaces between B and C.

2. Complete the following command (do not use spaces) to display exactly the following output:

```
echo -e "4+4=8" ; echo -e "10+14=24"
```

```
4+4     =8
```

```
10+14   =24
```

3. Use echo to display the following exactly:

4. Use one echo command to display three words on three lines.

5. Execute **cd /var** and **ls** in an embedded shell.

6. Create the variable embvar in an embedded shell and echo it. Does the variable exist in your current shell now ?

7. Explain what "set -x" does. Can this be useful ?

8. Given the following screenshot, add exactly four characters to that command line so that the total output is FirstMiddleLast.

```
[paul@RHEL4b ~]$ echo  First; echo  Middle; echo  Last
First
Middle
Last
[paul@RHEL4b ~]$
```

4.12. Solution: white space and quoting

1. Display **A B C** with two spaces between B and C.

```
echo "A B  C"
```

2. Complete the following command (do not use spaces) to display exactly the following output:

```
echo -e "4+4=8" ; echo -e "10+14=24"
```

```
4+4     =8
```

```
10+14   =24
```

The solution is to use tabs with \t.

```
echo -e "4+4\t=8" ; echo -e "10+14\t=24"
```

3. Use echo to display the following exactly:

```
""\\`;     "_+
```

```
echo \"\"\\\\\`\;\ \ \ \ \"\_\+
```

4. Use one echo command to display three words on three lines.

```
echo -e "one \ntwo \nthree"
```

5. Execute **cd /var** and **ls** in an embedded shell.

```
$(cd /var ; ls)
```

6. Create the variable embvar in an embedded shell and echo it. Does the variable exist in your current shell now ?

```
$(embvar=emb;echo $embvar) ; echo $embvar (the last echo fails).
```

```
$embvar does not exist in your current shell
```

7. Explain what "set -x" does. Can this be useful ?

```
I can be useful to display shell expansion for troubleshooting your command.
```

8. Given the following screenshot, add exactly four characters to that command line so that the total output is FirstMiddleLast.

```
[paul@RHEL4b ~]$ echo  First; echo  Middle; echo  Last
First
Middle
Last
[paul@RHEL4b ~]$
```

```
echo -n First; echo -n Middle; echo Last
```

4.13. file globbing

4.13.1. * asterisk

The shell is also responsible for **file globbing** (or dynamic filename generation). The asterisk ***** is interpreted by the shell as a sign to generate filenames, matching the asterisk to any combination of characters (even none). When no path is given, the shell will use filenames in the current directory. See the man page of **glob(7)** for more information. (This is part of LPI topic 1.103.3.)

```
[paul@RHELv4u3 gen]$ ls
file1  file2  file3  File4  File55  FileA  fileab  Fileab  FileAB  fileabc
[paul@RHELv4u3 gen]$ ls File*
File4  File55  FileA  Fileab  FileAB
[paul@RHELv4u3 gen]$ ls file*
file1  file2  file3  fileab  fileabc
[paul@RHELv4u3 gen]$ ls *ile55
File55
[paul@RHELv4u3 gen]$ ls F*ile55
File55
[paul@RHELv4u3 gen]$ ls F*55
File55
[paul@RHELv4u3 gen]$
```

4.13.2. ? question mark

Similar to the asterisk, the question mark **?** is interpreted by the shell as a sign to generate filenames, matching the question mark with exactly one character.

```
[paul@RHELv4u3 gen]$ ls
file1  file2  file3  File4  File55  FileA  fileab  Fileab  FileAB  fileabc
[paul@RHELv4u3 gen]$ ls File?
File4  FileA
[paul@RHELv4u3 gen]$ ls Fil?4
File4
[paul@RHELv4u3 gen]$ ls Fil??
File4  FileA
[paul@RHELv4u3 gen]$ ls File??
File55  Fileab  FileAB
[paul@RHELv4u3 gen]$
```

4.13.3. [] square brackets

The square bracket **[** is interpreted by the shell as a sign to generate filenames, matching any of the characters between **[** and the first subsequent **]**. The order in this list between the brackets is not important. Each pair of brackets is replaced by exactly one character.

```
[paul@RHELv4u3 gen]$ ls
file1  file2  file3  File4  File55  FileA  fileab  Fileab  FileAB  fileabc
[paul@RHELv4u3 gen]$ ls File[5A]
FileA
[paul@RHELv4u3 gen]$ ls File[A5]
FileA
[paul@RHELv4u3 gen]$ ls File[A5][5b]
File55
[paul@RHELv4u3 gen]$ ls File[a5][5b]
File55  Fileab
[paul@RHELv4u3 gen]$ ls File[a5][5b][abcdefghijklm]
ls: File[a5][5b][abcdefghijklm]: No such file or directory
[paul@RHELv4u3 gen]$ ls file[a5][5b][abcdefghijklm]
fileabc
[paul@RHELv4u3 gen]$
```

You can also exclude characters from a list between square brackets with the exclamation mark **!**. And you are allowed to make combinations of these **wild cards**.

```
[paul@RHELv4u3 gen]$ ls
file1  file2  file3  File4  File55  FileA  fileab  Fileab  FileAB  fileabc
[paul@RHELv4u3 gen]$ ls file[a5][!Z]
fileab
[paul@RHELv4u3 gen]$ ls file[!5]*
file1  file2  file3  fileab  fileabc
[paul@RHELv4u3 gen]$ ls file[!5]?
fileab
[paul@RHELv4u3 gen]$
```

4.13.4. a-z and 0-9 ranges

The bash shell will also understand ranges of characters between brackets.

```
[paul@RHELv4u3 gen]$ ls
file1  file3  File55  fileab  FileAB   fileabc
file2  File4  FileA   Fileab  fileab2
[paul@RHELv4u3 gen]$ ls file[a-z]*
fileab  fileab2  fileabc
[paul@RHELv4u3 gen]$ ls file[0-9]
file1  file2  file3
[paul@RHELv4u3 gen]$ ls file[a-z][a-z][0-9]*
fileab2
[paul@RHELv4u3 gen]$
```

4.13.5. $LANG and square brackets

But, don't forget the influence of the **LANG** variable. Some languages include lowercase letters in an uppercase range (and vice versa).

```
paul@RHELv4u4:~/test$ ls [A-Z]ile?
file1  file2  file3  File4
paul@RHELv4u4:~/test$ ls [a-z]ile?
file1  file2  file3  File4
```

```
paul@RHELv4u4:~/test$ echo $LANG
en_US.UTF-8
paul@RHELv4u4:~/test$ LANG=C
paul@RHELv4u4:~/test$ echo $LANG
C
paul@RHELv4u4:~/test$ ls [a-z]ile?
file1  file2  file3
paul@RHELv4u4:~/test$ ls [A-Z]ile?
File4
paul@RHELv4u4:~/test$
```

4.14. Bash shell options

Both **set** and **unset** are built-in shell commands. They can be used to set options of the bash shell itself. The next example will clarify this. By default, the shell will treat unset variables as a variable having no value. By setting the -u option, the shell will treat any reference to unset variables as an error. See the man page of bash for more information.

```
[paul@RHEL4b ~]$ echo $var123

[paul@RHEL4b ~]$ set -u
[paul@RHEL4b ~]$ echo $var123
-bash: var123: unbound variable
[paul@RHEL4b ~]$ set +u
[paul@RHEL4b ~]$ echo $var123

[paul@RHEL4b ~]$
```

To list all the set options for your Bash shell, use **echo $-**. The noclobber option will be explained later in this book (in the I/O redirection chapter).

```
[paul@RHEL4b ~]$ echo $-
himBH
[paul@RHEL4b ~]$ set -C ; set -u
[paul@RHEL4b ~]$ echo $-
himuBCH
[paul@RHEL4b ~]$ set +C ; set +u
[paul@RHEL4b ~]$ echo $-
himBH
[paul@RHEL4b ~]$
```

4.15. shell history

4.15.1. history variables

The **bash** shell will remember the commands you type, so you can easily repeat previous commands. Some variables are defining this process: **$HISTFILE** points to the location of the history file, **$HISTSIZE** will tell you how many commands will

be remembered in your current shell session, **$HISTFILESIZE** is the truncate limit for the number of commands in the history file. Your shell session history is written to the file when exiting the shell. This screenshot lists some history variables in bash.

```
[paul@RHELv4u3 ~]$ echo $HISTFILE
/home/paul/.bash_history
[paul@RHELv4u3 ~]$ echo $HISTFILESIZE
1000
[paul@RHELv4u3 ~]$ echo $HISTSIZE
1000
[paul@RHELv4u3 ~]$
```

The $HISTFILE (defaults to .sh_history in the home directory) and $HISTSIZE variables are also used by the Korn shell (ksh).

```
$ set | grep -i hist
HISTSIZE=5000
```

4.15.2. repeating commands in bash

To repeat the last command in bash, type **!!**. This is pronounced as **bang bang**. To repeat older commands, use **history** to display your history and type **!** followed by a number. The shell will echo the command and execute it.

```
[paul@RHELv4u3 ~]$ history
2  cat /etc/redhat-release
3  uname -r
4  rpm -qa | grep ^parted
...
[paul@RHELv4u3 ~]$ !3
uname -r
2.6.9-34.EL
[paul@RHELv4u3 ~]$
```

You can also use the bash with one or more characters, bash will then repeat the last command that started with those characters. But this can be very very dangerous, you have to be sure about the last command in your current shell history that starts with those characters!

```
[paul@RHEL4b ~]$ ls file4
file4
[paul@RHEL4b ~]$ !ls
ls file4
file4
```

You can also use a colon followed by a regular expression to manipulate the previous command.

```
[paul@RHEL4b ~]$ !ls:s/4/5
```

```
ls file5
file5
```

The history command can also receive the count of history lines required.

```
[paul@RHEL4b ~]$ history 4
  422  ls file4
  423  ls file4
  424  ls file5
  425  history 4
[paul@RHEL4b ~]$
```

4.15.3. repeating commands in ksh

Repeating a command in the Korn shell is very similar. The Korn shell also knows the **history** command, but uses the letter **r** to recall lines from history.

This screenshot shows the history command. Note the different meaning of the parameter.

```
$ history 17
17      clear
18      echo hoi
19      history 12
20      echo world
21      history 17
```

Repeating with r can be combined with the line numbers given by the history command, or with the first few letters of the command.

```
$ r e
echo world
world
$ cd /etc
$ r
cd /etc
$
```

4.16. Practice: discover the shell

1. Create a testdir and enter it.

2. Create files file1 file10 file11 file2 File2 File3 file33 fileAB filea fileA fileAAA file(file 2 (the last one has 6 characters including a space)

3. List (with ls) all files starting with file

4. List (with ls) all files starting with File

5. List (with ls) all files starting with file and ending in a number.

6. List (with ls) all files starting with file and ending with a letter

7. List (with ls) all files starting with File and having a digit as fifth character.

8. List (with ls) all files starting with File and having a digit as fifth character and nothing else.

9. List (with ls) all files starting with a letter and ending in a number.

10. List (with ls) all files that have exactly five characters.

11. List (with ls) all files that start with f or F and end with 3 or A.

12. List (with ls) all files that start with f have i or R as second character and end in a number.

13. List all files that do not start with the letter F.

14. Copy the value of $LANG to $MyLANG.

15. Show the influence of $LANG in listing A-Z or a-z ranges.

16. Write a command line that executes 'rm file55'. Your command line should print 'success' if file55 is removed, and print 'failed' if there was a problem.

17. You receive information that one of your servers was cracked, the cracker probably replaced the ls command. You know that the echo command is safe to use. Can echo replace ls ? How can you list the files in the current directory with echo ?

18. The cd command is also compromised, can echo be used to list files in other directories ? Explain how this works (list the contents of /etc and /bin without ls).

19. Is there another command besides cd to change directories ?

20. Make sure bash remembers the last 5000 commands you typed.

21. Open more than one console (press Ctrl-shift-t in gnome terminal) with the same user account. When is command history written to the history file ?

22. Issue the date command. Now display the date in YYYY/MM/DD format.

23. Issue the cal command. Display a calendar of 1582 and 1752. Notice anything special ?

4.17. Solution: discover the shell

1. Create a testdir and enter it.

```
mkdir testdir; cd testdir
```

2. Create files file1 file10 file11 file2 File2 File3 file33 fileAB filea fileA fileAAA file(file 2 (the last one has 6 characters including a space)

```
touch file1 file10 file11 file2 File2 File3 file33 fileAB filea fileA fileAAA
touch "file("
touch "file 2"
```

3. List (with ls) all files starting with file

```
ls file*
```

4. List (with ls) all files starting with File

```
ls File*
```

5. List (with ls) all files starting with file and ending in a number.

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

6. List (with ls) all files starting with file and ending with a letter

```
ls file*[a-z]
```

7. List (with ls) all files starting with File and having a digit as fifth character.

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

8. List (with ls) all files starting with File and having a digit as fifth character and nothing else.

```
ls File[0-9]
```

9. List (with ls) all files starting with a letter and ending in a number.

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

10. List (with ls) all files that have exactly five characters.

```
ls ?????
```

11. List (with ls) all files that start with f or F and end with 3 or A.

```
ls [fF]*[3A]
```

12. List (with ls) all files that start with f have i or R as second character and end in a number.

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

13. List all files that do not start with the letter F.

```
ls [!F]*
```

14. Copy the value of $LANG to $MyLANG.

```
MyLANG=$LANG
```

15. Show the influence of $LANG in listing A-Z or a-z ranges.

```
see example in book
```

16. Write a command line that executes 'rm file55'. Your command line should print 'success' if file55 is removed, and print 'failed' if there was a problem.

```
rm file55 && echo success || echo failed
```

17. You receive information that one of your servers was cracked, the cracker probably replaced the ls command. You know that the echo command is safe to use. Can echo replace ls ? How can you list the files in the current directory with echo ?

```
echo *
```

18. The cd command is also compromised, can echo be used to list files in other directories ? Explain how this works (list the contents of /etc and /bin without ls).

```
echo /etc/*.conf # the shell expands the directory for you
```

19. Is there another command besides cd to change directories ?

```
pushd popd
```

20. Make sure bash remembers the last 5000 commands you typed.

```
HISTSIZE=5000
```

21. Open more than one console (press Ctrl-shift-t in gnome terminal) with the same user account. When is command history written to the history file ?

```
when you type exit
```

22. Issue the date command. Now display the date in YYYY/MM/DD format.

```
date +%Y/%m/%d
```

23. Issue the cal command. Display a calendar of 1582 and 1752. Notice anything special ?

```
cal 1582
```

The calendars are different depending on the country. Check http://www.linux-training.be/studentfiles/dates.txt

Chapter 5. Introduction to vi

5.1. About vi

The editor **vi** is installed on almost every Unix system in the world. Linux will very often install **vim** (**Vi IMproved**) which is very similar, but improved. Every Linux system administrator should know vi (or rather vim), because it is often an easy tool to solve problems.

Many Unix and Linux distributions will also have **emacs**, **nano**, **pico**, **joe** or other editors installed. The choice of favorite editor is often a cause for **flame wars** or polls. Feel free to use any of the alternatives to vi(m).

The vi editor is not intuitive to novices, but once you get to know it, vi becomes a very powerful application. Some basic commands are a A i I o O r x G 'n G' b w dw dd d0 d$ yw yy y0 y$ 3dd p P u U :w :q :w! :q! :wq ZZ :r :!cmd ':r !cmd' ddp yyp /pattern. Most Linux distributions will include the **vimtutor** which is a 45 minute lesson in vi.

5.2. Introduction to using vi

5.2.1. command mode and insert mode

The vi editor starts in **command mode**. In command mode, you can type commands. The commands a A i I o O will bring you into **insert mode**. In insert mode, you can type text. The escape key will bring you back to command mode. When in insert mode, vi will display **-- INSERT --** in the bottom left corner.

5.2.2. Start typing (a A i I o O)

The difference between a A i I o and O is the location where you can start typing. a will append after the current character and A will append at the end of the line. i will insert before the current character and I will insert at the beginning of the line. o will put you in a new line after the current line and O will put you in a new line before the current line.

5.2.3. Replace and delete a character (r x)

When in command mode (it doensn't hurt to hit the escape key more than once) you can use the x key to delete the current character. Big X key (or shift x) will delete the character left of the cursor. Also when in command mode, you can use the r key to replace one single character. The r key will bring you in insert mode for just one key press, and will return you immediately to command mode.

5.2.4. Undo and repeat (u .)

When in command mode, you can undo your mistakes with u. You can do your mistakes twice with . (in other words the . will repeat your last command).

5.2.5. Cut, copy and paste a line (dd yy p P)

When in command mode, dd will cut the current line. yy will copy the current line. You can paste the last copied or cut line after (p) or before (P) the current line.

5.2.6. Cut, copy and paste lines (3dd 2yy)

When in command mode, before typing dd or yy, you can type a number to repeat the command a number of times. Thus, 5dd will cut 5 lines and 4yy will copy (yank) 4 lines. That last one will be noted by vi in the bottom left corner as "4 line yanked".

5.2.7. Start and end of a line (0 or ^ and $)

When in command mode, the 0 and the caret ^ will bring you to the start of the current line, whereas the $ will put the cursor at the end of the current line. You can add 0 and $ to the d command, d0 will delete every character between the current character and the start of the line. Likewise d$ will delete everything from the current character till the end of the line. Similarly y0 and y$ will yank till start and end of the current line.

5.2.8. Join two lines (J)

When in command mode, pressing J will append the next line to the current line.

5.2.9. Words (w b)

When in command mode, w will jump you to the next word, and b will get you to the previous word. w and b can also be combined with d and y to copy and cut words (dw db yw yb).

5.2.10. Save (or not) and exit (:w :q :q!)

Pressing the colon : will allow you to give instructions to vi. :w will write (save) the file, :q will quit un unchanged file without saving, :q! will quit vi discarding changes. :wq will save and quit and is the same as typing ZZ in command mode.

5.2.11. Searching (/ ?)

When in command mode typing / will allow you to search in vi for strings (can be a regular expression). Typing /foo will do a forward search for the string foo, typing ? bar will do a backward search for bar.

5.2.12. Replace all (:1,$ s/foo/bar/g)

To replace all occurences of the string foo in bar, first switch to ex mode with : . Then tell vi which lines to use, for example 1,$ will do the replace all from the first to the last line. You can write 1,5 to only process the first five lines. The s/foo/bar/g will replace all occurences of foo with bar.

5.2.13. Reading files (:r :r !cmd)

When in command mode, :r foo will read the file named foo, :r !foo will execute the command foo. The result will be put at the current location. Thus :r !ls will put a listing of the current directory in your textfile.

5.2.14. Setting options

Some options that you can set in vim.

```
:set number  ( also try :se nu )
:set nonumber
:syntax on
:syntax off
:set all  (list all options)
:set tabstop=8
:set tx   (CR/LF style endings)
:set notx
```

You can set these options (and much more) in **~/.vimrc**

```
paul@barry:~$ cat ~/.vimrc
set number
paul@barry:~$
```

5.3. Practice

1. Start the vimtutor and do some or all of the exercises.

2. What 3 key combination in command mode will duplicate the current line.

3. What 3 key combination in command mode will switch two lines' place (line five becomes line six and line six becomes line five).

4. What 2 key combination in command mode will switch a character's place with the next one.

5. vi can understand macro's. A macro can be recorded with q followed by the name of the macro. So qa will record the macro named a. Pressing q again will end the

recording. You can recall the macro with @ followed by the name of the macro. Try this example: i 1 'Escape Key' qa yyp 'Ctrl a' q 5@a (Ctrl a will increase the number with one).

6. Copy /etc/passwd to your ~/passwd. Open the last one in vi and press Ctrl v. Use the arrow keys to select a Visual Block, you can copy this with y or delete it with d. Try pasting it.

7. What does dwwP do when you are at the beginning of a word in a sentence ?

5.4. Solutions to the Practice

2. yyp

3. ddp

4. xp

7. dwwP can switch the current word with the next word.

Chapter 6. Introduction to Users

6.1. Users

6.1.1. User management

User management on any Unix can essentialy be done in three ways. You can use the **graphical** tools provided by your distribution. These tools have a look and feel that depends on the distribution. If you are a novice linux user on your home system, then use the graphical tool that is provided by your distribution. This will make sure that you do not run into problems.

Another option is to use **command line tools** like useradd, usermod, gpasswd, passwd and others. Server administrators are likely to use these tools, since they are familiar and very similar accross many different distributions. This chapter will focus on these command line tools.

A third and rather extremist way is to **edit the local configuration files** directly using vi (or vipw/vigr). Do not attempt this as a novice on production systems!

6.1.2. /etc/passwd

The local user database on Linux (and on most Unixes) is **/etc/passwd**.

```
[root@RHEL5 ~]# tail /etc/passwd
inge:x:518:524:art dealer:/home/inge:/bin/ksh
ann:x:519:525:flute player:/home/ann:/bin/bash
frederik:x:520:526:rubius poet:/home/frederik:/bin/bash
steven:x:521:527:roman emperor:/home/steven:/bin/bash
pascale:x:522:528:artist:/home/pascale:/bin/ksh
geert:x:524:530:kernel developer:/home/geert:/bin/bash
wim:x:525:531:master damuti:/home/wim:/bin/bash
sandra:x:526:532:radish stresser:/home/sandra:/bin/bash
annelies:x:527:533:sword fighter:/home/annelies:/bin/bash
laura:x:528:534:art dealer:/home/laura:/bin/ksh
```

As you can see, this file contains seven columns seperated by a colon. The columns contain the username, an x, the user id, the primary group id, a description, the name of the home directory and the login shell.

6.1.3. root

The **root** user also called the **superuser** is the most powerful account on your Linux system. This user can do almost everything, including the creation of other users. The root user always has userid 0 (regardless of the name of the account).

```
[root@RHEL5 ~]# head -1 /etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
```

6.1.4. useradd

You can add users with the **useradd** command. The example below shows how to add a user named yanina (last parameter), and at the same time forcing the creation of the home directory (-m), setting the name of the home directory (-d) and setting a description (-c).

```
[root@RHEL5 ~]# useradd -m -d /home/yanina -c "yanina wickmayer" yanina
[root@RHEL5 ~]# tail -1 /etc/passwd
yanina:x:529:529:yanina wickmayer:/home/yanina:/bin/bash
```

The user named yanina received userid 529 and **primary group** id 529.

6.1.5. default useradd options

Both Red Hat Enterprise Linux and Debian/Ubuntu have a file called **/etc/default/useradd** that contains some default user options. Besides using cat to display this file, you can also use **useradd -D**.

```
[root@RHEL4 ~]# useradd -D
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/bash
SKEL=/etc/skel
```

6.1.6. userdel

You can delete the user yanina with **userdel**. The -r option of userdel will also remove the home directory.

```
[root@RHEL5 ~]# userdel -r yanina
```

6.1.7. usermod

You can modify the properties of a user with the **usermod** command. This example uses **usermod** to change the description of the user harry.

```
[root@RHEL4 ~]# tail -1 /etc/passwd
harry:x:516:520:harry potter:/home/harry:/bin/bash
[root@RHEL4 ~]# usermod -c 'wizard' harry
[root@RHEL4 ~]# tail -1 /etc/passwd
harry:x:516:520:wizard:/home/harry:/bin/bash
```

6.2. Identify yourself

6.2.1. whoami

The **whoami** command exists to tell you your username.

```
[root@RHEL5 ~]# whoami
root
[root@RHEL5 ~]# su - paul
[paul@RHEL5 ~]$ whoami
paul
```

6.2.2. who

The **who** command will give you information about who is logged on to the system.

```
[paul@RHEL5 ~]$ who
root     tty1         2008-06-24 13:24
sandra   pts/0        2008-06-24 14:05 (192.168.1.34)
paul     pts/1        2008-06-24 16:23 (192.168.1.37)
```

6.2.3. who am i

With **who am i** the who command will display only the line pointing to your current session.

```
[paul@RHEL5 ~]$ who am i
paul     pts/1        2008-06-24 16:23 (192.168.1.34)
```

6.2.4. w

The **w** command shows you who is logged on and what they are doing.

```
$ w
 05:13:36 up 3 min,  4 users,  load average: 0.48, 0.72, 0.33
USER   TTY    FROM          LOGIN@   IDLE  JCPU  PCPU WHAT
root   tty1   -             05:11    2.00s 0.32s 0.27s find / -name shad
inge   pts/0  192.168.1.33  05:12    0.00s 0.02s 0.02s -ksh
laura  pts/1  192.168.1.34  05:12   46.00s 0.03s 0.03s -bash
paul   pts/2  192.168.1.34  05:13   25.00s 0.07s 0.04s top
```

6.2.5. id

The **id** command will give you your user id, primary group id and a list of the groups that you belong to.

```
root@laika:~# id
uid=0(root) gid=0(root) groups=0(root)
root@laika:~# su - brel
brel@laika:~$ id
uid=1001(brel) gid=1001(brel) groups=1001(brel),1008(chanson),11578(wolf)
```

6.3. Passwords

6.3.1. passwd

Passwords of users can be set with the **passwd** command. Users will have to provide their old password before entering the new one twice.

```
[harry@RHEL4 ~]$ passwd
Changing password for user harry.
Changing password for harry
(current) UNIX password:
New UNIX password:
BAD PASSWORD: it's WAY too short
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
[harry@RHEL4 ~]$
```

As you can see, the passwd tool will do some basic verification to prevent users from using too simple passwords. The root user does not have to follow these rules (there will be a warning though). The root user also does not need to provide the old password before entering the new password twice.

6.3.2. /etc/shadow

User passwords are encrypted and kept in **/etc/shadow**. The /etc/shadow file is read only, and can only be read by root. We will see in the file permissions section how it is possible for users to change their password. For now, you will have to know that users can change their password with the **/usr/bin/passwd** command.

```
[root@RHEL5 ~]# tail /etc/shadow
inge:$1$yWMSimOV$YsYvcVKqByFVYLKnU3ncd0:14054:0:99999:7:::
ann:!!:14054:0:99999:7:::
frederik:!!:14054:0:99999:7:::
steven:!!:14054:0:99999:7:::
pascale:!!:14054:0:99999:7:::
geert:!!:14054:0:99999:7:::
wim:!!:14054:0:99999:7:::
sandra:!!:14054:0:99999:7:::
annelies:!!:14054:0:99999:7:::
laura:$1$Tvby1Kpa$lL.WzgobujUS3LClIRmdv1:14054:0:99999:7:::
```

The /etc/shadow file contains nine colums, seperated by a colon. The nine fields contain (from left to right) the user name, the encrypted password (note that only inge and laura have an encrypted password), the day the password was last changed (day 1 is January 1, 1970), number of days the password has to be left unchanged, password expiry day, warning number of days before password expiry, number of days after expiry before disabling the account, the day the account was disabled (since 1970 again). The last field has no meaning yet.

6.3.3. password encryption

6.3.3.1. encryption with passwd

Passwords are stored in an encrypted format. This encryption is done by the **crypt** function. The easiest (and recommended) way to add a user with a password to the system is to add the user with the **useradd -m user** command, and then set the user's password with **passwd**.

```
[root@RHEL4 ~]# useradd -m xavier
[root@RHEL4 ~]# passwd xavier
Changing password for user xavier.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
[root@RHEL4 ~]#
```

6.3.3.2. encryption with openssl

Another way to create users with a password is to use the -p option of useradd, but that option requires an encrypted password. You can generate this encrypted password with the **openssl passwd** command.

```
[root@RHEL4 ~]# openssl passwd stargate
ZZNX16QZVgUQg
[root@RHEL4 ~]# useradd -m -p ZZNX16QZVgUQg mohamed
```

6.3.3.3. encryption with crypt

A third option is to create your own C program using the crypt function, and compile this into a command.

```
[paul@laika ~]$ cat MyCrypt.c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char** argv)
{
  printf("%s\n", crypt(argv[1], argv[2] ));
  return 0;
```

```
}
```

This little program can be compiled with g++ like this.

```
[paul@laika ~]$ g++ MyCrypt.c -o MyCrypt -lcrypt
```

To use it, we need to give two parameters to MyCript. The first is the unencrypted password, the second is the salt. The salt is used to perturb the encryption algorithm in one of 4096 different ways. This variation prevents two users with the same password from having the same entry in /etc/shadow.

```
paul@laika:~$ ./MyCrypt stargate 12
12L4FoTS3/k9U
paul@laika:~$ ./MyCrypt stargate 01
01Y.yPnlQ6R.Y
paul@laika:~$ ./MyCrypt stargate 33
330asFUbzgVeg
paul@laika:~$ ./MyCrypt stargate 42
42XFxoT4R75gk
```

Did you notice that the first two characters of the password are the salt ?

The standard output of the crypt function is using the DES algorithm, which is old and can be cracked in minutes. A better method is to use MD5 passwords, which can be recognized by a salt starting with 1.

```
paul@laika:~$ ./MyCrypt stargate '$1$12'
$1$12$xUIQ4116Us.Q5Osc2Khbm1
paul@laika:~$ ./MyCrypt stargate '$1$01'
$1$01$yNs8brjp4b4TEw.v9/IlJ/
paul@laika:~$ ./MyCrypt stargate '$1$33'
$1$33$tLh/Ldy2wskdKAJR.Ph4M0
paul@laika:~$ ./MyCrypt stargate '$1$42'
$1$42$Hb3nvP0KwHSQ7fQmIlY7R.
```

The MD5 salt can be up to eight characters long. The salt is displayed in /etc/shadow between the second and third $, so never use the password as the salt!

```
paul@laika:~$ ./MyCrypt stargate '$1$stargate'
$1$stargate$qqxoLqiSVNvGr5ybMxEVM1
```

6.3.4. password defaults

6.3.4.1. /etc/login.defs

The **/etc/login.defs** file contains some default settings for user passwords like password aging and length settings. (You will also find the numerical limits of user id's and group id's and whether or not a home directory should be created by default).

```
[root@RHEL4 ~]# grep -i pass /etc/login.defs
# Password aging controls:
# PASS_MAX_DAYS  Maximum number of days a password may be used.
# PASS_MIN_DAYS  Minimum number of days allowed between password changes.
# PASS_MIN_LEN   Minimum acceptable password length.
# PASS_WARN_AGE  Number of days warning given before a password expires.
PASS_MAX_DAYS   99999
PASS_MIN_DAYS   0
PASS_MIN_LEN    5
PASS_WARN_AGE   7
```

6.3.4.2. chage

The **chage** command can be used to set an expiration date for a user account (-E), set a mimimum (-m) and maximum (-M) password age, a password expiration date, and set the number of warning days before the password expiration date. A lot of this functionality is also available via the passwd command. The -l option of chage will list these settings for a user.

```
[root@RHEL4 ~]# chage -l harry
Minimum:        0
Maximum:        99999
Warning:        7
Inactive:       -1
Last Change:            Jul 23, 2007
Password Expires:       Never
Password Inactive:      Never
Account Expires:        Never
[root@RHEL4 ~]#
```

6.3.5. disabling a password

Passwords in /etc/shadow cannot begin with an exclamation mark. When the second field in /etc/passwd starts with an exclamation mark, then the password can not be used.

Using this feature is often called **locking**, **disabling** or **suspending** a user account. Bisides vi (or vipw) you can also accomplish this with **usermod**.

The first line in the next screenshot will disable the password of user harry, making it impossible for harry to authenticate using this password.

```
[root@RHEL4 ~]# usermod -L harry
[root@RHEL4 ~]# tail -1 /etc/shadow
harry:!$1$143TO9IZ$RLm/FpQkpDrV4/Tkhku5e1:13717:0:99999:7:::
```

The root user (and users with sudo rights on su) will still be able to su to harry (because the password is not needed here). Note also that harry will still be able to login if he set up passwordless ssh!

```
[root@RHEL4 ~]# su - harry
[harry@RHEL4 ~]$
```

You can unlock the account again with **usermod -U**.

Watch out for tiny differences in the command line options of passwd, usermod and useradd on different distributions! Verify the local files when using features like 'disabling', 'suspending' or 'locking' users and passwords!

6.3.6. editing local files

If after knowing all these commands for password management you still want to edit the /etc/passwd or /etc/shadow manually, then use **vipw** instead of vi(m) directly. The vipw tool will do proper locking of the file.

```
[root@RHEL5 ~]# vipw /etc/passwd
vipw: the password file is busy (/etc/ptmp present)
```

6.4. About Home Directories

6.4.1. creating home directories

The easy way to create a home directory is to supply the -m option with **useradd** (it is likely set as a default option on Linux).

The not so easy way is to create a home directory manually with **mkdir**, which also requires setting the owner and the permissions on the directory with **chmod** and **chown** (both commands are discussed in detail in another chapter).

```
[root@RHEL5 ~]# mkdir /home/laura
[root@RHEL5 ~]# chown laura:laura /home/laura
[root@RHEL5 ~]# chmod 700 /home/laura
[root@RHEL5 ~]# ls -ld /home/laura/
drwx------ 2 laura laura 4096 Jun 24 15:17 /home/laura/
```

6.4.2. /etc/skel/

When using useradd with the -m option, then the **/etc/skel/** directory is copied to the newly created home directory. The /etc/skel/ directory contains some (usually hidden) files that contain profile settings and default values for applications. In this way /etc/skel/ serves as a default home directory and as a default user profile.

```
[root@RHEL5 ~]# ls -la /etc/skel/
total 48
```

```
drwxr-xr-x  2 root root  4096 Apr  1 00:11 .
drwxr-xr-x 97 root root 12288 Jun 24 15:36 ..
-rw-r--r--  1 root root    24 Jul 12  2006 .bash_logout
-rw-r--r--  1 root root   176 Jul 12  2006 .bash_profile
-rw-r--r--  1 root root   124 Jul 12  2006 .bashrc
```

6.4.3. deleting home directories

The -r option of userdel will make sure that the home directory is deleted together with the user account.

```
[root@RHEL5 ~]# ls -ld /home/wim/
drwx------ 2 wim wim 4096 Jun 24 15:19 /home/wim/
[root@RHEL5 ~]# userdel -r wim
[root@RHEL5 ~]# ls -ld /home/wim/
ls: /home/wim/: No such file or directory
```

6.5. User Shell

6.5.1. login shell

The **/etc/passwd** file determines the login shell for the user. In the screenshot below you can see that user annelies receives the /bin/bash shell, and user laura receives the /bin/ksh shell when they login.

```
[root@RHEL5 ~]# tail -2 /etc/passwd
annelies:x:527:533:sword fighter:/home/annelies:/bin/bash
laura:x:528:534:art dealer:/home/laura:/bin/ksh
```

You can use the usermod command to change the shell for a user.

```
[root@RHEL5 ~]# usermod -s /bin/bash laura
[root@RHEL5 ~]# tail -1 /etc/passwd
laura:x:528:534:art dealer:/home/laura:/bin/bash
```

6.5.2. chsh

Users can change their own login shell with the **chsh** command. User harry here is first obtaining a list of available shells (the user could have done a cat **/etc/shells**) and then changes his login shell to the **Korn shell** (/bin/ksh). At the next login, harry will default into ksh instead of bash.

```
[harry@RHEL4 ~]$ chsh -l
/bin/sh
/bin/bash
/sbin/nologin
```

```
/bin/ash
/bin/bsh
/bin/ksh
/usr/bin/ksh
/usr/bin/pdksh
/bin/tcsh
/bin/csh
/bin/zsh
[harry@RHEL4 ~]$ chsh -s /bin/ksh
Changing shell for harry.
Password:
Shell changed.
[harry@RHEL4 ~]$
```

6.6. Switch users with su

The **su** command allows a user to run a shell as another user. Running a shell as another user requires that you know the password of the other user, unless you are root. The root user can become any other user without knowing the user's password.

```
[paul@RHEL4b ~]$ su harry
Password:
[harry@RHEL4b paul]$ su root
Password:
[root@RHEL4b paul]# su serena
[serena@RHEL4b paul]$
```

By default, the su command keeps the same shell environment. To become another user and also get the target user's environment, issue the **su -** command followed by the target username.

```
[paul@RHEL4b ~]$ su - harry
Password:
[harry@RHEL4b ~]$
```

When no username is provided to su or su - then the command will assume root is the target.

```
[harry@RHEL4b ~]$ su -
Password:
[root@RHEL4b ~]#
```

6.7. Run a program as another user

6.7.1. About sudo

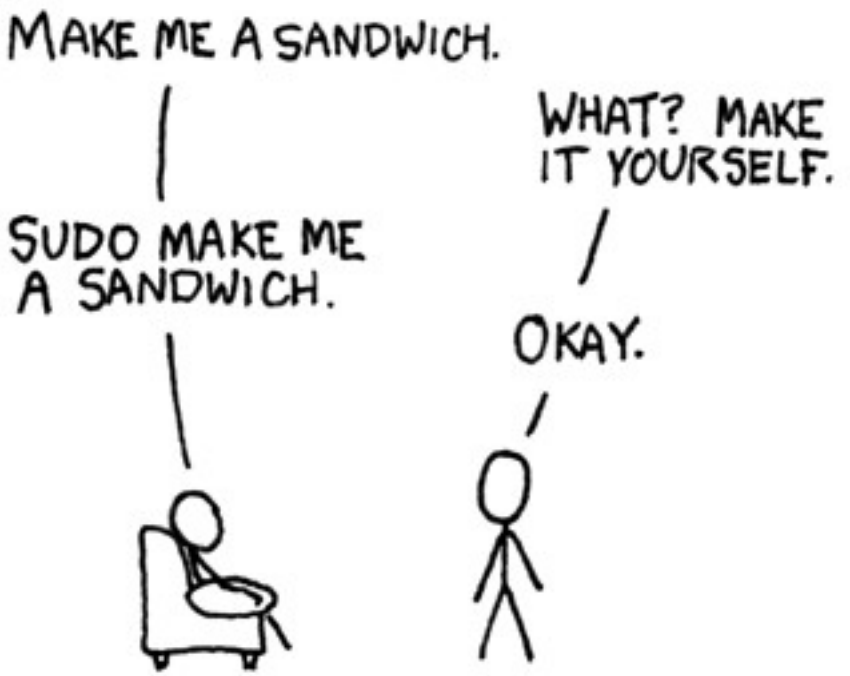
The sudo program allows a user to start a program with the credentials of another user. Before this works, the system administrator has to set up the **/etc/sudoers** file.

This can be useful to delegate administrative tasks to another user (without giving the root password).

The screenshot below shows the usage of sudo. User paul received the right to run useradd with the credentials of root. This allows paul to create new users on the system without becoming root and without knowing the root password.

```
paul@laika:~$ useradd -m inge
useradd: unable to lock password file
paul@laika:~$ sudo useradd -m inge
[sudo] password for paul:
paul@laika:~$
```

Image copied from **xkcd.com**.



6.7.2. setuid on sudo

The sudo binary has the setuid bit set, so any user can run it with effective userid set as root.

```
paul@laika:~$ ls -l `which sudo`
-rwsr-xr-x 2 root root 107872 2008-05-15 02:41 /usr/bin/sudo
paul@laika:~$
```

6.7.3. visudo

Check the man page of **visudo** before playing with the /etc/sudoers file.

6.7.4. sudo su

On some linux systems like Ubuntu and Kubuntu, the root user does not have a password set. This means that it is not possible to login as root (extra security). To perform tasks as root, the first user is given all **sudo rights** via the **/etc/sudoers**. In fact all users that are members of the admin group can use sudo to run all commands as root.

```
root@laika:~# grep admin /etc/sudoers
# Members of the admin group may gain root privileges
%admin ALL=(ALL) ALL
```

The end result of this is that the user can type **sudo su -** and become root without having to enter the root password. The sudo command does require you to enter your own password. Thus the password prompt in the screenshot below is for sudo, not for su.

```
paul@laika:~$ sudo su -
Password:
root@laika:~#
```

6.8. Practice Users

1. Create the users Serena Williams, Venus Williams and Justine Henin. all of them with password set to stargate, with username (lowercase!) as their first name, and their full name in the comment. Verify that the users and their home directory are properly created.

2. Create a user called kornuser, give him the Korn shell (/bin/ksh) as his default shell. Log on with this user (on a command line or in a tty).

3. Create a user named Einstime without home directory, give him /bin/date as his default logon shell. What happens when you log on with this user ? Can you think of a useful real world example for changing a user's login shell to an application ?

4. Try the commands who, whoami, who am i, w, id, echo $USER $UID .

5a. Lock the Venus user account with usermod.

5b. Use passwd -d to disable the serena password. Verify the serena line in /etc/shadow before and after disabling.

5c. What is the difference between locking a user account and disabling a user account's password ?

6. As root change the password of Einstime to stargate.

7. Now try changing the password of serena to serena as serena.

8. Make sure every new user needs to change his password every 10 days.

9. Set the warning number of days to four for the kornuser.

10a. Set the password of two seperate users to stargate. Look at the encrypted stargate's in /etc/shadow and explain.

10b. Take a backup as root of /etc/shadow. Use vi to copy an encrypted stargate to another user. Can this other user now log on with stargate as a password ?

11. Put a file in the skeleton directory and check whether it is copied to user's home directory. When is the skeleton directory copied ?

12. Why use vipw instead of vi ? What could be the problem when using vi or vim ?

13. Use chsh to list all shells, and compare to cat /etc/shells. Change your login shell to the Korn shell, log out and back in. Now change back to bash.

14. Which useradd option allows you to name a home directory ?

15. How can you see whether the password of user harry is locked or unlocked ? Give a solution with grep and a solution with passwd.

6.9. Solutions to the practice

1. Create the users Serena Williams, Venus Williams and Justine Henin. all of them with password set to stargate, with username (lowercase) as their first name, and their full name in the comment. Verify that the users and their home directory are properly created.

useradd -c "Serena Williams" serena ; passwd serena

useradd -c "Venus Williams" venus ; passwd venus

useradd -c "Justine Henin" justine ; passwd justine

tail /etc/passwd ; tail /etc/shadow ; ls /home

Keep user logon names in lowercase!

2. Create a user called kornuser, give him the Korn shell (/bin/ksh) as his default shell. Log on with this user (on a command line or in a tty).

useradd -s /bin/ksh kornuser ; passwd kornuser

3. Create a user named Einstime without home directory, give him /bin/date as his default logon shell. What happens when you log on with this user ? Can you think of a useful real world example for changing a user's login shell to an application ?

useradd -s /bin/date einstime ; passwd einstime

It can be useful when users need to access only one application on the server. Just logging on opens the application for them, and closing the application automatically logs them off.

4. Try the commands who, whoami, who am i, w, id, echo $USER $UID .

who ; whoami ; who am i ; w ; id ; echo $USER $UID

5a. Lock the venus user account with usermod.

usermod -L venus

5b. Use passwd -d to disable the serena password. Verify the serena line in /etc/shadow before and after disabling.

grep serena /etc/shadow; passwd -d serena ; grep serena /etc/shadow

5c. What is the difference between locking a user account and disabling a user account's password ?

Locking will prevent the user from logging on to the system with his password (by putting a ! in front of the password in /etc/shadow). Disbling with passwd will erase the password from /etc/shadow.

6. As root change the password of Einstime to stargate.

Log on as root and type: passwd Einstime

7. Now try changing the password of serena to serena as serena.

log on as serena, then execute: passwd serena... it should fail!

8. Make sure every new user needs to change his password every 10 days.

For an existing user: chage -M 10 serena

For all new users: vi /etc/login.defs (and change PASS_MAX_DAYS to 10)

9. Set the warning number of days to four for the kornuser.

chage -W 4 kornuser

10a. Set the password of two seperate users to stargate. Look at the encrypted stargate's in /etc/shadow and explain.

If you used passwd, then the salt will be different for the two encrypted passwords.

10b. Take a backup as root of /etc/shadow. Use vi to copy an encrypted stargate to another user. Can this other user now log on with stargate as a password ?

Yes.

11. Put a file in the skeleton directory and check whether it is copied to user's home directory. When is the skeleton directory copied ?

When you create a user account with a new home directory.

12. Why use vipw instead of vi ? What could be the problem when using vi or vim ?

vipw will give a warning when someone else is already using vipw on that file.

13. Use chsh to list all shells, and compare to cat /etc/shells. Change your login shell to the Korn shell, log out and back in. Now change back to bash.

On Red Hat Enterprise Linux: chsh -l

14. Which useradd option allows you to name a home directory ?

-d

15. How can you see whether the password of user harry is locked or unlocked ? Give a solution with grep and a solution with passwd.

grep harry /etc/shadow

passwd -S harry

Chapter 7. Introduction to Groups

7.1. About Groups

Users can be listed in groups. Groups allow you to set permissions on the group level instead of setting permissions for every individual users. Every Unix or Linux distribution will have a graphical tool to manage groups. Novice users are adviced to use this graphical tool. More experienced users can use commandline tools to manage users, but be careful: some distribution do not allow the mixed use of GUI and CLI tools to manage groups (YaST in Novell Suse). Senior administrators can edit the relevant files directly with vi or vigr.

7.2. groupadd

Groups can be created with the **groupadd** command. The example below shows the creation of five (empty) groups.

```
root@laika:~# groupadd tennis
root@laika:~# groupadd football
root@laika:~# groupadd snooker
root@laika:~# groupadd formula1
root@laika:~# groupadd salsa
```

7.3. /etc/group

Users can be a member of several groups. Group membership is defined by the **/etc/group** file.

```
root@laika:~# tail -5 /etc/group
tennis:x:1006:
football:x:1007:
snooker:x:1008:
formula1:x:1009:
salsa:x:1010:
root@laika:~#
```

Before the colon is the group's name. The second field is the group's (encrypted) password (can be empty). The third field is the group identification or **GID**. The fourth field is the list of members, these groups have no members.

7.4. usermod

Group membership can be modified with the useradd or **usermod** command.

```
root@laika:~# usermod -a -G tennis inge
root@laika:~# usermod -a -G tennis katrien
```

```
root@laika:~# usermod -a -G salsa katrien
root@laika:~# usermod -a -G snooker sandra
root@laika:~# usermod -a -G formula1 annelies
root@laika:~# tail -5 /etc/group
tennis:x:1006:inge,katrien
football:x:1007:
snooker:x:1008:sandra
formula1:x:1009:annelies
salsa:x:1010:katrien
root@laika:~#
```

Be careful when using usermod to add people to groups. The usermod command will by default **remove** users from the groups that are not listed in the command! Using the **-a** (append) switch prevents this behaviour.

7.5. groupmod

You can change the group name with the **groupmod** command.

```
root@laika:~# groupmod -n darts snooker
root@laika:~# tail -5 /etc/group
tennis:x:1006:inge,katrien
football:x:1007:
formula1:x:1009:annelies
salsa:x:1010:katrien
darts:x:1008:sandra
```

7.6. groupdel

You can permanently remove a group with the **groupdel** command.

```
root@laika:~# groupdel tennis
root@laika:~#
```

7.7. groups

A user can type the **groups** command to see a list of groups where the user belongs to.

```
[harry@RHEL4b ~]$ groups
harry sports
[harry@RHEL4b ~]$
```

7.8. gpasswd

You can delegate control of group membership to another user with the **gpasswd** command. In the example below we delegate permissions to add and remove group

members to the sports group to serena. Then we su to serena and add harry to the sports group.

```
[root@RHEL4b ~]# gpasswd -A serena sports
[root@RHEL4b ~]# su - serena
[serena@RHEL4b ~]$ id harry
uid=516(harry) gid=520(harry) groups=520(harry)
[serena@RHEL4b ~]$ gpasswd -a harry sports
Adding user harry to group sports
[serena@RHEL4b ~]$ id harry
uid=516(harry) gid=520(harry) groups=520(harry),522(sports)
[serena@RHEL4b ~]$ tail -1 /etc/group
sports:x:522:serena,venus,harry
[serena@RHEL4b ~]$
```

Group administrators do not need to be a member of the group. They can even remove themselves from the group, this does not influence their ability to add or remove members.

```
[serena@RHEL4b ~]$ gpasswd -d serena sports
Removing user serena from group sports
[serena@RHEL4b ~]$ exit
```

Information about group administrators is kept in the **/etc/gshadow** file.

```
[root@RHEL4b ~]# tail -1 /etc/gshadow
sports:!::serena:venus,harry
[root@RHEL4b ~]#
```

7.9. vigr

Similar to vipw, the **vigr** must be used to manually edit the /etc/group file, since it will do proper locking of the file. Only experienced senior administrators should use vi or vigr to manage groups.

7.10. Practice: Groups

1. Create the groups tennis, football and sports.

2. In one command, make venus a member of tennis and sports.

3. Rename the football group to foot.

4. Use vi to add serena to the tennis group.

5. Use the id command to verify that serena is a member of tennis.

6. Make someone responsible for managing group membership of foot and sports. Test that it works.

7.11. Solution: Groups

1. Create the groups tennis, football and sports.

groupadd tennis ; groupadd football ; groupadd sports

2. In one command, make venus a member of tennis and sports.

usermod -a -G tennis,sports venus

3. Rename the football group to foot.

groupmod -n foot football

4. Use vi to add serena to the tennis group.

vi /etc/group

5. Use the id command to verify that serena is a member of tennis.

id (and after logoff logon serena should be member)

6. Make someone responsible for managing group membership of foot and sports. Test that it works.

gpasswd -A (to make member)

gpasswd -a (to add member)

Chapter 8. Standard File Permissions

8.1. file ownership

8.1.1. user owner and group owner

The **users** and **groups** of a system can be locally managed in **/etc/passwd** and **/etc/group**, or they can be in a NIS, LDAP or Samba domain. These users and groups can **own** files. Actually, every file has a **user owner** and a **group owner**, as can be seen in the following screenshot.

```
paul@RHELv4u4:~/test$ ls -l
total 24
-rw-rw-r--  1 paul paul  17 Feb  7 11:53 file1
-rw-rw-r--  1 paul paul 106 Feb  5 17:04 file2
-rw-rw-r--  1 paul proj 984 Feb  5 15:38 data.odt
-rw-r--r--  1 root root   0 Feb  7 16:07 stuff.txt
paul@RHELv4u4:~/test$
```

User paul owns three files, two of those are also owned by the group paul, data.odt is owned by the group proj. The root user owns the file stuff.txt, as does the group root.

8.1.2. chgrp

You can change the group owner of a file using the **chgrp** command.

```
root@laika:/home/paul# touch FileForPaul
root@laika:/home/paul# ls -l FileForPaul
-rw-r--r-- 1 root root 0 2008-08-06 14:11 FileForPaul
root@laika:/home/paul# chgrp paul FileForPaul
root@laika:/home/paul# ls -l FileForPaul
-rw-r--r-- 1 root paul 0 2008-08-06 14:11 FileForPaul
```

8.1.3. chown

The user owner of a file can be changed with **chown** command.

```
root@laika:/home/paul# ls -l FileForPaul
-rw-r--r-- 1 root paul 0 2008-08-06 14:11 FileForPaul
root@laika:/home/paul# chown paul FileForPaul
root@laika:/home/paul# ls -l FileForPaul
-rw-r--r-- 1 paul paul 0 2008-08-06 14:11 FileForPaul
```

You can also use chown to change both the user owner and the group owner.

```
root@laika:/home/paul# ls -l FileForPaul
-rw-r--r-- 1 paul paul 0 2008-08-06 14:11 FileForPaul
root@laika:/home/paul# chown root:project42 FileForPaul
root@laika:/home/paul# ls -l FileForPaul
-rw-r--r-- 1 root project42 0 2008-08-06 14:11 FileForPaul
```

8.2. special files

When you use **ls -l**, then you can see, for each file, ten characters before the user and group owner. The first character tells us the type of file. Regular files get a **-**, directories get a **d**, symbolic links are shown with an **l**, pipes get a **p**, character devices a **c**, block devices a **b** and sockets an **s**.

Table 8.1. Unix special files

first character	file type
-	normal file
d	directory
l	symbolic link
p	named pipe
b	block device
c	character device
s	socket

8.3. permissions

The nine characters following the file type denote the permissions in three triplets. A permission can be **r** for read access, **w** for write access and **x** for execute. You need the r permission to list (ls) the contents of a directory and x permission to enter (cd) a directory, and you need the w permission to create files in or remove files from a directory.

Table 8.2. standard Unix file permissions

permission	on a file	on a directory
r (read)	read file contents (cat)	read directory contents (ls)
w (write)	change file contents (vi)	create files in (touch)
x (execute)	execute the file	enter the directory (cd)

Some example combinations on files and directories in this screenshot. The name of the file explains the permissions.

```
paul@laika:~/perms$ ls -lh
total 12K
drwxr-xr-x 2 paul paul 4.0K 2007-02-07 22:26 AllEnter_UserCreateDelete
-rwxrwxrwx 1 paul paul    0 2007-02-07 22:21 EveryoneFullControl.txt
-r--r----- 1 paul paul    0 2007-02-07 22:21 OnlyOwnersRead.txt
-rwxrwx--- 1 paul paul    0 2007-02-07 22:21 OwnersAll_RestNothing.txt
dr-xr-x--- 2 paul paul 4.0K 2007-02-07 22:25 UserAndGroupEnter
dr-x------ 2 paul paul 4.0K 2007-02-07 22:25 OmlyUserEnter
paul@laika:~/perms$
```

It is important to know that the first triplet represents the **user owner**, the second is the **group owner**, and the third triplet is all the **other** users that are not the user owner and are not a member of the group owner.

8.4. setting permissions (chmod)

Permissions can be changed with **chmod**. The first example gives the user owner execute permissions.

```
paul@laika:~/perms$ ls -l permissions.txt
-rw-r--r-- 1 paul paul 0 2007-02-07 22:34 permissions.txt
paul@laika:~/perms$ chmod u+x permissions.txt
paul@laika:~/perms$ ls -l permissions.txt
-rwxr--r-- 1 paul paul 0 2007-02-07 22:34 permissions.txt
```

This example removes the group owners read permission.

```
paul@laika:~/perms$ chmod g-r permissions.txt
paul@laika:~/perms$ ls -l permissions.txt
-rwx---r-- 1 paul paul 0 2007-02-07 22:34 permissions.txt
```

This example removes the others read permission.

```
paul@laika:~/perms$ chmod o-r permissions.txt
paul@laika:~/perms$ ls -l permissions.txt
-rwx------ 1 paul paul 0 2007-02-07 22:34 permissions.txt
```

This example gives all of them the write permission.

```
paul@laika:~/perms$ chmod a+w permissions.txt
paul@laika:~/perms$ ls -l permissions.txt
-rwx-w--w- 1 paul paul 0 2007-02-07 22:34 permissions.txt
```

You don't even have to type the a.

```
paul@laika:~/perms$ chmod +x permissions.txt
paul@laika:~/perms$ ls -l permissions.txt
-rwx-wx-wx 1 paul paul 0 2007-02-07 22:34 permissions.txt
```

You can also set explicit permissions.

```
paul@laika:~/perms$ chmod u=rw permissions.txt
paul@laika:~/perms$ ls -l permissions.txt
-rw--wx-wx 1 paul paul 0 2007-02-07 22:34 permissions.txt
```

Feel free to make any kind of combinations.

```
paul@laika:~/perms$ chmod u=rw,g=rw,o=r permissions.txt
paul@laika:~/perms$ ls -l permissions.txt
-rw-rw-r-- 1 paul paul 0 2007-02-07 22:34 permissions.txt
```

Even the fishy combinations are accepted by chmod.

```
paul@laika:~/perms$ chmod u=rwx,ug+rw,o=r permissions.txt
paul@laika:~/perms$ ls -l permissions.txt
-rwxrw-r-- 1 paul paul 0 2007-02-07 22:34 permissions.txt
```

8.5. setting octal permissions

Most Unix administrators will use the **old school** octal system to talk about and set permissions. Look at the triplet bitwise, equaling r to 4, w to 2 and x to 1.

Table 8.3. Octal permissions

binary	octal	permission
000	0	---
001	1	--x
010	2	-w-
011	3	-wx
100	4	r--
101	5	r-x
110	6	rw-
111	7	rwx

This makes **777** equal to rwxrwxrwx and by the same logic has 654 mean rw-r-xr-- . The **chmod** command will accept these numbers.

```
paul@laika:~/perms$ chmod 777 permissions.txt
paul@laika:~/perms$ ls -l permissions.txt
-rwxrwxrwx 1 paul paul 0 2007-02-07 22:34 permissions.txt
paul@laika:~/perms$ chmod 664 permissions.txt
paul@laika:~/perms$ ls -l permissions.txt
-rw-rw-r-- 1 paul paul 0 2007-02-07 22:34 permissions.txt
paul@laika:~/perms$ chmod 750 permissions.txt
paul@laika:~/perms$ ls -l permissions.txt
```

```
-rwxr-x--- 1 paul paul 0 2007-02-07 22:34 permissions.txt
```

8.6. umask

When creating a file or directory, a set of default permissions are applied. These default permissions are determined by the **umask**. The umask specifies permissions that you do not want set by default. You can display the umask with the umask command.

```
[Harry@RHEL4b ~]$ umask
0002
[Harry@RHEL4b ~]$ touch test
[Harry@RHEL4b ~]$ ls -l test
-rw-rw-r--  1 Harry Harry 0 Jul 24 06:03 test
[Harry@RHEL4b ~]$
```

As you can see, the file is also not executable by default. This is a general security feature among Unixes, newly created files are never executable by default. You have to explicitely do a **chmod +x** to make a file executable. This also means that the 1 bit in the umask has no meaning, a umask of 0022 is the same as 0033.

8.7. Practice: File Permissions

1. As normal user, create a directory ~/permissions. Create a file owned by yourself in there.

2. Copy a file owned by root from /etc/ to your permissions dir, who owns this file now ?

3. As root, create a file in the users ~/permissions directory.

4. As normal user, look at who owns this file created by root.

5. Change the ownership of all files in ~/permissions to yourself.

6. Make sure you have all rights to these files, and others can only read.

7. With chmod, is 770 the same as rwxrwx--- ?

8. With chmod, is 664 the same as r-xr-xr-- ?

9. With chmod, is 400 the same as r-------- ?

10. With chmod, is 734 the same as rwxr-xr-- ?

11a. Display the umask in octal and in symbolic form.

11b. Set the umask to 077, but use the symbolic format to set it. Verify that this works.

12. Create a file as root, give only read to others. Can a normal user read this file ? Test writing to this file with vi.

13a. Create a file as normal user, give only read to others. Can another normal user read this file ? Test writing to this file with vi.

13b. Can root read this file ? Can root write to this file with vi ?

14. Create a directory that belongs to a group, where every member of that group can read and write to files, and create files. Make sure that people can only delete their own files.

8.8. Solutions: File Permissions

1. As normal user, create a directory ~/permissions. Create a file owned by yourself in there.

```
mkdir ~/permissions ; touch ~/permissions/myfile.txt
```

2. Copy a file owned by root from /etc/ to your permissions dir, who owns this file now ?

```
cp /etc/hosts ~/permissions/
```

The copy is owned by you.

3. As root, create a file in the users ~/permissions directory.

```
(become root)# touch /home/username/permissions/rootfile
```

4. As normal user, look at who owns this file created by root.

```
ls -l ~/permissions
```

The file created by root is owned by root.

5. Change the ownership of all files in ~/permissions to yourself.

```
chown user ~/permissions/*
```

You cannot become owner of the file that belongs to root.

6. Make sure you have all rights to these files, and others can only read.

```
chmod 644 (on files)
```

```
chmod 755 (on directories)
```

7. With chmod, is 770 the same as rwxrwx--- ?

yes

8. With chmod, is 664 the same as r-xr-xr-- ?

No

9. With chmod, is 400 the same as r-------- ?

yes

10. With chmod, is 734 the same as rwxr-xr-- ?

no

11a. Display the umask in octal and in symbolic form.

```
umask ; umask -S
```

11b. Set the umask to 077, but use the symbolic format to set it. Verify that this works.

```
umask -S u=rwx,go=
```

12. Create a file as root, give only read to others. Can a normal user read this file ? Test writing to this file with vi.

```
(become root)
# echo hello > /home/username/root.txt
# chmod 744 /home/username/root.txt
(become user)
vi ~/root.txt
```

13a. Create a file as normal user, give only read to others. Can another normal user read this file ? Test writing to this file with vi.

```
echo hello > file ; chmod 744 file
```

Yes, others can read this file

13b. Can root read this file ? Can root write to this file with vi ?

Yes, root can read and write to this file. Permissions do not apply to root.

14. Create a directory that belongs to a group, where every member of that group can read and write to files, and create files. Make sure that people can only delete their own files.

```
mkdir /home/project42 ; groupadd project42
chgrp project42 /home/project42 ; chmod 775 /home/project42
```

You can not yet do the last part of this exercise...

8.9. Sticky bit on directory

You can set the **sticky bit** on a directory to prevent users from removing files that they do not own as a user owner. The sticky bit is displayed at the same location as the x permission for others. The sticky bit is represented by a **t** (meaning x is also there) or a **T** (when there is no x for others).

```
root@RHELv4u4:~# mkdir /project55
root@RHELv4u4:~# ls -ld /project55
drwxr-xr-x  2 root root 4096 Feb  7 17:38 /project55
root@RHELv4u4:~# chmod +t /project55/
root@RHELv4u4:~# ls -ld /project55
drwxr-xr-t  2 root root 4096 Feb  7 17:38 /project55
root@RHELv4u4:~#
```

The sticky bit can also be set with octal permissions, it is binary 1 in the first of four triplets.

```
root@RHELv4u4:~# chmod 1775 /project55/
root@RHELv4u4:~# ls -ld /project55
drwxrwxr-t  2 root root 4096 Feb  7 17:38 /project55
root@RHELv4u4:~#
```

8.10. SetGID bit on directory

The **SetGID** can be used on directories to make sure that all files inside the directory are group owned by the group owner of the directory. The SetGID bit is displayed at the same location as the x permission for group owner. The SetGID bit is represented by an **s** (meaning x is also there) or a **S** (when there is no x for the group owner). Like this example shows, even though root does not belong to the group proj55, the files created by root in /project55 will belong to proj55 when the SetGID is set.

```
root@RHELv4u4:~# groupadd proj55
root@RHELv4u4:~# chown root:proj55 /project55/
root@RHELv4u4:~# chmod 2775 /project55/
root@RHELv4u4:~# touch /project55/fromroot.txt
root@RHELv4u4:~# ls -ld /project55/
drwxrwsr-x  2 root proj55 4096 Feb  7 17:45 /project55/
root@RHELv4u4:~# ls -l /project55/
total 4
-rw-r--r--  1 root proj55 0 Feb  7 17:45 fromroot.txt
root@RHELv4u4:~#
```

You can use the **find** command to find all **setgid** programs.

```
paul@laika:~$ find / -type d -perm -2000 2> /dev/null
/var/log/mysql
/var/log/news
/var/local
...
```

8.11. Practice: sticky and setGID on directory

1. Set up a directory, owned by the group sports.

2. Members of the sports group should be able to create files in this directory.

3. All files created in this directory should be group-owned by the sports group.

4. Users should be able to delete only their own user-owned files.

5. Test that this works!

6. If time permits (or if you are waiting for other students to finish this practice), read about file attributes in the man page of chattr and lsattr. Try setting the i attribute on a file and test that it works.

8.12. Solution: sticky and setGID on directory

1. Set up a directory, owned by the group sports.

```
groupadd sports ; mkdir /home/sports ; chown root:sports /home/sports
```

2. Members of the sports group should be able to create files in this directory.

```
chmod 770 /home/sports
```

3. All files created in this directory should be group-owned by the sports group.

```
chmod 2770 /home/sports
```

4. Users should be able to delete only their own user-owned files.

```
chmod +t /home/sports
```

5. Test that this works!

Log in with different users (group members and others and root), create files and watch the permissions. Try changing and deleting files...

6. If time permits (or if you are waiting for other students to finish this practice), read about file attributes in the man page of chattr and lsattr. Try setting the i attribute on a file and test that it works.

```
paul@laika:~$ sudo su -
[sudo] password for paul:
root@laika:~# mkdir attr
```

```
root@laika:~# cd attr/
root@laika:~/attr# touch file42
root@laika:~/attr# lsattr
------------------ ./file42
root@laika:~/attr# chattr +i file42
root@laika:~/attr# lsattr
----i------------- ./file42
root@laika:~/attr# rm -rf file42
rm: cannot remove `file42': Operation not permitted
root@laika:~/attr# chattr -i file42
root@laika:~/attr# rm -rf file42
root@laika:~/attr#
```

Chapter 9. File Links

9.1. about inodes

To understand links in a file system, you first have to understand what an **inode** is. When the file system stores a new file on the hard disk, it stores not only the contents (data) of the file, but also some extra properties like the name of the file, the creation date, the permissions, the owner of the file... and more. All this information (except the name of the file and the data) is stored in the inode of the file.

All the inodes are created when you create the file system (with mkfs). Most of them are unused and empty, each inode has a unique number (the inode number). You can see the inode numbers with the **ls -li** command.

```
paul@RHELv4u4:~/test$ touch file1
paul@RHELv4u4:~/test$ touch file2
paul@RHELv4u4:~/test$ touch file3
paul@RHELv4u4:~/test$ ls -li
total 12
817266 -rw-rw-r--  1 paul paul 0 Feb  5 15:38 file1
817267 -rw-rw-r--  1 paul paul 0 Feb  5 15:38 file2
817268 -rw-rw-r--  1 paul paul 0 Feb  5 15:38 file3
paul@RHELv4u4:~/test$
```

Three files created one after the other get three different inodes (the first column). All the information you see with this ls command resides in the inode, except for the filename (which is contained in the directory). Let's put some data in one of the files.

```
paul@RHELv4u4:~/test$ ls -li
total 16
817266 -rw-rw-r--  1 paul paul  0 Feb  5 15:38 file1
817270 -rw-rw-r--  1 paul paul 92 Feb  5 15:42 file2
817268 -rw-rw-r--  1 paul paul  0 Feb  5 15:38 file3
paul@RHELv4u4:~/test$ cat file2
It is winter now and it is very cold.
We do not like the cold, we prefer hot summer nights.
paul@RHELv4u4:~/test$
```

The data that is displayed by the cat commend is not in the inode, but somewhere else on the disk. But the inode contains a pointer to the data.

9.2. about directories

A **directory** is a special kind of file. It contains a table mapping filenames to inodes. Looking at our current directory with **ls -ali** will display the contents of the directory file.

```
paul@RHELv4u4:~/test$ ls -ali
total 32
817262 drwxrwxr-x   2 paul paul 4096 Feb  5 15:42 .
800768 drwx------  16 paul paul 4096 Feb  5 15:42 ..
817266 -rw-rw-r--   1 paul paul    0 Feb  5 15:38 file1
817270 -rw-rw-r--   1 paul paul   92 Feb  5 15:42 file2
817268 -rw-rw-r--   1 paul paul    0 Feb  5 15:38 file3
paul@RHELv4u4:~/test$
```

You can see five names, and the mapping to their five inodes. The dot **.** is a mapping to itself, and the dotdot **..** is a mapping to the parent directory. The three others are mappings to files.

9.3. hard links

When we create a **hard link** to a file with **ln**, then an extra entry is added in the directory. A new file name is mapped to an existing inode.

```
paul@RHELv4u4:~/test$ ln file2 hardlink_to_file2
paul@RHELv4u4:~/test$ ls -li
total 24
817266 -rw-rw-r--  1 paul paul  0 Feb  5 15:38 file1
817270 -rw-rw-r--  2 paul paul 92 Feb  5 15:42 file2
817268 -rw-rw-r--  1 paul paul  0 Feb  5 15:38 file3
817270 -rw-rw-r--  2 paul paul 92 Feb  5 15:42 hardlink_to_file2
paul@RHELv4u4:~/test$
```

Both files have the same inode, so they will always have the same permissions and the same owner. And they will both have the same content. Actually, both files are equal now, meaning you can safely remove the original file, the hardlinked file will remain. The inode contains a counter, counting the number of hard links to itself. When the counter drops to zero, then the inode is emptied.

You can use the **find** command to look for files with a certain inode. The screenshot below proves that / and /boot are different partitions, since every **inode** number is unique to the partition.

```
paul@RHELv4u4:~/test$ find / -inum 2 2> /dev/null
/
/boot
/var/lib/nfs/rpc_pipefs/lockd
/proc/self
paul@RHELv4u4:~/test$
```

9.4. symbolic links

Symbolic links (sometimes called **soft links**) do not link to inodes, but create a name to name mapping. Symbolic links are created with **ln -s**. As you can see below, the **symbolic link** gets an inode of its own.

```
paul@RHELv4u4:~/test$ ln -s file2 symlink_to_file2
paul@RHELv4u4:~/test$ ls -li
total 32
817273 -rw-rw-r--  1 paul paul  13 Feb  5 17:06 file1
817270 -rw-rw-r--  2 paul paul 106 Feb  5 17:04 file2
817268 -rw-rw-r--  1 paul paul   0 Feb  5 15:38 file3
817270 -rw-rw-r--  2 paul paul 106 Feb  5 17:04 hardlink_to_file2
817267 lrwxrwxrwx  1 paul paul   5 Feb  5 16:55 symlink_to_file2 -> file2
paul@RHELv4u4:~/test$
```

Permissions on a symbolic link have no meaning, since the permissions of the target apply. Hard links are limited to their own partition (because they point to an inode), symbolic links can link anywhere (other file systems, even networked).

9.5. removing links

Links can be removed with **rm**.

```
paul@laika:~$ touch data.txt
paul@laika:~$ ln -s data.txt sl_data.txt
paul@laika:~$ ln data.txt hl_data.txt
paul@laika:~$ rm sl_data.txt
paul@laika:~$ rm hl_data.txt
```

9.6. Practice: Links

1. Create two files named winter.txt and summer.txt, put some text in them.

2. Create a hard link to winter.txt named hlwinter.txt.

3. Display the inode numbers of these three files, the hard links should have the same inode.

4. Use the find command to list the two hardlinked files

5. Everything about a file is in the inode, except two things : name them!

6. Create a symbolic link to summer.txt called slsummer.txt.

7. Find all files with inode number 2. What does this information tell you ?

8. Look at the directories /etc/init.d/ /etc/rc.d/ /etc/rc3.d/ ... do you see the links ?

9. Look in /lib with ls -l...

9.7. Solutions: Links

1. Create two files named winter.txt and summer.txt, put some text in them.

```
echo cold > winter.txt ; echo hot > summer.txt
```

2. Create a hard link to winter.txt named hlwinter.txt.

```
ln winter.txt hlwinter.txt
```

3. Display the inode numbers of these three files, the hard links should have the same inode.

```
ls -li winter.txt summer.txt hlwinter.txt
```

4. Use the find command to list the two hardlinked files

```
find . -inum xyz
```

5. Everything about a file is in the inode, except two things : name them!

The name of the file is in a directory, and the contents is somewhere on the disk.

6. Create a symbolic link to summer.txt called slsummer.txt.

```
ln -s summer.txt slsummer.txt
```

7. Find all files with inode number 2. What does this information tell you ?

It tells you there is more than one inode table (one for every formatted partition + virtual file systems)

8. Look at the directories /etc/init.d/ /etc/rc.d/ /etc/rc3.d/ ... do you see the links ?

```
ls -l /etc/init.d
```

```
ls -l /etc/rc.d
```

```
ls -l /etc/rc3.d
```

9. Look in /lib with ls -l...

```
ls -l /lib
```

Chapter 10. Introduction to Scripting

10.1. About scripting

Bash has support for programming constructs that can be saved as scripts. These scripts in turn then become more bash commands. In fact, a lot of linux commands are scripts. This means that system administrators also need a basic knowledge of scripting to understand how their servers and their applications are started, updated, upgraded, patched, maintained, configured and removed.

10.2. Hello World

Just like in every programming course, we start with a simple Hello World script. The following script will output Hello World.

```
#!/bin/bash
# Hello World Script
echo Hello World
```

After creating this simple script in vi, you'll have to **chmod +x** the script to make it executable. And unless you add the scripts directory to your path, you'll have to type the path to the script for the shell to be able to find it.

```
[paul@RHEL4a ~]$ chmod +x hello_world
[paul@RHEL4a ~]$ ./hello_world
Hello World
[paul@RHEL4a ~]$
```

10.3. Variables

```
#!/bin/bash
var1=4
echo var1 = $var1
```

Scripts can contain variables, but since scripts are run in their own shell, the variables do not survive the end of the script.

```
[paul@RHEL4a ~]$ echo $var1

[paul@RHEL4a ~]$ ./vars
var1 = 4
[paul@RHEL4a ~]$ echo $var1

[paul@RHEL4a ~]$
```

Luckily you can force a script to run in the same shell, this is called sourcing a script.

```
[paul@RHEL4a ~]$ source ./vars
var1 = 4
[paul@RHEL4a ~]$ echo $var1
4
[paul@RHEL4a ~]$
```

The above is identical to the below.

```
[paul@RHEL4a ~]$ . ./vars
var1 = 4
[paul@RHEL4a ~]$ echo $var1
4
[paul@RHEL4a ~]$
```

10.4. Shell

You can never be sure which shell a user is running. A script that works flawlessly in bash, might not work in ksh or csh or dash. To instruct a shell to run your script in a certain shell, you can start your script with a shebang followed by the shell it is supposed to run in. This script will run in a bash shell.

```
#!/bin/bash
echo -n hello
echo A bash subshell `echo -n hello`
```

This script will run in a Korn shell (unless /bin/ksh is a link to /bin/bash). The **/etc/shells** file contains a list of shells on your system.

```
#!/bin/ksh
echo -n hello
echo a Korn subshell `echo -n hello`
```

Some user may perform setuid based script root spoofing. This is a rare but possible attack. To improve script security, and to avoid interpreter spoofing you need to add -- to #!/bin/bash, which disables further option processing, so bash will not accept any options.

```
#!/bin/bash -
```

or

```
#!/bin/bash --
```

Any arguments after the -- are treated as filenames and arguments. An argument of - is equivalent to --.

10.5. for loop

The example below shows the syntax of a classical **for loop** in bash.

```
for i in 1 2 4
do
   echo $i
done
```

An example of a for loop combined with an embedded shell to generate the list.

```
for file in `ls *.txt`
do
   cp $file $file.bak
   echo Backup of $file put in $file.bak
done
```

10.6. while loop

Below a simple example of a **while loop**.

```
let i=100;
while [ $i -ge 0 ] ;
do
   echo Counting down, from 100 to 0, now at $i;
   let i--;
done
```

10.7. until loop

Below a simple example of an **until loop**.

```
let i=100;
until [ $i -le 0 ] ;
do
   echo Counting down, from 100 to 1, now at $i;
   let i--;
done
```

10.8. parameters

A bash shell script can have parameters. The numbering you see in the script below continues if you have more parameters. You also have special parameters for the number of parameters, a string of all of them, and also the process id and the last error code. The man page of bash has a full list.

```
#!/bin/bash
echo The first argument is $1
echo The second argument is $2
echo The third argument is $3

echo \$ $$  PID of the script
echo \# $#  count arguments
echo \? $?  last error code
echo \* $*  all the arguments
```

Below is the output of the script above in action.

```
[paul@RHEL4a scripts]$ ./pars one two three
The first argument is one
The second argument is two
The third argument is three
$ 5610 PID of the script
# 3 count arguments
? 0 last error code
* one two three all the arguments
[paul@RHEL4a scripts]$ ./pars a b c
The first argument is a
The second argument is b
The third argument is c
$ 5611 PID of the script
# 3 count arguments
? 0 last error code
* a b c all the arguments
[paul@RHEL4a scripts]$ ./pars 1 2
The first argument is 1
The second argument is 2
The third argument is
$ 5612 PID of the script
# 2 count arguments
? 0 last error code
* 1 2 all the arguments
[paul@RHEL4a scripts]$
```

10.9. test [ ]

The **test** command can test whether something is true or false. Let's start by testing whether 10 is greater than 55.

```
[paul@RHEL4b ~]$ test 10 -gt 55 ; echo $?
1
[paul@RHEL4b ~]$
```

The test command returns 1 if the test fails. And as you see in the next screenshot, test returns 0 when a test succeeds.

```
[paul@RHEL4b ~]$ test 56 -gt 55 ; echo $?
0
[paul@RHEL4b ~]$
```

If you prefer true and false, then write the test like this.

```
[paul@RHEL4b ~]$ test 56 -gt 55 && echo true || echo false
true
[paul@RHEL4b ~]$ test 6 -gt 55 && echo true || echo false
false
```

The test command can also be written as square brackets, the screenshot below is identical to the one above.

```
[paul@RHEL4b ~]$ [ 56 -gt 55 ] && echo true || echo false
true
[paul@RHEL4b ~]$ [ 6 -gt 55 ] && echo true || echo false
false
```

Below are some example tests. Take a look at **man test** to see more options for tests.

```
[ -d foo ]              Does the directory foo exist ?
[ '/etc' = $PWD ]       Is the string /etc equal to the variable $PWD ?
[ $1 != 'secret' ]      Is the first parameter different from secret ?
[ 55 -lt $bar ]         Is 55 less than the value of $bar ?
[ $foo -ge 1000 ]       Is the value of $foo greater or equal to 1000 ?
[ "abc" < $bar ]  Does abc sort before the value of $bar ?
[ -f foo ]              Is foo a regular file ?
[ -r bar ]              Is bar a readable file ?
[ foo -nt bar ]         Is file foo newer than file bar ?
[ -o nounset ]          Is the shell option nounset set ?
```

Tests can be combined with logical AND and OR.

```
[paul@RHEL4b ~]$ [ 66 -gt 55 -a 66 -lt 500 ] && echo true || echo false
true
[paul@RHEL4b ~]$ [ 66 -gt 55 -a 660 -lt 500 ] && echo true || echo false
false
[paul@RHEL4b ~]$ [ 66 -gt 55 -o 660 -lt 500 ] && echo true || echo false
true
```

10.10. if if, then then, or else

The **if then else** construction is about choice. If a certain condition is met, then execute something, else execute something else. The example below tests whether a file exists, if the file exists then a proper message is echoed.

```
#!/bin/bash

if [ -f isit.txt ]
then echo isit.txt exists!
else echo isit.txt not found!
fi
```

If we name the above script 'choice', then it executes like this.

```
[paul@RHEL4a scripts]$ ./choice
isit.txt not found!
[paul@RHEL4a scripts]$ touch isit.txt
[paul@RHEL4a scripts]$ ./choice
isit.txt exists!
[paul@RHEL4a scripts]$
```

10.11. let

The **let** command allows for evalutation of arithmetic expressions.

```
[paul@RHEL4b ~]$ let x="3 + 4" ; echo $x
7
[paul@RHEL4b ~]$ let x="10 + 100/10" ; echo $x
20
[paul@RHEL4b ~]$ let x="10-2+100/10" ; echo $x
18
[paul@RHEL4b ~]$ let x="10*2+100/10" ; echo $x
30
```

The let command can also convert between different bases.

```
[paul@RHEL4b ~]$ let x="0xFF" ; echo $x
255
[paul@RHEL4b ~]$ let x="0xC0" ; echo $x
192
[paul@RHEL4b ~]$ let x="0xA8" ; echo $x
168
[paul@RHEL4b ~]$ let x="8#70" ; echo $x
56
[paul@RHEL4b ~]$ let x="8#77" ; echo $x
63
[paul@RHEL4b ~]$ let x="16#c0" ; echo $x
192
```

10.12. runtime input

You can ask the user for input with the **read** command in a script.

```
#!/bin/bash
echo -n Enter a number:
read number
```

10.13. sourcing a config file

```
[paul@RHEL4a scripts]$ cat myApp.conf
# The config file of myApp

# Enter the path here
myAppPath=/var/myApp

# Enter the number of quines here
quines=5

[paul@RHEL4a scripts]$ cat myApp.bash
#!/bin/bash
#
# Welcome to the myApp application
#

. ./myApp.conf

echo There are $quines quines

[paul@RHEL4a scripts]$ ./myApp.bash
There are 5 quines
[paul@RHEL4a scripts]$
```

10.14. case

You can sometimes simplify nested if statements with a case construct.

```
[paul@RHEL4b ~]$ ./help
What animal did you see ? lion
You better start running fast!
[paul@RHEL4b ~]$ ./help
What animal did you see ? dog
Don't worry, give it a cookie.
[paul@RHEL4b ~]$ cat help
#!/bin/bash
#
# Wild Animals Helpdesk Advice
#
echo -n "What animal did you see ? "
read animal
case $animal in
        "lion" | "tiger")
                echo "You better start running fast!"
        ;;
        "cat")
                echo "Let that mouse go..."
        ;;
        "dog")
                echo "Don't worry, give it a cookie."
        ;;
        "chicken" | "goose" | "duck" )
                echo "Eggs for breakfast!"
        ;;
        "liger")
                echo "Approach and say 'Ah you big fluffy kitty...'."
        ;;
        "babelfish")
                echo "Did it fall out your ear ?"
        ;;
        *)
```

```
                echo "You discovered an unknown animal, name it!"
        ;;
esac
[paul@RHEL4b ~]$
```

10.15. shopt

You can toggle the values of variables controlling optional shell behavior with the **shopt** built-in shell command. The example below first verifies whether the cdspell option is set, it is not. The next shopt command sets the value, and the third shopt command verifies that the option really is set. You can now use minor spelling mistakes in the cd command. The man page of bash has a complete list of options.

```
paul@laika:~$ shopt -q cdspell ; echo $?
1
paul@laika:~$ shopt -s cdspell
paul@laika:~$ shopt -q cdspell ; echo $?
0
paul@laika:~$ cd /Etc
/etc
paul@laika:/etc$
```

10.16. Practice : scripts

0. Give each script a different name, keep them for later!

1. Write a script that receives four parameters, and outputs them in reverse order.

2. Write a script that receives two parameters (two filenames) and outputs whether those files exist.

3. Write a script that counts the number of files ending in .txt in the current directory.

4. Write a script that asks for two numbers, and outputs the sum and product (as shown here).

```
Enter a number: 5
Enter another number: 2

Sum:       5 + 2 = 7
Product:   5 x 2 = 10
```

5. Improve the previous script to test that the numbers are between 1 and 100, exit with an error if necessary.

6. Improve the previous script to congratulate the user if the sum equals the product.

7. Improve the script from question 2. to complain if it does not receive exactly two parameters.

8. Write a script that counts from 3 to 7 and then from 7 to 3, and all this three times, once with a for loop, once with a while loop and once with a until loop. Show the teacher that it works!

9. Write a script that asks for a filename. Verify existance of the file, then verify that you own the file, and whether it is writable. If not, then make it writable.

10. Make a configuration file for the previous script. Put a logging switch in the config file, logging means writing detailed output of everything the script does to a log file in /tmp.

11. Make the case statement in "Wild Animals Helpdesk Advice" case insensitive. Use shopt (with the correct toggled option) for this, but reset the value back to it's original after the end of the case statement. (A solution is available in appendix 1, but try to find it yourself.)

12. If time permits (or if you are waiting for other students to finish this practice), take a look at linux system scripts in /etc/init.d and /etc/rc.d and try to understand them. Where does execution of a script start in /etc/init.d/samba ? There are also some hidden scripts in ~, we will discuss them later.

10.17. Solutions

1. Write a script that receives four parameters, and outputs them in reverse order.

```
echo $4 $3 $2 $1
```

2. Write a script that receives two parameters (two filenames) and outputs whether those files exist.

```
#!/bin/bash

if [ -f $1 ]
then echo $1 exists!
else echo $1 not found!
fi

if [ -f $2 ]
then echo $2 exists!
else echo $2 not found!
fi
```

3. Write a script that counts the number of files ending in .txt in the current directory.

```
#!/bin/bash

let i=0
for file in *.txt
do
let i++
done
```

```
echo "There are $i files ending in .txt"
```

4. Write a script that asks for two numbers, and outputs the sum and product (as shown here).

```
Enter a number: 5
Enter another number: 2

Sum:       5 + 2 = 7
Product:   5 x 2 = 10
```

```
#!/bin/bash

echo -n "Enter a number : "
read n1

echo -n "Enter another number : "
read n2

let sum="$n1+$n2"
let pro="$n1*$n2"

echo -e "Sum\t: $n1 + $n2 = $sum"
echo -e "Product\t: $n1 * $n2 = $pro"
```

5. Improve the previous script to test that the numbers are between 1 and 100, exit with an error if necessary.

```
echo -n "Enter a number between 1 and 100 : "
read n1

if [ $n1 -lt 1 -o $n1 -gt 100 ]
then
       echo Wrong number...
       exit 1
fi
```

6. Improve the previous script to congratulate the user if the sum equals the product.

```
if [ $sum -eq $pro ]
then echo Congratulations $sum == $pro
fi
```

7. Improve the script from question 2. to complain if it does not receive exactly two parameters.

```
if [ $# -ne 2 ]
then
 echo Must get two parameters...
 exit 1
fi
```

8. Write a script that counts from 3 to 7 and then from 7 to 3, and all this three times, once with a for loop, once with a while loop and once with a until loop. Show the teacher that it works!

9. Write a script that asks for a filename. Verify existance of the file, then verify that you own the file, and whether it is writable. If not, then make it writable.

10. Make a configuration file for the previous script. Put a logging switch in the config file, logging means writing detailed output of everything the script does to a log file in /tmp.

11. A script with a case insensitive case statement, using the shopt nocasematch option. The nocasematch option is reset to the value it had before the scripts started.

```
#!/bin/bash
#
# Wild Animals Case Insensitive Helpdesk Advice
#

if shopt -q nocasematch; then
  nocase=yes;
else
  nocase=no;
  shopt -s nocasematch;
fi

echo -n "What animal did you see ? "
read animal

case $animal in
  "lion" | "tiger")
    echo "You better start running fast!"
  ;;
  "cat")
    echo "Let that mouse go..."
  ;;
  "dog")
    echo "Don't worry, give it a cookie."
  ;;
  "chicken" | "goose" | "duck" )
    echo "Eggs for breakfast!"
  ;;
  "liger")
    echo "Approach and say 'Ah you big fluffy kitty.'"
  ;;
  "babelfish")
    echo "Did it fall out your ear ?"
  ;;
  *)
    echo "You discovered an unknown animal, name it!"
  ;;
esac

if [ nocase = yes ] ; then
        shopt -s nocasematch;
else
        shopt -u nocasematch;
fi
```

12. If time permits (or if you are waiting for other students to finish this practice), take a look at linux system scripts in /etc/init.d and /etc/rc.d and try to understand them. Where does execution of a script start in /etc/init.d/samba ? There are also some hidden scripts in ~, we will discuss them later.

Chapter 11. Process Management

11.1. About processes

A **process** is compiled source code that is currently running on the system. All processes have a **process ID** or **PID**, and a parent process (with a **PPID**). The **child** process is often started by the **parent** process. The **init** process always has process ID 1, and does not have a parent. But init serves as a **foster parent** for **orphaned** processes. When a process stops running, the process dies, when you want a process to die, you **kill** it. Processes that start at system startup and keep running forever are called **daemon** processes. Daemons never die. When a process is killed, but it still shows up on the system, then the process is referred to as **zombie**. You cannot kill zombies, because they are already dead.

11.2. Process ID

Some shell environment variables contain information about processes. The **$$** variable will hold your current process ID (PID), and **$PPID** contains the parent PID. Actually $$ is a shell parameter and not a variable, you cannot assign a value to $$.

```
[paul@RHEL4b ~]$ echo $$ $PPID
4224 4223
[paul@RHEL4b ~]$ bash
[paul@RHEL4b ~]$ echo $$ $PPID
4812 4224
[paul@RHEL4b ~]$ bash
[paul@RHEL4b ~]$ echo $$ $PPID
4830 4812
[paul@RHEL4b ~]$ exit
exit
[paul@RHEL4b ~]$ echo $$ $PPID
4812 4224
[paul@RHEL4b ~]$ exit
exit
[paul@RHEL4b ~]$ echo $$ $PPID
4224 4223
[paul@RHEL4b ~]$
```

11.3. fork

A process starts another process in two fases. First the process creates a **fork** of itself, an identical copy. Then the forked process executes an **exec** to replace the forked process with the target child process.

```
[paul@RHEL4b ~]$ echo $$
4224
[paul@RHEL4b ~]$ bash
[paul@RHEL4b ~]$ echo $$ $PPID
```

```
5310 4224
[paul@RHEL4b ~]$
```

11.4. exec

With the **exec** command, you can execute a process without forking a new process. In the following screenshot i start a Korn shell (ksh) and replace it with a bash shell using the exec command. The PID of the bash shell is the same as the PID of the Korn shell. Exiting the child bash shell will get me back to the parent bash, not to the Korn (which does not exist anymore).

```
[paul@RHEL4b ~]$ echo $$
4224
[paul@RHEL4b ~]$ ksh
$ echo $$ $PPID
5343 4224
$ exec bash
[paul@RHEL4b ~]$ echo $$ $PPID
5343 4224
[paul@RHEL4b ~]$ exit
exit
[paul@RHEL4b ~]$ echo $$
4224
```

11.5. ps

One of the most common tools on Unix to look at processes is **ps**. The following screenshot shows the parent child relationship between three bash processes.

```
[paul@RHEL4b ~]$ echo $$ $PPID
4224 4223
[paul@RHEL4b ~]$ bash
[paul@RHEL4b ~]$ echo $$ $PPID
4866 4224
[paul@RHEL4b ~]$ bash
[paul@RHEL4b ~]$ echo $$ $PPID
4884 4866
[paul@RHEL4b ~]$ ps fx
  PID TTY      STAT   TIME COMMAND
 4223 ?        S      0:01 sshd: paul@pts/0
 4224 pts/0    Ss     0:00  \_ -bash
 4866 pts/0    S      0:00      \_ bash
 4884 pts/0    S      0:00          \_ bash
 4902 pts/0    R+     0:00              \_ ps fx
[paul@RHEL4b ~]$ exit
exit
[paul@RHEL4b ~]$ ps fx
  PID TTY      STAT   TIME COMMAND
 4223 ?        S      0:01 sshd: paul@pts/0
 4224 pts/0    Ss     0:00  \_ -bash
 4866 pts/0    S      0:00      \_ bash
 4903 pts/0    R+     0:00          \_ ps fx
[paul@RHEL4b ~]$ exit
exit
```

```
[paul@RHEL4b ~]$ ps fx
  PID TTY      STAT   TIME COMMAND
 4223 ?        S      0:01 sshd: paul@pts/0
 4224 pts/0    Ss     0:00  \_ -bash
 4904 pts/0    R+     0:00      \_ ps fx
[paul@RHEL4b ~]$
```

On Linux, **ps fax** is often used. On Solaris **ps -ef** is common. Here is a partial output from ps fax.

```
[paul@RHEL4a ~]$ ps fax
PID TTY      STAT   TIME COMMAND
1 ?        S      0:00 init [5]

...

3713 ?        Ss     0:00 /usr/sbin/sshd
5042 ?        Ss     0:00  \_ sshd: paul [priv]
5044 ?        S      0:00      \_ sshd: paul@pts/1
5045 pts/1    Ss     0:00          \_ -bash
5077 pts/1    R+     0:00              \_ ps fax
```

11.6. top

Another popular tool on Linux is **top**. The **top** tool can order processes according to CPU usage or other properties. You can also kill processes from within top. In case of trouble, top is often the first tool to fire up, since it also provides you memory and swap space information.

11.7. priority and nice values

All processes have a certain **priority** and a **nice** value. Higher priority processes will get more CPU time than low priority processes. You can influence this with the **nice** and **renice** commands.

The **top** screenshot below shows four processes, all of then using approximately 25 percent of the CPU. PID 5087 and 5088 are catting the letter x to each other, PID 5090 and 5091 do the same with the letter z.

```
PID  USER PR NI VIRT RES SHR S %CPU %MEM TIME+   COMMAND
5088 paul 25  0 4128 404 348 R 25.6  0.2 0:13.99 cat
5091 paul 25  0 3628 400 348 R 25.6  0.2 0:07.99 cat
5090 paul 15  0 4484 404 348 S 24.6  0.2 0:07.78 cat
5087 paul 15  0 3932 400 348 S 24.3  0.2 0:14.16 cat
```

Since the processes are already running, we need to use the **renice** command to change their nice value. The **nice** command can only be used when starting a process. The screenshot below shows how to make two running processes nice.

```
[paul@RHEL4a ~]$ renice +5 5090
5090: old priority 0, new priority 5
[paul@RHEL4a ~]$ renice +5 5091
5091: old priority 0, new priority 5
```

Two processes (5090 and 5091) are playing nice now, they allow other processes to use more CPU time.

```
PID  USER PR NI VIRT RES SHR S %CPU %MEM  TIME+   COMMAND
5087 paul 15  0 3932 400 348 S 37.3  0.2  1:19.97 cat
5088 paul 25  0 4128 404 348 R 36.6  0.2  1:19.20 cat
5090 paul 21  5 4484 404 348 S 13.7  0.2  1:10.64 cat
5091 paul 29  5 3628 400 348 R 12.7  0.2  1:10.64 cat
```

Be careful when playing with negative nice values (the range is from -20 to 19), the responsiveness of your system can be affected. Luckily only root can issue negative nice values, in other words, you can only lower the priority of your running processes.

11.8. signals (kill)

Running processes can receive signals from each other, or from the users. You can have a list of signals by typing **kill -l**, that is a letter l, not the number 1.

```
[paul@RHEL4a ~]$ kill -l
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1
36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4  39) SIGRTMIN+5
40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8  43) SIGRTMIN+9
44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1
64) SIGRTMAX
[paul@RHEL4a ~]$
```

It is common on Linux to use the first signal SIGHUP (or HUP or 1) to tell a process that it should re-read its configuration file. Thus, the **kill -1 1** command forces the init process to re-read its configuration file. It is up to the developer of the process to decide whether the process can do this running, or whether it needs to stop and start. The **killall** command will also default to sending a signal 15 to the processes.

The SIGTERM (15) is used to ask a process to stop running, normally the process should die. If it refuses to die, then you can issue the **kill -9** command (aka the **sure kill**). The SIGKILL (9) signal is the only one that a developer cannot intercept. The signal goes directly to the kernel, which will stop the running process (without giving

it a chance to save data). When using the kill command without specifying a signal, it defaults to SIGTERM (15).

```
[paul@RHEL4a ~]$ ps fax | grep cat
5087 pts/1    S     10:04              \_ cat - pipe1
5088 pts/1    R     10:06              \_ cat
5090 pts/1    SN     4:26              \_ cat - pipe3
5091 pts/1    RN     4:28              \_ cat
5220 pts/1    S+     0:00              \_ grep cat
[paul@RHEL4a ~]$ kill 5087
[1]   Terminated              echo -n x | cat - pipe1 >pipe2
[paul@RHEL4a ~]$
```

11.9. pgrep

Similar to the **ps -C**, you can also use **pgrep** to search for a process by its command name.

```
[paul@RHEL5 ~]$ sleep 1000 &
[1] 32558
[paul@RHEL5 ~]$ pgrep sleep
32558
[paul@RHEL5 ~]$ ps -C sleep
  PID TTY          TIME CMD
32558 pts/3    00:00:00 sleep
[paul@RHEL5 ~]$
```

You can also list the command name of the process with pgrep.

```
paul@laika:~$ pgrep -l sleep
9661 sleep
paul@laika:~$
```

11.10. pkill

You can use the **pkill** command to kill a process by its command name.

```
[paul@RHEL5 ~]$ sleep 1000 &
[1] 30203
[paul@RHEL5 ~]$ pkill sleep
[1]+  Terminated              sleep 1000
[paul@RHEL5 ~]$
```

11.11. jobs

Some processes can be frozen with the **Ctrl-Z** key combination. This sends a SIGSTOP to the process. When doing this in vi, the vi goes to the background, and can be seen with the **jobs** command. Processes started with an ampersand (&) at the end of the command line can also be seen with **jobs**.

```
[paul@RHEL4a ~]$ vi procdemo.txt

[5]+  Stopped                 vim procdemo.txt
[paul@RHEL4a ~]$ jobs
[5]+  Stopped                 vim procdemo.txt
[paul@RHEL4a ~]$ find / > allfiles.txt 2> /dev/null &
[6] 5230
[paul@RHEL4a ~]$ jobs
[5]+  Stopped                 vim procdemo.txt
[6]-  Running                 find / >allfiles.txt 2>/dev/null &
[paul@RHEL4a ~]$
```

An interesting option is **jobs -p** to see the PID of background jobs.

```
[paul@RHEL4b ~]$ sleep 500 &
[1] 4902
[paul@RHEL4b ~]$ sleep 400 &
[2] 4903
[paul@RHEL4b ~]$ jobs -p
4902
4903
[paul@RHEL4b ~]$ ps `jobs -p`
  PID TTY      STAT   TIME COMMAND
 4902 pts/0    S      0:00 sleep 500
 4903 pts/0    S      0:00 sleep 400
[paul@RHEL4b ~]$
```

11.12. fg

Running the **fg** command will bring a background job to the foreground. The number of the background job to bring forward is the parameter of fg.

```
[paul@RHEL5 ~]$ jobs
[1]   Running                 sleep 1000 &
[2]-  Running                 sleep 1000 &
[3]+  Running                 sleep 2000 &
[paul@RHEL5 ~]$ fg 3
sleep 2000
```

11.13. bg

Jobs that are stopped in background can be started in background with **bg**. Below an example of the sleep command (stopped with Ctrl-Z) that is reactivated in background with bg.

```
[paul@RHEL5 ~]$ jobs
[paul@RHEL5 ~]$ sleep 5000 &
[1] 6702
[paul@RHEL5 ~]$ sleep 3000

[2]+  Stopped                 sleep 3000
```

```
[paul@RHEL5 ~]$ jobs
[1]-  Running                 sleep 5000 &
[2]+  Stopped                 sleep 3000
[paul@RHEL5 ~]$ bg 2
[2]+ sleep 3000 &
[paul@RHEL5 ~]$ jobs
[1]-  Running                 sleep 5000 &
[2]+  Running                 sleep 3000 &
[paul@RHEL5 ~]$
```

11.14. Zombiecreator ;-)

This is a procedure to create zombies on Linux.

First create a script that contains one line: sleep 5000. Make it executable (chmod +x). Then start this script and use Ctrl-Z to background the script. Use ps -C sleep to find the PID of the sleep process. Then do a kill -9 of this PID. ps -C sleep now shows a defunct sleep. ps -fax and top will show a zombie sleep process.

11.15. Practice

1. Explain in detail where the numbers come from in the next screenshot. When are the variables replaced by their value ? By which shell ?

```
[paul@RHEL4b ~]$ echo $$ $PPID
4224 4223
[paul@RHEL4b ~]$ bash -c "echo $$ $PPID"
4224 4223
[paul@RHEL4b ~]$ bash -c 'echo $$ $PPID'
5059 4224
[paul@RHEL4b ~]$ bash -c `echo $$ $PPID`
5143: 4224: command not found
```

2. Write a script that echoes its process ID and then sleeps for an hour. Find your script with ps.

3. Read the man page of ps and find your script by command name with ps.

4. Kill your script with the kill command.

5. Run your script again, now use top to display only your script and the init process.

6. Use top to kill your script.

7. Use top, organise all processes by memory usage.

8. Write a script with a 'while true' loop that does some calculation. Copy this script.

9. Start the while script. Start the copy of it in a nice way. Do you see the difference with top ? with ps ?

10. Kill all your running scripts.

11. Start editing the while script, put it in background. Same for the copy script. List your background jobs.

12. Start the sleep script in background. List the background jobs. Activate the copy script to foreground.

11.16. Solutions to the Practice

1. The current bash shell will replace the $$ and $PPID while scanning the line, and before executing the echo command.

```
[paul@RHEL4b ~]$ echo $$ $PPID
4224 4223
```

The variables are now double quoted, but the current bash shell will replace $$ and $PPID while scanning the line, and before executing the bach -c command.

```
[paul@RHEL4b ~]$ bash -c "echo $$ $PPID"
4224 4223
```

The variables are now single quoted. The current bash shell will not replace the $$ and the $PPID. The bash -c command will be executed before the variables replaced with their value. This latter bash is the one replacing the $$ and $PPID with their value.

```
[paul@RHEL4b ~]$ bash -c 'echo $$ $PPID'
5059 4224
```

With backticks the shell executes the command between backticks. The result is the two process id's, which are not commands.

```
[paul@RHEL4b ~]$ bash -c `echo $$ $PPID`
5059: 4224: command not found
```

2. The script can look like this.

```
#!/bin/bash

echo My Process ID = $$
sleep 3600
```

3. ps -C sleep

4. kill (followed by the PID)

5. top p 1,6705 (replace 6705 with your PID)

6. when inside top, press k

7. use the greater than and smaller than keys from within top

8. an example of an (almost) endless loop:

```
let i=1;
while [ $i -gt 0 ] ;
do
   let i++;
done
```

9. it will show up as 'bash' in top

Chapter 12. More about Bash

12.1. bash shell environment

It is nice to have all these preset and custom aliases and variables, but where do they all come from ? Bash has a number of startup files that are checked (and executed) whenever bash is invoked. Bash first reads and executes **/etc/profile**. Then bash searches for **.bash_profile**, **.bash_login** and **.profile** in the home directory. Bash will execute the first of these three that it finds. Typically these files will expand your $PATH environment variable.

```
[paul@RHELv4u3 ~]$ cat .bash_profile | grep PATH
PATH=$PATH:$HOME/bin
export PATH
[paul@RHELv4u3 ~]$
```

If this is an interactive shell, then bash will also execute **.bashrc**. In the case of Red Hat, the .bashrc file will source **/etc/bashrc**.

```
[paul@RHELv4u3 ~]$ cat .bashrc
# .bashrc

# User specific aliases and functions

# Source global definitions
if [ -f /etc/bashrc ]; then
. /etc/bashrc
fi
[paul@RHELv4u3 ~]$
```

When you exit the shell, then **~/.bash_logout** is executed.

A similar system exists for the Korn shell with .kshrc and other files. Actually a similar system exists for almost all shells.

12.2. path

The **$PATH** variable is very important, it determines where the shell is looking for commands to execute (unless the command is built-in). The shell will not look in the current directory for commands to execute! (Looking for executables in the current directory provided an easy way to crack DOS computers). If you want the shell to look in the current directory, then add a . to your path.

```
[[paul@RHEL4b ~]$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:
[paul@RHEL4b ~]$ PATH=$PATH:.
[paul@RHEL4b ~]$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:.
```

```
[paul@RHEL4b ~]$
```

Your path might be different when using su instead of **su -** because the latter will take on the environment of the target user. The root user will have some sbin directories added to the PATH variable.

```
[paul@RHEL3 ~]$ su
Password:
[root@RHEL3 paul]# echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
[root@RHEL3 paul]# exit
[paul@RHEL3 ~]$ su -
Password:
[root@RHEL3 ~]# echo $PATH
/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:
[root@RHEL3 ~]#
```

12.3. Shell I/O redirection

The shell (and almost every other Linux command) takes input from **stdin** (stream **0**) and sends output to **stdout** (stream **1**) and error messages to **stderr** (stream **2**) . Stdin is usually the keyboard, stdout and stderr are the screen. The shell allows you to redirect these streams.

12.3.1. output redirection

Stdout can be redirected with a **greater than** sign. While scanning the line, the shell will see the > sign and will clear the file.

```
[paul@RHELv4u3 ~]$ echo It is cold today!
It is cold today!
[paul@RHELv4u3 ~]$ echo It is cold today! > winter.txt
[paul@RHELv4u3 ~]$ cat winter.txt
It is cold today!
[paul@RHELv4u3 ~]$
```

Let me repeat myself here: While scanning the line, the shell will see the > sign and will clear the file! This means that even when the command fails, the file will be cleared!

```
[paul@RHELv4u3 ~]$ cat winter.txt
It is cold today!
[paul@RHELv4u3 ~]$ zcho It is cold today! > winter.txt
-bash: zcho: command not found
[paul@RHELv4u3 ~]$ cat winter.txt
[paul@RHELv4u3 ~]$
```

Note that the > notation is in fact the abbreviation of **1>** (**stdout** being referred to as stream **1**.

12.3.2. noclobber

This can be prevented by setting the **noclobber** option.

```
[paul@RHELv4u3 ~]$ cat winter.txt
It is cold today!
[paul@RHELv4u3 ~]$ set -o noclobber
[paul@RHELv4u3 ~]$ echo It is cold today! > winter.txt
-bash: winter.txt: cannot overwrite existing file
[paul@RHELv4u3 ~]$ set +o noclobber
[paul@RHELv4u3 ~]$
```

The noclobber can be overruled with >|.

```
[paul@RHELv4u3 ~]$ set -o noclobber
[paul@RHELv4u3 ~]$ echo It is cold today! > winter.txt
-bash: winter.txt: cannot overwrite existing file
[paul@RHELv4u3 ~]$ echo It is very cold today! >| winter.txt
[paul@RHELv4u3 ~]$ cat winter.txt
It is very cold today!
[paul@RHELv4u3 ~]$
```

12.3.3. append

You can always use >> to append output to a file.

```
[paul@RHELv4u3 ~]$ echo It is cold today! > winter.txt
[paul@RHELv4u3 ~]$ cat winter.txt
It is cold today!
[paul@RHELv4u3 ~]$ echo Where is the summer ? >> winter.txt
[paul@RHELv4u3 ~]$ cat winter.txt
It is cold today!
Where is the summer ?
[paul@RHELv4u3 ~]$
```

12.3.4. error redirection

Redirecting stderr is done with **2>**. This can be very useful to prevent error messages from cluttering your screen. The screenshot below shows redirection of stdout to a file, and stderr to /dev/null. Writing **1>** is the same as >.

```
[paul@RHELv4u3 ~]$ find / > allfiles.txt 2> /dev/null
[paul@RHELv4u3 ~]$
```

To redirect both stdout and stderr to the same file, use **2>&1**.

```
[paul@RHELv4u3 ~]$ find / > allfiles_and_errors.txt 2>&1
[paul@RHELv4u3 ~]$
```

12.3.5. input redirection

Redirecting stdin is done with < (short for 0<).

```
[paul@RHEL4b ~]$ cat < text.txt
one
two
[paul@RHEL4b ~]$ tr 'onetw' 'ONEZZ' < text.txt
ONE
ZZO
[paul@RHEL4b ~]$
```

12.3.6. here document

The here document (sometimes called here-is-document) is a way to append input until a certain sequence (usually EOF) is encountered. The **EOF** marker can be typed literally or can be called with Ctrl-D.

```
[paul@RHEL4b ~]$ cat <<EOF > text.txt
> one
> two
> EOF
[paul@RHEL4b ~]$ cat text.txt
one
two
[paul@RHEL4b ~]$ cat <<brol > text.txt
> brel
> brol
[paul@RHEL4b ~]$ cat text.txt
brel
[paul@RHEL4b ~]$
```

12.4. Confusing I/O redirection

The shell will scan the whole line before applying redirection. The following command line is very readable and is correct.

```
cat winter.txt > snow.txt 2> errors.txt
```

But this one is also correct, but less readable.

```
2> errors.txt cat winter.txt > snow.txt
```

Even this will be understood perfectly by the shell.

```
< winter.txt > snow.txt 2> errors.txt cat
```

So what is the quickest way to clear a file ?

```
>foo
```

12.5. swapping stdout and stderr

When filtering an output stream, e.g. through a pipe (|) you only can filter **stdout**. Say you want to filter out some unimportant error, out of the **stderr** stream. This cannot be done directly, and you need to 'swap' **stdout** and **stderr**. This can be done by using a 4th stream referred to with number 3:

```
3>&1 1>&2 2>&3
```

This Tower Of Hanoi like construction uses a temporary stream 3, to be able to swap stdout (1) and stderr (2). The following is an example of how to filter out all lines in the stderr stream, containing $uninterestingerror.

```
/usr/bin/$somecommand 3>&1 1>&2 2>&3 | grep -v $uninterestingerror ) 3>&1 1>&2 2>&3
```

12.6. Practice: more bash

1. Take a backup copy of /etc/bashrc, /etc/profile, ~/.profile, ~/.bashrc and ~/.bash_profile (put them in ~/profilebackups).

2. Set and export a variable named profwinner in all these scripts, the value is the name of the script (profwinner=etc_bashrc in /etc/bashrc, profwinner=dot_profile in ~/.profile, and so on)

3. Set a unique variable in all these scripts (etc_bashrun=yes in /etc/bashrc, dot_profilerun=yes in ~/.profile, and so on)

4. Log on to a tty and to a gnome-terminal, and verify the values of the variables you set in questions 2 and 3. Which of the scripts were executed ? Which not ? Which was executed last ?

5. Does it matter on which line we set our variables in .bash_profile and .bashrc ?

6. Where is the command history stored ? And what about command history for Korn users ?

7. Define an alias 'dog' for the tac command in one of your profile scripts. Which script did you choose and why ?

Chapter 13. Pipes and Filters

13.1. pipes

One of the most powerful advantages of unix is the use of **pipes**, and the ability of almost any program to be used in a pipe. A pipe takes **stdout** from the previous command and sends it as **stdin** to the next command in the pipe. Pipes can have many commands, and all commands in a pipe can be running simultaneously.

What follows after the introduction to pipes is a number of small unix tools that do one specific task very well. These can be used as building blocks for more complex applications and solutions.

You still remember cat and tac right ?

```
[paul@RHEL4b pipes]$ cat count.txt
one
two
three
four
five
[paul@RHEL4b pipes]$ tac count.txt
five
four
three
two
one
[paul@RHEL4b pipes]$
```

A pipe is represented by a vertical bar **|** in between two commands. Below a very simple pipe.

```
[paul@RHEL4b pipes]$ cat count.txt | tac
five
four
three
two
one
[paul@RHEL4b pipes]$
```

But pipes can be longer, as in this example.

```
[paul@RHEL4b pipes]$ cat count.txt | tac | tac
one
two
three
four
five
[paul@RHEL4b pipes]$
```

Remember that I told you in the beginning of this book that the cat command is actually doing nothing ?

```
[paul@RHEL4b pipes]$ tac count.txt | cat | cat | cat | cat | cat
five
four
three
two
one
[paul@RHEL4b pipes]$
```

13.2. tee

Writing long pipes in unix is fun, but sometimes you might want intermediate results. This is were **tee** comes in handy, tee outputs both to a file and to stdout. So tee is almost the same as cat, except that it has two identical outputs.

```
[paul@RHEL4b pipes]$ tac count.txt | tee temp.txt | tac
one
two
three
four
five
[paul@RHEL4b pipes]$ cat temp.txt
five
four
three
two
one
[paul@RHEL4b pipes]$
```

13.3. grep

Time for the real tools now. With all the uses of **grep** you can probably fill a book. The most common use of grep is to filter results on keywords.

```
[paul@RHEL4b pipes]$ cat tennis.txt
Amelie Mauresmo, Fra
Kim Clijsters, BEL
Justine Henin, Bel
Serena Williams, usa
Venus Williams, USA
[paul@RHEL4b pipes]$ cat tennis.txt | grep Williams
Serena Williams, usa
Venus Williams, USA
[paul@RHEL4b pipes]$
```

You can write this without the cat.

```
[paul@RHEL4b pipes]$ grep Williams tennis.txt
Serena Williams, usa
Venus Williams, USA
[paul@RHEL4b pipes]$
```

One of the most useful options of grep is **grep -i** which filters in a case insensitive way.

```
[paul@RHEL4b pipes]$ grep Bel tennis.txt
Justine Henin, Bel
[paul@RHEL4b pipes]$ grep -i Bel tennis.txt
Kim Clijsters, BEL
Justine Henin, Bel
[paul@RHEL4b pipes]$
```

Another very useful option is **grep -v** which outputs lines not matching the string.

```
[paul@RHEL4b pipes]$ grep -v Fra tennis.txt
Kim Clijsters, BEL
Justine Henin, Bel
Serena Williams, usa
Venus Williams, USA
[paul@RHEL4b pipes]$
```

And of course, both options can be combined.

```
[paul@RHEL4b pipes]$ grep -vi usa tennis.txt
Amelie Mauresmo, Fra
Kim Clijsters, BEL
Justine Henin, Bel
[paul@RHEL4b pipes]$
```

13.4. cut

With **cut** you can select columns from files, depending on a delimiter or a count of bytes. The screenshot below uses cut to filter for the username and userid in the /etc/passwd file. It uses the colon as a delimiter, and select fields 1 and 3.

```
[[paul@RHEL4b pipes]$ cut -d: -f1,3 /etc/passwd | tail -4
Figo:510
Pfaff:511
Harry:516
Hermione:517
[paul@RHEL4b pipes]$
```

When using a space as the delimiter for cut, you have to quote the space.

```
[paul@RHEL4b pipes]$ cut -d" " -f1 tennis.txt
Amelie
Kim
Justine
Serena
Venus
[paul@RHEL4b pipes]$
```

One last example, cutting the second to the seventh character of /etc/passwd.

```
[paul@RHEL4b pipes]$ cut -c2-7 /etc/passwd | tail -4
igo:x:
faff:x
arry:x
ermion
[paul@RHEL4b pipes]$
```

13.5. tr

You can translate characters with **tr**. The screenshot translates all occurences of e to E.

```
[paul@RHEL4b pipes]$ cat tennis.txt
Amelie Mauresmo, Fra
Kim Clijsters, BEL
Justine Henin, Bel
Serena Williams, usa
Venus Williams, USA
[paul@RHEL4b pipes]$ cat tennis.txt | tr 'e' 'E'
AmEliE MaurEsmo, Fra
Kim ClijstErs, BEL
JustinE HEnin, BEl
SErEna Williams, usa
VEnus Williams, USA
[paul@RHEL4b pipes]$
```

Here we set all letters to uppercase by defining two ranges.

```
[paul@RHEL4b pipes]$ cat tennis.txt | tr 'a-z' 'A-Z'
AMELIE MAURESMO, FRA
KIM CLIJSTERS, BEL
JUSTINE HENIN, BEL
SERENA WILLIAMS, USA
VENUS WILLIAMS, USA
[paul@RHEL4b pipes]$
```

Here we translate all newlines to spaces.

```
[paul@RHEL4b pipes]$ cat count.txt
one
two
three
four
five
[paul@RHEL4b pipes]$ cat count.txt | tr '\n' ' '
one two three four five [paul@RHEL4b pipes]$
```

The tr filter can also be used to squeeze multiple occurences of a character to one.

```
[paul@RHEL4b pipes]$ cat spaces.txt
one    two        three
     four   five  six
[paul@RHEL4b pipes]$ cat spaces.txt | tr -s ' '
```

```
one two three
 four five six
[paul@RHEL4b pipes]$
```

You can also use tr to 'encrypt' texts with rot13.

```
[paul@RHEL4b pipes]$ cat count.txt | tr 'a-z' 'nopqrstuvwxyzabcdefghijklm'
bar
gjb
guerr
sbhe
svir
[paul@RHEL4b pipes]$ cat count.txt | tr 'a-z' 'n-za-m'
bar
gjb
guerr
sbhe
svir
[paul@RHEL4b pipes]$
```

13.6. wc

Counting words, lines and characters is easy with **wc**.

```
[paul@RHEL4b pipes]$ wc tennis.txt
  5  15 100 tennis.txt
[paul@RHEL4b pipes]$ wc -l tennis.txt
5 tennis.txt
[paul@RHEL4b pipes]$ wc -w tennis.txt
15 tennis.txt
[paul@RHEL4b pipes]$ wc -c tennis.txt
100 tennis.txt
[paul@RHEL4b pipes]$
```

How many users are logged on to this system ?

```
[paul@RHEL4b pipes]$ who
root     tty1         Jul 25 10:50
paul     pts/0        Jul 25 09:29 (laika)
Harry    pts/1        Jul 25 12:26 (barry)
paul     pts/2        Jul 25 12:26 (pasha)
[paul@RHEL4b pipes]$ who | wc -l
4
[paul@RHEL4b pipes]$
```

13.7. sort

Sorting is always useful. The **sort** filter has a lot of options. How about a sorted list of logged on users.

```
[paul@RHEL4b pipes]$ who | cut -d' ' -f1 | sort
```

```
Harry
paul
paul
root
[paul@RHEL4b pipes]$
```

Sorting on column 1 or column 2.

```
[paul@RHEL4b pipes]$ sort -k1 country.txt
Belgium, Brussels, 10
France, Paris, 60
Germany, Berlin, 100
Iran, Teheran, 70
Italy, Rome, 50
[paul@RHEL4b pipes]$ sort -k2 country.txt
Germany, Berlin, 100
Belgium, Brussels, 10
France, Paris, 60
Italy, Rome, 50
Iran, Teheran, 70
[paul@RHEL4b pipes]$
```

The screenshot below shows the difference between an alfabetical sort and a numerical sort (both on the third column).

```
[paul@RHEL4b pipes]$ sort -k3 country.txt
Belgium, Brussels, 10
Germany, Berlin, 100
Italy, Rome, 50
France, Paris, 60
Iran, Teheran, 70
[paul@RHEL4b pipes]$ sort -n -k3 country.txt
Belgium, Brussels, 10
Italy, Rome, 50
France, Paris, 60
Iran, Teheran, 70
Germany, Berlin, 100
[paul@RHEL4b pipes]$
```

13.8. uniq

With **uniq** you can remove duplicates from a sorted list. Here's a sorted list of logged on users, first with and then without duplicates.

```
[paul@RHEL4b pipes]$ who | cut -d' ' -f1 | sort
Harry
paul
paul
root
[paul@RHEL4b pipes]$ who | cut -d' ' -f1 | sort | uniq
Harry
paul
root
[paul@RHEL4b pipes]$
```

13.9. find

The **find** tool is used very often in linux. Find is useful at the start of a pipe, to search for files. Here are some examples. In real life, you will want to add 2>/dev/null to the command lines to avoid cluttering your screen with error messages.

Find all files in /etc and put the list in etcfiles.txt

```
find /etc > etcfiles.txt
```

Find all files of the entire system and put the list in allfiles.txt

```
find / > allfiles.txt
```

Find files that end in .conf in the current directory (and all subdirs).

```
find . -name "*.conf"
```

Find files of type file (so not directory or pipe...) that end in .conf.

```
find . -type f -name "*.conf"
```

Find files of type directory that end in .bak.

```
find /data -type d -name "*.bak"
```

Find files that are newer than file44.txt

```
find . -newer fil44.txt
```

Find can also execute another command on every file found. This example will look for *.odf files and copy them to /backup/.

```
find "/data/*.odf" -exec cp {} /backup/ \;
```

Find can also execute, after your confirmation, another command on every file found. This example will remove *.odf files if you approve of it for every file found.

```
find "/data/*.odf" -ok rm {} \;
```

The find tool can do much more, see the man page.

13.10. locate

The **locate** tool is very different from **find** in that it uses an index to locate files. This is a lot faster than traversing all the directories, but it also means that it is always outdated. If the index does not exist yet, then you have to create it (as root on Red Hat Enterprise Linux) with the **updatedb** command.

```
[paul@RHEL4b ~]$ locate Samba
warning: locate: could not open database: /var/lib/slocate/slocate.db:...
warning: You need to run the 'updatedb' command (as root) to create th...
Please have a look at /etc/updatedb.conf to enable the daily cron job.
[paul@RHEL4b ~]$ updatedb
fatal error: updatedb: You are not authorized to create a default sloc...
```

```
[paul@RHEL4b ~]$ su -
Password:
[root@RHEL4b ~]# updatedb
[root@RHEL4b ~]#
```

13.11. diff

To compare two files line by line, you can use **diff**. To ignore blanks, use **diff -b**, and to ignore case, use **diff -i**.

In this examples diff tells you 2c2 the second line in file one was changed with the second line in file two.

```
[paul@RHEL4b test]$ cat > count.txt
one
two
three
four
[paul@RHEL4b test]$ cat > count2.txt
one
Two
three
four
[paul@RHEL4b test]$ diff count.txt count2.txt
2c2
< two
---
> Two
[paul@RHEL4b test]$
```

Another example of diff. The second file now has one more line than the first file. After line 2, a line was added as line 3 (2a3) to the second file.

```
[paul@RHEL4b test]$ cat > count.txt
one
two
four
[paul@RHEL4b test]$ cat > count2.txt
one
two
three
four
[paul@RHEL4b test]$ diff count.txt count2.txt
2a3
> three
[paul@RHEL4b test]$
```

13.12. comm

You can use **comm** to quickly compare two sorted files. By default comm will output three columns. In this example, Abba, Cure and Queen are in both lists, Bowie and Sweet are only in the first file, Turner is only in the second.

```
[paul@RHEL4b test]$ cat > list1.txt
Abba
Bowie
Cure
Queen
Sweet
[paul@RHEL4b test]$ cat > list2.txt
Abba
Cure
Queen
Turner
[paul@RHEL4b test]$ comm list1.txt list2.txt
                Abba
Bowie
                Cure
                Queen
Sweet
        Turner
[paul@RHEL4b test]$
```

13.13. compress

Users never have enough space, so compression comes in handy. The **compress** command can make files take up less space. You can get the original back with **uncompress**. In the backup chapter we will also discuss gzip, gunzip, bzip2 and bunzip2.

```
[paul@RHEL4b test]$ ls -lh
total 19M
-rw-rw-r--  1 paul paul 19M Jul 26 04:21 allfiles.txt
[paul@RHEL4b test]$ compress allfiles.txt
[paul@RHEL4b test]$ ls -lh
total 3.2M
-rw-rw-r--  1 paul paul 3.2M Jul 26 04:21 allfiles.txt.Z
[paul@RHEL4b test]$ uncompress allfiles.txt
[paul@RHEL4b test]$ ls -lh
total 19M
-rw-rw-r--  1 paul paul 19M Jul 26 04:21 allfiles.txt
[paul@RHEL4b test]$
```

13.14. od

European humans like to work with ascii characters, but computers store files in bytes. The example below creates a simple file, and then uses **od** to show the contents of the file in hexadecimal bytes, in octal bytes and in ascii (or backslashed) characters.

```
paul@laika:~/test$ cat > text.txt
abcdefg
1234567
paul@laika:~/test$ od -t x1 text.txt
0000000 61 62 63 64 65 66 67 0a 31 32 33 34 35 36 37 0a
0000020
paul@laika:~/test$ od -b text.txt
```

```
0000000 141 142 143 144 145 146 147 012 061 062 063 064 065 066 067 012
0000020
paul@laika:~/test$ od -c text.txt
0000000   a   b   c   d   e   f   g  \n   1   2   3   4   5   6   7  \n
0000020
paul@laika:~/test$
```

13.15. other tools and filters

You might want to look at expand, unexpand, pr, nl, fmt, paste, join, sed, awk, ...

13.16. Practice tools and filters

1. Explain the difference between these two commands. This question is very important. If you don't know the answer, then look back at the bash chapters.

```
find . -name "*.txt"
```

```
find . -name *.txt
```

2. Explain the difference between these two statements. Will they both work when there are 200 .odf files in /data/ ? How about when there are 2 million .odf files ?

```
find /data -name "*.odf" > data_odf.txt
```

```
find /data/*.odf > data_odf.txt
```

3. Write a find command that finds all files created after january 30th this year.

4. Write a find command that finds all *.odf files created in september last year.

5. Put a sorted list of all bash users in bashusers.txt.

6. Put a sorted list of all bash users, with their username, userid and home directory in bashusers.info.

7. Make a list of all non-bash and non-korn users.

8. Make a list of all files (not directories) in /etc/ that contain the string smb, nmb or samba.

9. Look at the output of /sbin/ifconfig. Make an ipconfig command that shows only the nic name (eth0), the ip address and the subnet mask.

10. Make a command abc that removes all non-letters from a file (and replaces them with spaces).

11. Count the number of *.conf files in /etc and all its subdirs.

12. Two commands that do the same thing: copy *.odf files to /backup/ . What would be a reason to replace the first command with the second ? Again, this is an important question.

```
cp -r /data/*.odf /backup/
```

```
find /data -name "*.odf" -exec cp {} /backup/ \;
```

13. Create a file called loctest.txt. Can you find this file with locate ? Why not ? How do you make locate find this file ?

14. Create a file named text.txt that contains this sentence: The zun is shining today. Create a file DICT that contains the words "is shining sun the today", one word on each line. The first file is a text, the second file is a dictionary. Now create a spell checker that uses those two files and outputs the misspelled words (in this case that would be 'zun').

15. Use find and -exec to rename all .htm files to .html.

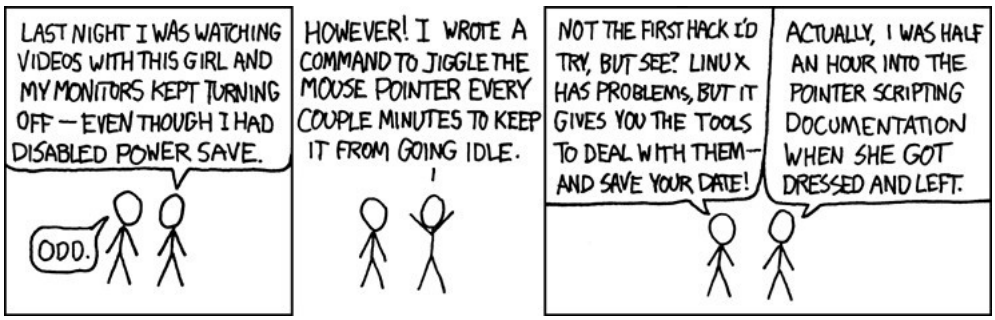
16. Find the hexadecimal byte value for ascii characters : " 'space' 'tab' A and a .

17. List all files in the current directory of size between 10 and 20 bytes.

18. List all files in your home directory that have more than one hard link (hint: use the find tool).

Always take time to properly **document** every script that you write!

Image copied from **xkcd.com**.



13.17. Solutions: tools and filters

1. The shell will not touch the *.txt because it is between double quotes. The find tool will look in the current directory for all files ending in .txt.

```
find . -name "*.txt"
```

The shell will expand the *.txt to all files in the current directory that end in .txt. Then find will give you a syntax error.

```
find . -name *.txt
```

14. The one line spell checker.

```
[paul]$ echo "The zun is shining today" > text.txt
[paul]$ cat > DICT
is
shining
sun
the
today
[paul]$ cat text.txt| tr 'A-Z ' 'a-z\n' |sort|uniq|comm -2 -3 - DICT
zun
[paul]$
```

18. Use find to look in your home directory(~) for regular files(-type f) that do not(!) have one hard link(-links 1).

```
find ~ ! -links 1 -type f
```

Appendix A. Keyboard settings

A.1. About Keyboard Layout

Many people (like US-Americans) prefer the default US-qwerty keyboard layout. So when you are not from the USA and want a local keyboard layout on your system, then the best practice is to select this keyboard at installation time. Then the keyboard layout will always be correct. Also, whenever you use ssh to remotely manage a linux system, your local keyboard layout will be used, independent of the server keyboard configuration. So you will not find much information on changing keyboard layout on the fly on linux, because not many people need it. Below are some tips to help you.

A.2. X Keyboard Layout

This is the relevant portion in /etc/X11/xorg.conf, first for Belgian azerty, then for US-qwerty.

```
[paul@RHEL5 ~]$ grep -i xkb /etc/X11/xorg.conf
        Option      "XkbModel" "pc105"
        Option      "XkbLayout" "be"


[paul@RHEL5 ~]$ grep -i xkb /etc/X11/xorg.conf
        Option      "XkbModel" "pc105"
        Option      "XkbLayout" "us"
```

When in Gnome or KDE or any other graphical environment, look in the graphical menu in preferences, there will be a keyboard section to choose your layout. Use the graphical menu instead of editing xorg.conf.

A.3. Shell Keyboard Layout

When in bash, take a look in the /etc/sysconfig/keyboard file. Below a sample US-qwerty configuration, followed by a Belgian azerty configuration.

```
[paul@RHEL5 ~]$ cat /etc/sysconfig/keyboard
KEYBOARDTYPE="pc"
KEYTABLE="us"


[paul@RHEL5 ~]$ cat /etc/sysconfig/keyboard
KEYBOARDTYPE="pc"
KEYTABLE="be-latin1"
```

The keymaps themselves can be found in /usr/share/keymaps or /lib/kbd/keymaps.

```
[paul@RHEL5 ~]$ ls -l /lib/kbd/keymaps/
total 52
drwxr-xr-x 2 root root 4096 Apr  1 00:14 amiga
drwxr-xr-x 2 root root 4096 Apr  1 00:14 atari
drwxr-xr-x 8 root root 4096 Apr  1 00:14 i386
drwxr-xr-x 2 root root 4096 Apr  1 00:14 include
drwxr-xr-x 4 root root 4096 Apr  1 00:14 mac
lrwxrwxrwx 1 root root    3 Apr  1 00:14 ppc -> mac
drwxr-xr-x 2 root root 4096 Apr  1 00:14 sun
```

Index

Symbols

A

B

C

D

E

U

V

W

X

Z