


Chapter 1. GNU/Linux tutorials

Table of Contents

I think learning a computer system is like learning a new foreign language. Although tutorial books and documentation are helpful, you have to practice it yourself. In order to help you get started smoothly, I elaborate a few basic points.

The powerful design of Debian GNU/Linux comes from the Unix operating system, i.e., a multiuser, multitasking operating system. You must learn to take advantage of the power of these features and similarities between Unix and GNU/Linux.

Don't shy away from Unix oriented texts and don't rely solely on GNU/Linux texts, as this robs you of much useful information.

Note

If you have been using any Unix-like system for a while with command line tools, you probably know everything I explain here. Please use this as a reality check and refresher.

1.1. Console basics

1.1.1. The shell prompt

Upon starting the system, you are presented with the character based login screen if you did not install X Window System with the display manager such as **gdm3**. Suppose your hostname is **foo**, the login prompt looks as

follows.

```
foo login:
```

If you did install a GUI environment such as GNOME or KDE, then you can get to a login prompt by Ctrl-Alt-F1, and you can return to the GUI environment via Alt-F7 (see Section 1.1.6, “Virtual consoles” below for more).

At the login prompt, you type your username, e.g. **penguin**, and press the Enter-key, then type your password and press the Enter-key again.

Note

Following the Unix tradition, the username and password of the Debian system are case sensitive. The username is usually chosen only from the lowercase. The first user account is usually created during the installation. Additional user accounts can be created with adduser(8) by root.

The system starts with the greeting message stored in "**/etc/motd**" (Message Of The Day) and presents a command prompt.

```
Debian GNU/Linux jessie/sid foo tty1
foo login: penguin
Password:
Last login: Mon Sep 23 19:36:44 JST 2013 on tty3
Linux snoopy 3.11-1-amd64 #1 SMP Debian 3.11.6-2
(2013-11-01) x86_64

The programs included with the Debian GNU/Linux system are
free software;
the exact distribution terms for each program are described
in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the
extent
permitted by applicable law.
foo:~$
```

Here, the main part of the greeting message can be customized by editing the "**/etc/motd.tail**" file. The first line is generated from the system information using "**uname -snrvm**".

Now you are in the shell. The shell interprets your commands.

1.1.2. The shell prompt under X

If you installed X Window System with a display manager such as GNOME's **gdm3** by selecting "Desktop environment" task during the installation, you

are presented with the graphical login screen upon starting your system. You type your username and your password to login to the non-privileged user account. Use tab to navigate between username and password, or use the mouse and primary click.

You can gain the shell prompt under X by starting a **x-terminal-emulator** program such as gnome-terminal(1), rxvt(1) or xterm(1). Under the GNOME Desktop environment, clicking "Applications" → "Accessories" → "Terminal" does the trick.

You can also see the section below Section 1.1.6, “Virtual consoles”.

Under some other Desktop systems (like **fluxbox**), there may be no obvious starting point for the menu. If this happens, just try (right) clicking the background of the desktop screen and hope for a menu to pop-up.

1.1.3. The root account

The root account is also called superuser or privileged user. From this account, you can perform the following system administration tasks.

- Read, write, and remove any files on the system irrespective of their file permissions
- Set file ownership and permissions of any files on the system
- Set the password of any non-privileged users on the system
- Login to any accounts without their passwords

This unlimited power of root account requires you to be considerate and responsible when using it.

Warning

Never share the root password with others.

Note

File permissions of a file (including hardware devices such as CD-ROM etc. which are just another file for the Debian system) may render it unusable or inaccessible by non-root users. Although the use of root account is a quick way to test this kind of situation, its resolution should be done through proper setting of file permissions and user's group membership (see Section 1.2.3, “Filesystem permissions”).

1.1.4. The root shell prompt

Here are a few basic methods to gain the root shell prompt by using the root password.

- Type **root** at the character based login prompt.

- Click "Applications" → "Accessories" → "Root Terminal", under the GNOME Desktop environment.
- Type "**su -l**" from any user shell prompt.
  - This does not preserve the environment of the current user.
- Type "**su**" from any user shell prompt.
  - This preserves some of the environment of the current user.

1.1.5. GUI system administration tools

When your desktop menu does not start GUI system administration tools automatically with the appropriate privilege, you can start them from the root shell prompt of the X terminal emulator, such as gnome-terminal(1), rxvt(1), or xterm(1). See Section 1.1.4, “The root shell prompt” and Section 7.8.4, “Running X clients as root”.

Warning

Never start the X display/session manager under the root account by typing in **root** to the prompt of the display manager such as gdm3(1).

Warning

Never run untrusted remote GUI program under X Window when critical information is displayed since it may eavesdrop your X screen.

1.1.6. Virtual consoles

In the default Debian system, there are six switchable VT100-like character consoles available to start the command shell directly on the Linux host. Unless you are in a GUI environment, you can switch between the virtual consoles by pressing the **Left-Alt-key** and one of the **F1** — **F6** keys simultaneously. Each character console allows independent login to the account and offers the multiuser environment. This multiuser environment is a great Unix feature, and very addictive.

If you are under the X Window System, you gain access to the character console 1 by pressing **Ctrl-Alt-F1** key, i.e., the **left-Ctrl-key**, the **left-Alt-key**, and the **F1-key** are pressed together. You can get back to the X Window System, normally running on the virtual console 7, by pressing **Alt-F7**.

You can alternatively change to another virtual console, e.g. to the console 1, from the commandline.

```
# chvt 1
```

1.1.7. How to leave the command prompt

You type **Ctrl-D**, i.e., the **left-Ctrl-key** and the **d-key** pressed together, at the command prompt to close the shell activity. If you are at the character console, you return to the login prompt with this. Even though these control characters are referred as "control D" with the upper case, you do not need to press the Shift-key. The short hand expression, **^D**, is also used for **Ctrl-D**. Alternately, you can type "exit".

If you are at x-terminal-emulator(1), you can close **x-terminal-emulator** window with this.

1.1.8. How to shutdown the system

Just like any other modern OS where the file operation involves caching data in memory for improved performance, the Debian system needs the proper shutdown procedure before power can safely be turned off. This is to maintain the integrity of files, by forcing all changes in memory to be written to disk. If the software power control is available, the shutdown procedure automatically turns off power of the system. (Otherwise, you may have to press power button for few seconds after the shutdown procedure.)

You can shutdown the system under the normal multiuser mode from the commandline.

```
# shutdown -h now
```

You can shutdown the system under the single-user mode from the commandline.

```
# poweroff -i -f
```

Alternatively, you may type **Ctrl-Alt-Delete** (The **left-Ctrl-key**, the **left-Alt-Key**, and the **Delete** are pressed together) to shutdown if "**/etc/inittab**" contains "**ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -h now**" in it. See inittab(5) for details.

See Section 6.9.6, “How to shutdown the remote system on SSH”.

1.1.9. Recovering a sane console

When the screen goes berserk after doing some funny things such as "**cat <some-binary-file>**", type "**reset**" at the command prompt. You may not be able to see the command echoed as you type. You may also issue "**clear**" to clean up the screen.

1.1.10. Additional package suggestions for the newbie

Although even the minimal installation of the Debian system without any desktop environment tasks provides the basic Unix functionality, it is a good idea to install few additional commandline and curses based character terminal packages such as **mc** and **vim** with apt-get(8) for beginners to get started by the following.

```
# apt-get update
 ...
# apt-get install mc vim sudo
 ...
```

If you already had these packages installed, no new packages are installed.

Table 1.1. List of interesting text-mode program packages

package	popcon	size	description
mc	V:78, I:236	1341	A text-mode full-screen file manager
sudo	V:368, I:758	2740	A program to allow limited root privileges to users
vim	V:148, I:379	2063	Unix text editor Vi IMproved, a programmers text editor (standard version)
vim-tiny	V:92, I:965	948	Unix text editor Vi IMproved, a programmers text editor (compact version)
emacs23	V:36, I:86	12936	GNU project Emacs, the Lisp based extensible text editor (version 23)
w3m	V:258, I:878	2131	Text-mode WWW browsers
gpm	V:18, I:28	500	The Unix style cut-and-paste on the text console (daemon)

It may be a good idea to read some informative documentations.

Table 1.2. List of informative documentation packages

package	popcon	size	description
doc-debian	I:861	147	Debian Project documentation, (Debian FAQ) and other documents
debian-policy	I:102	3684	Debian Policy Manual and related documents
developers-reference	I:8	1162	Guidelines and information for Debian developers
maint-guide	I:6	1023	Debian New Maintainers' Guide
debian-history	I:1	4625	History of the Debian Project
debian-faq	I:813	1294	Debian FAQ
sysadmin-guide	I:1	964	The Linux System Administrators' Guide

You can install some of these packages by the following.

```
# apt-get install package_name
```

1.1.11. An extra user account

If you do not want to use your main user account for the following training activities, you can create a training user account, e.g. **fish** by the following.

```
# adduser fish
```

Answer all questions.

This creates a new account named as **fish**. After your practice, you can remove this user account and its home directory by the following.

```
# deluser --remove-home fish
```

1.1.12. sudo configuration

For the typical single user workstation such as the desktop Debian system on the laptop PC, it is common to deploy simple configuration of sudo(8) as follows to let the non-privileged user, e.g. **penguin**, to gain administrative privilege just with his user password but without the root password.

```
# echo "penguin  ALL=(ALL) ALL" >> /etc/sudoers
```

Alternatively, it is also common to do as follows to let the non-privileged user, e.g. **penguin**, to gain administrative privilege without any password.

```
# echo "penguin  ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers
```

This trick should only be used for the single user workstation which you administer and where you are the only user.

Warning

Do not set up accounts of regular users on multiuser workstation like this because it would be very bad for system security.

Caution

The password and the account of the **penguin** in the above example requires as much protection as the root password and the root account.

Caution

Administrative privilege in this context belongs to someone authorized to perform the system administration task on the workstation. Never give some manager in the Admin department of your company or your boss such privilege unless they are authorized and capable.

Note

For providing access privilege to limited devices and limited files, you should consider to use **group** to provide limited access instead of using the **root** privilege via sudo(8).

Note

With more thoughtful and careful configuration, sudo(8) can grant limited administrative privileges to other users on a shared system without sharing the root password. This can help with accountability with hosts with multiple administrators so you can tell who did what. On the other hand, you might not want anyone else to have such privileges.

1.1.13. Play time

Now you are ready to play with the Debian system without risks as long as you use the non-privileged user account.

This is because the Debian system is, even after the default installation, configured with proper file permissions which prevent non-privileged users from damaging the system. Of course, there may still be some holes which can be exploited but those who worry about these issues should not be reading this section but should be reading Securing Debian Manual.

We learn the Debian system as a Unix-like system with the following.

- Section 1.2, “Unix-like filesystem” (basic concept)
- Section 1.3, “Midnight Commander (MC)” (survival method)
- Section 1.4, “The basic Unix-like work environment” (basic method)
- Section 1.5, “The simple shell command” (shell mechanism)
- Section 1.6, “Unix-like text processing” (text processing method)

1.2. Unix-like filesystem

In GNU/Linux and other Unix-like operating systems, files are organized into directories. All files and directories are arranged in one big tree rooted at "/". It's called a tree because if you draw the filesystem, it looks like a tree but it is upside down.

These files and directories can be spread out over several devices. mount(8) serves to attach the filesystem found on some device to the big file tree. Conversely, umount(8) detaches it again. On recent Linux kernels, mount(8) with some options can bind part of a file tree somewhere else or can mount filesystem as shared, private, slave, or unbindable. Supported mount options for each filesystem are available in "**/share/doc/linux-doc-*/Documentation/filesystems/**".

Directories on Unix systems are called **folders** on some other systems. Please also note that there is no concept for **drive** such as "**A:**" on any Unix system. There is one filesystem, and everything is included. This is a huge advantage compared to Windows.

1.2.1. Unix file basics

Here are some Unix file basics.

- Filenames are **case sensitive**. That is, "**MYFILE**" and "**MyFile**" are different files.
- The **root directory** means root of the filesystem referred as simply "**/**". Don't confuse this with the home directory for the root user: "**/root**".
- Every directory has a name which can contain any letters or symbols **except "/"**. The root directory is an exception; its name is "**/**" (pronounced "slash" or "the root directory") and it cannot be renamed.
- Each file or directory is designated by a **fully-qualified filename**, **absolute filename**, or **path**, giving the sequence of directories which must be passed through to reach it. The three terms are synonymous.
- All **fully-qualified filenames** begin with the "**/**" directory, and there's a "**/**" between each directory or file in the filename. The first "**/**" is the top level directory, and the other "**/**"'s separate successive subdirectories, until we reach the last entry which is the name of the actual file. The words used here can be confusing. Take the following **fully-qualified filename** as an example: "**/usr/share/keytables/us.map.gz**". However, people also refers to its basename "**us.map.gz**" alone as a filename.
- The root directory has a number of branches, such as "**/etc/**" and "**/usr/**". These subdirectories in turn branch into still more subdirectories, such as "**/etc/init.d/**" and "**/usr/local/**". The whole thing viewed collectively is called the **directory tree**. You can think of an absolute filename as a route from the base of the tree ("**/**") to the end of some branch (a file). You also hear people talk about the directory tree as if it were a **family** tree encompassing all direct descendants of a single figure called the root directory ("**/**"): thus subdirectories have **parents**, and a path shows the complete ancestry of a file. There are also relative paths that begin somewhere other than the root directory. You should remember that the directory "**../**" refers to the parent directory. This terminology also applies to other directory like structures, such as hierarchical data structures.
- There's no special directory path name component that corresponds to a physical device, such as your hard disk. This differs from RT-11, CP/M, OpenVMS, MS-DOS, AmigaOS, and Microsoft Windows, where the path contains a device name such as "**C:**". (However, directory

entries do exist that refer to physical devices as a part of the normal filesystem. See Section 1.2.2, “Filesystem internals”.)

Note

While you **can** use almost any letters or symbols in a file name, in practice it is a bad idea to do so. It is better to avoid any characters that often have special meanings on the command line, including spaces, tabs, newlines, and other special characters: **`{ } ( ) [ ] ' ` " \ / > < | ; ! # & ^ * % @ $`** . If you want to separate words in a name, good choices are the period, hyphen, and underscore. You could also capitalize each word, "**`LikeThis`**". Experienced Linux users tend to avoid spaces in filenames.

Note

The word "root" can mean either "root user" or "root directory". The context of their usage should make it clear.

Note

The word **path** is used not only for **fully-qualified filename** as above but also for the **command search path**. The intended meaning is usually clear from the context.

The detailed best practices for the file hierarchy are described in the Filesystem Hierarchy Standard ("**`/usr/share/doc/debian-policy/fhs/fhs-2.3.txt.gz`**" and hier(7)). You should remember the following facts as the starter.

Table 1.3. List of usage of key directories

directory	usage of the directory
`/`	the root directory
`/etc/`	system wide configuration files
`/var/log/`	system log files
`/home/`	all the home directories for all non-privileged users

1.2.2. Filesystem internals

Following the **Unix tradition**, the Debian GNU/Linux system provides the filesystem under which physical data on hard disks and other storage devices reside, and the interaction with the hardware devices such as console screens and remote serial consoles are represented in an unified manner under "**`/dev/`**".

Each file, directory, named pipe (a way two programs can share data), or physical device on a Debian GNU/Linux system has a data structure called an inode which describes its associated attributes such as the user who owns it (owner), the group that it belongs to, the time last accessed, etc. If you are really interested, see "**`/usr/include/linux/fs.h`**" for the exact definition of "**`struct inode`**" in the Debian GNU/Linux system. The idea of representing just about everything in the filesystem was a Unix innovation, and modern Linux kernels have developed this idea ever further. Now, even information about processes running in the computer can be found in the filesystem.

This abstract and unified representation of physical entities and internal processes is very powerful since this allows us to use the same command for the same kind of operation on many totally different devices. It is even possible to change the way the kernel works by writing data to special files that are linked to running processes.

Tip

If you need to identify the correspondence between the file tree and the physical entity, execute mount(8) with no arguments.

1.2.3. Filesystem permissions

Filesystem permissions of Unix-like system are defined for three categories of affected users.

- The **user** who owns the file (**u**)
- Other users in the **group** which the file belongs to (**g**)
- All **other** users (**o**) also referred to as "world" and "everyone"

For the file, each corresponding permission allows following actions.

- The **read** (**r**) permission allows owner to examine contents of the file.
- The **write** (**w**) permission allows owner to modify the file.
- The **execute** (**x**) permission allows owner to run the file as a command.

For the directory, each corresponding permission allows following actions.

- The **read** (**r**) permission allows owner to list contents of the directory.
- The **write** (**w**) permission allows owner to add or remove files in the directory.
- The **execute** (**x**) permission allows owner to access files in the directory.

Here, the **execute** permission on a directory means not only to allow reading of files in that directory but also to allow viewing their attributes,

such as the size and the modification time.

ls(1) is used to display permission information (and more) for files and directories. When it is invoked with the "**-l**" option, it displays the following information in the order given.

- **Type of file** (first character)
- Access **permission** of the file (nine characters, consisting of three characters each for user, group, and other in this order)
- **Number of hard links** to the file
- Name of the **user** who owns the file
- Name of the **group** which the file belongs to
- **Size** of the file in characters (bytes)
- **Date and time** of the file (mtime)
- **Name** of the file

Table 1.4. List of the first character of "`ls -l`" output

character	meaning
-	normal file
d	directory
l	symlink
c	character device node
b	block device node
p	named pipe
s	socket

chown(1) is used from the root account to change the owner of the file. chgrp(1) is used from the file's owner or root account to change the group of the file. chmod(1) is used from the file's owner or root account to change file and directory access permissions. Basic syntax to manipulate a **foo** file is the following.

```
# chown <newowner> foo
# chgrp <newgroup> foo
# chmod  [ugoa][+-=][rwxXst][,...] foo
```

For example, you can make a directory tree to be owned by a user **foo** and shared by a group **bar** by the following.

```
# cd /some/location/
# chown -R foo:bar .
# chmod -R ug+rwX,o=rX .
```

There are three more special permission bits.

- The **set user ID** bit (**s** or **S** instead of user's **x**)
- The **set group ID** bit (**s** or **S** instead of group's **x**)
- The **sticky** bit (**t** or **T** instead of other's **x**)

Here the output of "**`ls -l`**" for these bits is **capitalized** if execution bits hidden by these outputs are **unset**.

Setting **set user ID** on an executable file allows a user to execute the executable file with the owner ID of the file (for example **root**). Similarly, setting **set group ID** on an executable file allows a user to execute the executable file with the group ID of the file (for example **root**). Because these settings can cause security risks, enabling them requires extra caution.

Setting **set group ID** on a directory enables the BSD-like file creation scheme where all files created in the directory belong to the **group** of the directory.

Setting the **sticky bit** on a directory prevents a file in the directory from being removed by a user who is not the owner of the file. In order to secure contents of a file in world-writable directories such as "**`/tmp`**" or in group-writable directories, one must not only reset the **write** permission for the file but also set the **sticky bit** on the directory. Otherwise, the file can be removed and a new file can be created with the same name by any user who has write access to the directory.

Here are a few interesting examples of file permissions.

```
$ ls -l /etc/passwd /etc/shadow /dev/ppp /usr/sbin/exim4
crw------T 1 root root   108, 0 Oct 16 20:57 /dev/ppp
-rw-r--r-- 1 root root     2761 Aug 30 10:38 /etc/passwd
-rw-r----- 1 root shadow   1695 Aug 30 10:38 /etc/shadow
-rwsr-xr-x 1 root root   973824 Sep 23 20:04 /usr/sbin/exim4
$ ls -ld /tmp /var/tmp /usr/local /var/mail /usr/src
drwxrwxrwt 14 root root  20480 Oct 16 21:25 /tmp
drwxrwsr-x 10 root staff  4096 Sep 29 22:50 /usr/local
drwxr-xr-x 10 root root   4096 Oct 11 00:28 /usr/src
drwxrwsr-x  2 root mail   4096 Oct 15 21:40 /var/mail
drwxrwxrwt  3 root root   4096 Oct 16 21:20 /var/tmp
```

There is an alternative numeric mode to describe file permissions with chmod(1). This numeric mode uses 3 to 4 digit wide octal (radix=8) numbers.

Table 1.5. The numeric mode for file permissions in chmod(1) commands

digit	meaning
1st optional digit	sum of **set user ID** (=4), **set group ID** (=2), and **sticky bit** (=1)

digit	meaning
2nd digit	sum of **read** (=4), **write** (=2), and **execute** (=1) permissions for **user**
3rd digit	ditto for **group**
4th digit	ditto for **other**

This sounds complicated but it is actually quite simple. If you look at the first few (2-10) columns from "**ls -l**" command output and read it as a binary (radix=2) representation of file permissions ("-" being "0" and "rwx" being "1"), the last 3 digit of the numeric mode value should make sense as an octal (radix=8) representation of file permissions to you.

For example, try the following

```
$ touch foo bar
$ chmod u=rw,go=r foo
$ chmod 644 bar
$ ls -l foo bar
-rw-r--r-- 1 penguin penguin 0 Oct 16 21:39 bar
-rw-r--r-- 1 penguin penguin 0 Oct 16 21:35 foo
```

Tip

If you need to access information displayed by "**ls -l**" in shell script, you should use pertinent commands such as test(1), stat(1) and readlink(1). The shell builtin such as "**[**" or "**test**" may be used too.

1.2.4. Control of permissions for newly created files: umask

What permissions are applied to a newly created file or directory is restricted by the **umask** shell builtin command. See dash(1), bash(1), and builtins(7).

```
  (file permissions) = (requested file permissions) &
~(umask value)
```

Table 1.6. The umask value examples

umask	file permissions created	directory permissions created	usage
0022	**-rw-r--r--**	**-rwxr-xr-x**	writable only by the user
0002	**-rw-rw-r--**	**-rwxrwxr-x**	writable by the group

The Debian system uses a user private group (UPG) scheme as its default. A UPG is created whenever a new user is added to the system. A UPG has the

same name as the user for which it was created and that user is the only member of the UPG. UPG scheme makes it safe to set umask to **0002** since every user has their own private group. (In some Unix variants, it is quite common to setup all normal users belonging to a single **users** group and is a good idea to set umask to **0022** for security in such cases.)

Tip

Enable UPG by putting "**umask 002**" in the **~/.bashrc** file.

1.2.5. Permissions for groups of users (group)

In order to make group permissions to be applied to a particular user, that user needs to be made a member of the group using "**sudo vigr**" for **/etc/group** and "**sudo vigr -s**" for **/etc/gshadow**. You need to login after logout (or run "**exec newgrp**") to enable the new group configuration.

Note

Alternatively, you may dynamically add users to groups during the authentication process by adding "**auth optional pam_group.so**" line to "**/etc/pam.d/common-auth**" and setting "**/etc/security/group.conf**". (See Chapter 4, *Authentication*.)

The hardware devices are just another kind of file on the Debian system. If you have problems accessing devices such as CD-ROM and USB memory stick from a user account, you should make that user a member of the relevant group.

Some notable system-provided groups allow their members to access particular files and devices without **root** privilege.

Table 1.7. List of notable system-provided groups for file access

group	description for accessible files and devices
dialout	full and direct access to serial ports ("**/dev/ttyS[0-3]**")
dip	limited access to serial ports for **Dialup IP** connection to trusted peers
cdrom	CD-ROM, DVD+/-RW drives
audio	audio device
video	video device
scanner	scanner(s)
adm	system monitoring logs
staff	some directories for junior administrative work: "**/usr/local**", "**/home**"

Tip

You need to belong to the **dialout** group to reconfigure modem, dial anywhere, etc. But if **root** creates pre-defined configuration files for trusted peers in "**/etc/ppp/peers/**", you only need to belong to the **dip** group to create **Dialup IP** connection to those trusted peers using pppd(8), pon(1), and poff(1) commands.

Some notable system-provided groups allow their members to execute particular commands without **root** privilege.

Table 1.8. List of notable system provided groups for particular command executions

group	accessible commands
sudo	execute **sudo** without their password
lpadmin	execute commands to add, modify, and remove printers from printer databases

For the full listing of the system provided users and groups, see the recent version of the "Users and Groups" document in "**/usr/share/doc/base-passwd/users-and-groups.html**" provided by the **base-passwd** package.

See passwd(5), group(5), shadow(5), newgrp(1), vipw(8), vigr(8), and pam_group(8) for management commands of the user and group system.

1.2.6. Timestamps

There are three types of timestamps for a GNU/Linux file.

Table 1.9. List of types of timestamps

type	meaning
mtime	the file modification time (**ls -l**)
ctime	the file status change time (**ls -lc**)
atime	the last file access time (**ls -lu**)

Note

ctime is not file creation time.

- Overwriting a file changes all of the **mtime**, **ctime**, and **atime** attributes of the file.
- Changing ownership or permission of a file changes the **ctime** and

atime attributes of the file.

- Reading a file changes the **atime** of the file.

Note

Even simply reading a file on the Debian system normally causes a file write operation to update **atime** information in the **inode**. Mounting a filesystem with "**noatime**" or "**relatime**" option makes the system skip this operation and results in faster file access for the read. This is often recommended for laptops, because it reduces hard drive activity and saves power. See mount(8).

Use touch(1) command to change timestamps of existing files.

For timestamps, the **ls** command outputs different strings under non-English locale ("**fr_FR.UTF-8**") from under the old one ("**C**").

```
$ LANG=fr_FR.UTF-8  ls -l foo
-rw-rw-r-- 1 penguin penguin 0 oct. 16 21:35 foo
$ LANG=C  ls -l foo
-rw-rw-r-- 1 penguin penguin 0 Oct 16 21:35 foo
```

Tip

See Section 9.2.5, “Customized display of time and date” to customize "**ls -l**" output.

1.2.7. Links

There are two methods of associating a file "**foo**" with a different filename "**bar**".

- Hard link
  - Duplicate name for an existing file
  - "**ln foo bar**"
- Symbolic link or symlink
  - Special file that points to another file by name
  - "**ln -s foo bar**"

See the following example for changes in link counts and the subtle differences in the result of the **rm** command.

```
$ umask 002
$ echo "Original Content" > foo
$ ls -li foo
1449840 -rw-rw-r-- 1 penguin penguin 17 Oct 16 21:42 foo
$ ln foo bar     # hard link
$ ln -s foo baz  # symlink
$ ls -li foo bar baz
1449840 -rw-rw-r-- 2 penguin penguin 17 Oct 16 21:42 bar
1450180 lrwxrwxrwx 1 penguin penguin  3 Oct 16 21:47 baz ->
foo
1449840 -rw-rw-r-- 2 penguin penguin 17 Oct 16 21:42 foo
$ rm foo
$ echo "New Content" > foo
$ ls -li foo bar baz
1449840 -rw-rw-r-- 1 penguin penguin 17 Oct 16 21:42 bar
1450180 lrwxrwxrwx 1 penguin penguin  3 Oct 16 21:47 baz ->
foo
1450183 -rw-rw-r-- 1 penguin penguin 12 Oct 16 21:48 foo
$ cat bar
Original Content
$ cat baz
New Content
```

The hardlink can be made within the same filesystem and shares the same inode number which the "**-i**" option with ls(1) reveals.

The symlink always has nominal file access permissions of "**rwxrwxrwx**", as shown in the above example, with the effective access permissions dictated by permissions of the file that it points to.

Caution

It is generally a good idea not to create complicated symbolic links or hardlinks at all unless you have a very good reason. It may cause nightmares where the logical combination of the symbolic links results in loops in the filesystem.

Note

It is generally preferable to use symbolic links rather than hardlinks unless you have a good reason for using a hardlink.

The "**.**" directory links to the directory that it appears in, thus the link count of any new directory starts at 2. The "**..**" directory links to the parent directory, thus the link count of the directory increases with the addition of new subdirectories.

If you are just moving to Linux from Windows, it soon becomes clear how well-designed the filename linking of Unix is, compared with the nearest Windows equivalent of "shortcuts". Because it is implemented in the

filesystem, applications can't see any difference between a linked file and the original. In the case of hardlinks, there really is no difference.

1.2.8. Named pipes (FIFOs)

A named pipe is a file that acts like a pipe. You put something into the file, and it comes out the other end. Thus it's called a FIFO, or First-In-First-Out: the first thing you put in the pipe is the first thing to come out the other end.

If you write to a named pipe, the process which is writing to the pipe doesn't terminate until the information being written is read from the pipe. If you read from a named pipe, the reading process waits until there is nothing to read before terminating. The size of the pipe is always zero --- it does not store data, it just links two processes like the functionality offered by the shell "**|**" syntax. However, since this pipe has a name, the two processes don't have to be on the same command line or even be run by the same user. Pipes were a very influential innovation of Unix.

For example, try the following

```
$ cd; mkfifo mypipe
$ echo "hello" >mypipe & # put into background
[1] 8022
$ ls -l mypipe
prw-rw-r-- 1 penguin penguin 0 Oct 16 21:49 mypipe
$ cat mypipe
hello
[1]+  Done                    echo "hello" >mypipe
$ ls mypipe
mypipe
$ rm mypipe
```

1.2.9. Sockets

Sockets are used extensively by all the Internet communication, databases, and the operating system itself. It is similar to the named pipe (FIFO) and allows processes to exchange information even between different computers. For the socket, those processes do not need to be running at the same time nor to be running as the children of the same ancestor process. This is the endpoint for the inter process communication (IPC). The exchange of information may occur over the network between different hosts. The two most common ones are the Internet socket and the Unix domain socket.

Tip

"**`netstat -an`**" provides a very useful overview of sockets that are open on a given system.

1.2.10. Device files

Device files refer to physical or virtual devices on your system, such as your

hard disk, video card, screen, or keyboard. An example of a virtual device is the console, represented by "**/dev/console**".

There are 2 types of device files.

- **Character device**
  - Accessed one character at a time
  - 1 character = 1 byte
  - E.g. keyboard device, serial port, …
- **Block device**
  - accessed in larger units called blocks
  - 1 block > 1 byte
  - E.g. hard disk, …

You can read and write device files, though the file may well contain binary data which may be an incomprehensible-to-humans gibberish. Writing data directly to these files is sometimes useful for the troubleshooting of hardware connections. For example, you can dump a text file to the printer device "**/dev/lp0**" or send modem commands to the appropriate serial port "**/dev/ttyS0**". But, unless this is done carefully, it may cause a major disaster. So be cautious.

Note

For the normal access to a printer, use lp(1).

The device node number are displayed by executing ls(1) as the following.

```
$ ls -l /dev/sda /dev/sr0 /dev/ttyS0 /dev/zero
brw-rw---T  1 root disk     8,  0 Oct 16 20:57 /dev/sda
brw-rw---T+ 1 root cdrom   11,  0 Oct 16 21:53 /dev/sr0
crw-rw---T  1 root dialout  4, 64 Oct 16 20:57 /dev/ttyS0
crw-rw-rw-  1 root root     1,  5 Oct 16 20:57 /dev/zero
```

- "**/dev/sda**" has the major device number 8 and the minor device number 0. This is read/write accessible by users belonging to the **disk** group.
- "**/dev/sr0**" has the major device number 11 and the minor device number 0. This is read/write accessible by users belonging to the **cdrom** group.
- "**/dev/ttyS0**" has the major device number 4 and the minor device number 64. This is read/write accessible by users belonging to the **dialout** group.
- "**/dev/zero**" has the major device number 1 and the minor device

number 5. This is read/write accessible by anyone.

On the modern Linux system, the filesystem under "**/dev/**" is automatically populated by the udev(7) mechanism.

1.2.11. Special device files

There are some special device files.

Table 1.10. List of special device files

device file	action	description of response
/dev/null	read	return "end-of-file (EOF) character"
/dev/null	write	return nothing (a bottomless data dump pit)
/dev/zero	read	return "the **\0** (NUL) character" (not the same as the number zero ASCII)
/dev/random	read	return random characters from a true random number generator, delivering real entropy (slow)
/dev/urandom	read	return random characters from a cryptographically secure pseudorandom number generator
/dev/full	write	return the disk-full (ENOSPC) error

These are frequently used in conjunction with the shell redirection (see Section 1.5.8, “Typical command sequences and shell redirection”).

1.2.12. procfs and sysfs

The procfs and sysfs mounted on "**/proc**" and "**/sys**" are the pseudo-filesystem and expose internal data structures of the kernel to the userspace. In other word, these entries are virtual, meaning that they act as a convenient window into the operation of the operating system.

The directory "**/proc**" contains (among other things) one subdirectory for each process running on the system, which is named after the process ID (PID). System utilities that access process information, such as ps(1), get their information from this directory structure.

The directories under "**/proc/sys/**" contain interfaces to change certain kernel parameters at run time. (You may do the same through the specialized sysctl(8) command or its preload/configuration file "**/etc/sysctl.conf**".)

People frequently panic when they notice one file in particular - "**/proc/kcore**" - which is generally huge. This is (more or less) a copy of the content of your computer's memory. It's used to debug the kernel. It is a virtual file that points to computer memory, so don't worry about its size.

The directory under "**/sys**" contains exported kernel data structures, their attributes, and their linkages between them. It also contains interfaces to change certain kernel parameters at run time.

See "**proc.txt(.gz)**", "**sysfs.txt(.gz)**" and other related documents in the Linux kernel documentation ("**/usr/share/doc/linux-doc-*/Documentation/filesystems/***") provided by the **linux-doc-*** package.

1.2.13. tmpfs

The tmpfs is a temporary filesystem which keeps all files in the virtual memory. The data of the tmpfs in the page cache on memory may be swapped out to the swap space on disk as needed.

The directory "**/run**" is mounted as the tmpfs in the early boot process. This enables writing to it even when the directory "**/**" is mounted as read-only. This is the new location for the storage of transient state files and replaces several locations described in the Filesystem Hierarchy Standard version 2.3:

- "**/var/run**" → "**/run**"
- "**/var/lock**" → "**/run/lock**"
- "**/dev/shm**" → "**/run/shm**"

See "**tmpfs.txt(.gz)**" in the Linux kernel documentation ("**/usr/share/doc/linux-doc-*/Documentation/filesystems/***") provided by the **linux-doc-*** package.

1.3. Midnight Commander (MC)

Midnight Commander (MC) is a GNU "Swiss army knife" for the Linux console and other terminal environments. This gives newbie a menu driven console experience which is much easier to learn than standard Unix commands.

You may need to install the Midnight Commander package which is titled "**mc**" by the following.

```
$ sudo apt-get install mc
```

Use the mc(1) command to explore the Debian system. This is the best way to learn. Please explore few interesting locations just using the cursor keys and Enter key.

- "**/etc**" and its subdirectories
- "**/var/log**" and its subdirectories
- "**/usr/share/doc**" and its subdirectories
- "**/sbin**" and "**/bin**"

1.3.1. Customization of MC

In order to make MC to change working directory upon exit and **cd** to the directory, I suggest to modify "**~/.bashrc**" to include a script provided by

the **mc** package.

```
. /usr/lib/mc/mc.sh
```

See mc(1) (under the "**-P**" option) for the reason. (If you do not understand what exactly I am talking here, you can do this later.)

1.3.2. Starting MC

MC can be started by the following.

```
$ mc
```

MC takes care of all file operations through its menu, requiring minimal user effort. Just press F1 to get the help screen. You can play with MC just by pressing cursor-keys and function-keys.

> **Note**
>
> In some consoles such as gnome-terminal(1), key strokes of function-keys may be stolen by the console program. You can disable these features by "Edit" → "Keyboard Shortcuts" for **gnome-terminal**.

If you encounter character encoding problem which displays garbage characters, adding "**-a**" to MC's command line may help prevent problems.

If this doesn't clear up your display problems with MC, see Section 9.4.6, “The terminal configuration”.

1.3.3. File manager in MC

The default is two directory panels containing file lists. Another useful mode is to set the right window to "information" to see file access privilege information, etc. Following are some essential keystrokes. With the gpm(8) daemon running, one can use a mouse on Linux character consoles, too. (Make sure to press the shift-key to obtain the normal behavior of cut and paste in MC.)

Table 1.11. The key bindings of MC

key	key binding
F1	help menu
F3	internal file viewer
F4	internal editor
F9	activate pull down menu
F10	exit Midnight Commander
Tab	move between two windows
Insert or **Ctrl-T**	mark file for a multiple-file operation such as copy

key	key binding
Del	delete file (be careful---set MC to safe delete mode)
Cursor keys	self-explanatory

1.3.4. Command-line tricks in MC

- **cd** command changes the directory shown on the selected screen.
- **Ctrl-Enter** or **Alt-Enter** copies a filename to the command line. Use this with cp(1) and mv(1) commands together with command-line editing.
- **Alt-Tab** shows shell filename expansion choices.
- One can specify the starting directory for both windows as arguments to MC; for example, "**mc /etc /root**".
- **Esc** + **n-key** → **Fn** (i.e., **Esc** + **1** → **F1**, etc.; **Esc** + **0** → **F10**)
- Pressing **Esc** before the key has the same effect as pressing the **Alt** and the key together.; i.e., type **Esc** + **c** for **Alt-C**. **Esc** is called meta-key and sometimes noted as "**M-**".

1.3.5. The internal editor in MC

The internal editor has an interesting cut-and-paste scheme. Pressing **F3** marks the start of a selection, a second **F3** marks the end of selection and highlights the selection. Then you can move your cursor. If you press F6, the selected area is moved to the cursor location. If you press F5, the selected area is copied and inserted at the cursor location. **F2** saves the file. **F10** gets you out. Most cursor keys work intuitively.

This editor can be directly started on a file using one of the following commands.

```
$ mc -e filename_to_edit
```

```
$ mcedit filename_to_edit
```

This is not a multi-window editor, but one can use multiple Linux consoles to achieve the same effect. To copy between windows, use Alt-F<n> keys to switch virtual consoles and use "File→Insert file" or "File→Copy to file" to move a portion of a file to another file.

This internal editor can be replaced with any external editor of choice.

Also, many programs use the environment variables "**$EDITOR**" or "**$VISUAL**" to decide which editor to use. If you are uncomfortable with vim(1) or nano(1) initially, you may set these to "**mcedit**" by adding the following lines to "**~/.bashrc**".

```
export EDITOR=mcedit
export VISUAL=mcedit
```

I do recommend setting these to "**vim**" if possible.

If you are uncomfortable with vim(1), you can keep using mcedit(1) for most system maintenance tasks.

1.3.6. The internal viewer in MC

MC is a very smart viewer. This is a great tool for searching words in documents. I always use this for files in the "**/usr/share/doc**" directory. This is the fastest way to browse through masses of Linux information. This viewer can be directly started using one of the following commands.

```
$ mc -v path/to/filename_to_view
```

```
$ mcview path/to/filename_to_view
```

1.3.7. Auto-start features of MC

Press Enter on a file, and the appropriate program handles the content of the file (see Section 9.3.11, “Customizing program to be started”). This is a very convenient MC feature.

Table 1.12. The reaction to the enter key in MC

file type	reaction to enter key
executable file	execute command
man file	pipe content to viewer software
html file	pipe content to web browser
"***.tar.gz**" and "***.deb**" file	browse its contents as if subdirectory

In order to allow these viewer and virtual file features to function, viewable files should not be set as executable. Change their status using chmod(1) or via the MC file menu.

1.3.8. FTP virtual filesystem of MC

MC can be used to access files over the Internet using FTP. Go to the menu by pressing **F9**, then type "**p**" to activate the FTP virtual filesystem. Enter a URL in the form "**username:passwd@hostname.domainname**", which retrieves a remote directory that appears like a local one.

Try "[http.us.debian.org/debian]" as the URL and browse the Debian archive.

1.4. The basic Unix-like work environment

Although MC enables you to do almost everything, it is very important for

you to learn how to use the command line tools invoked from the shell prompt and become familiar with the Unix-like work environment.

1.4.1. The login shell

You can select your login shell with chsh(1).

Table 1.13. List of shell programs

package	popcon	size	POSIX shell	description
bash	V:845, I:999	5363	Yes	Bash: the GNU Bourne Again SHell (de facto standard)
tcsh	V:20, I:71	1382	No	TENEX C Shell: an enhanced version of Berkeley csh
dash	V:869, I:951	222	Yes	Debian Almquist Shell, good for shell script
zsh	V:30, I:62	1890	Yes	Z shell: the standard shell with many enhancements
pdksh	V:1, I:9	44	Yes	public domain version of the Korn shell
csh	V:3, I:14	307	No	OpenBSD C Shell, a version of Berkeley csh
sash	V:2, I:8	998	Yes	Stand-alone shell with builtin commands (Not meant for standard "**/bin/sh**")
ksh	V:5, I:25	3145	Yes	the real, AT&T version of the Korn shell
rc	V:0, I:8	176	No	implementation of the AT&T Plan 9 rc shell
posh	V:0, I:0	201	Yes	Policy-compliant Ordinary SHell (**pdksh** derivative)

> **Tip**
>
> Although POSIX-like shells share the basic syntax, they can differ in behavior for things as basic as shell variables and glob expansions. Please check their documentation for details.

In this tutorial chapter, the interactive shell always means **bash**.

1.4.2. Customizing bash

You can customize bash(1) behavior by "**~/.bashrc**".

For example, try the following.

```
# CD upon exiting MC
. /usr/lib/mc/mc.sh

# set CDPATH to a good one
CDPATH=.:/usr/share/doc:~:~/Desktop:~
export CDPATH

PATH="${PATH}":/usr/sbin:/sbin
# set PATH so it includes user's private bin if it exists
if [ -d ~/bin ] ; then
  PATH=~/bin:"${PATH}"
fi
export PATH

EDITOR=vim
export EDITOR
```

Tip

You can find more **bash** customization tips, such as Section 9.2.7, “Colorized commands”, in Chapter 9, *System tips*.

1.4.3. Special key strokes

In the Unix-like environment, there are few key strokes which have special meanings. Please note that on a normal Linux character console, only the left-hand **Ctrl** and **Alt** keys work as expected. Here are few notable key strokes to remember.

Table 1.14. List of key bindings for bash

key	description of key binding
Ctrl-U	erase line before cursor
Ctrl-H	erase a character before cursor
Ctrl-D	terminate input (exit shell if you are using shell)
Ctrl-C	terminate a running program
Ctrl-Z	temporarily stop program by moving it to the background job
Ctrl-S	halt output to screen
Ctrl-Q	reactivate output to screen
Ctrl-Alt-Del	reboot/halt the system, see inittab(5)
Left-Alt-key (optionally, **Windows-key**)	meta-key for Emacs and the similar UI
Up-arrow	start command history search under **bash**
Ctrl-R	start incremental command history search under **bash**

key	description of key binding
Tab	complete input of the filename to the command line under **bash**
Ctrl-V Tab	input **Tab** without expansion to the command line under **bash**

Tip

The terminal feature of **Ctrl-S** can be disabled using stty(1).

1.4.4. Unix style mouse operations

Unix style mouse operations are based on the 3 button mouse system.

Table 1.15. List of Unix style mouse operations

action	response
Left-click-and-drag mouse	select and copy to the clipboard
Left-click	select the start of selection
Right-click	select the end of selection and copy to the clipboard
Middle-click	paste clipboard at the cursor

The center wheel on the modern wheel mouse is considered middle mouse button and can be used for middle-click. Clicking left and right mouse buttons together serves as the middle-click under the 2 button mouse system situation. In order to use a mouse in Linux character consoles, you need to have gpm(8) running as daemon.

1.4.5. The pager

The less(1) command is the enhanced pager (file content browser). It reads the file specified by its command argument or its standard input. Hit "**h**" if you need help while browsing with the **less** command. It can do much more than more(1) and can be supercharged by executing "**eval $(lesspipe)**" or "**eval $(lessfile)**" in the shell startup script. See more in "**/usr/share/doc/lessf/LESSOPEN**". The "**-R**" option allows raw character output and enables ANSI color escape sequences. See less(1).

1.4.6. The text editor

You should become proficient in one of variants of Vim or Emacs programs which are popular in the Unix-like system.

I think getting used to Vim commands is the right thing to do, since Vi-editor is always there in the Linux/Unix world. (Actually, original **vi** or new **nvi** are programs you find everywhere. I chose Vim instead for newbie

since it offers you help through **F1** key while it is similar enough and more powerful.)

If you chose either Emacs or XEmacs instead as your choice of the editor, that is another good choice indeed, particularly for programming. Emacs has a plethora of other features as well, including functioning as a newsreader, directory editor, mail program, etc. When used for programming or editing shell scripts, it intelligently recognizes the format of what you are working on, and tries to provide assistance. Some people maintain that the only program they need on Linux is Emacs. Ten minutes learning Emacs now can save hours later. Having the GNU Emacs manual for reference when learning Emacs is highly recommended.

All these programs usually come with tutoring program for you to learn them by practice. Start Vim by typing "**vim**" and press F1-key. You should at least read the first 35 lines. Then do the online training course by moving cursor to "**|tutor|**" and pressing **Ctrl-]**.

Note

Good editors, such as Vim and Emacs, can handle UTF-8 and other exotic encoding texts correctly. It is a good idea to use the X environment in the UTF-8 locale and to install required programs and fonts to it. Editors have options to set the file encoding independent of the X environment. Please refer to their documentation on multibyte text.

1.4.7. Setting a default text editor

Debian comes with a number of different editors. We recommend to install the **vim** package, as mentioned above.

Debian provides unified access to the system default editor via command "**/usr/bin/editor**" so other programs (e.g., reportbug(1)) can invoke it. You can change it by the following.

```
$ sudo update-alternatives --config editor
```

The choice "**/usr/bin/vim.basic**" over "**/usr/bin/vim.tiny**" is my recommendation for newbies since it supports syntax highlighting.

Tip

Many programs use the environment variables "**$EDITOR**" or "**$VISUAL**" to decide which editor to use (see Section 1.3.5, “The internal editor in MC” and Section 9.3.11, “Customizing program to be started”). For the consistency on the Debian system, set these to "**/usr/bin/editor**". (Historically, "**$EDITOR**" was "**ed**" and "**$VISUAL**" was "**vi**".)

1.4.8. Customizing vim

You can customize vim(1) behavior by "**~/.vimrc**".

For example, try the following

```
" -----------------------------
" Local configuration
"
set nocompatible
set nopaste
set pastetoggle=<f2>
syn on
if $USER == "root"
 set nomodeline
 set noswapfile
else
 set modeline
 set swapfile
endif
" filler to avoid the line above being recognized as a
modeline
" filler
" filler
```

1.4.9. Recording the shell activities

The output of the shell command may roll off your screen and may be lost forever. It is a good practice to log shell activities into the file for you to review them later. This kind of record is essential when you perform any system administration tasks.

The basic method of recording the shell activity is to run it under script(1).

For example, try the following

```
$ script
Script started, file is typescript
```

Do whatever shell commands under **script**.

Press **Ctrl-D** to exit **script**.

```
$ vim typescript
```

See Section 9.2.3, “Recording the shell activities cleanly” .

1.4.10. Basic Unix commands

Let's learn basic Unix commands. Here I use "Unix" in its generic sense. Any Unix clone OSs usually offer equivalent commands. The Debian system is no exception. Do not worry if some commands do not work as you wish now. If **alias** is used in the shell, its corresponding command outputs are

different. These examples are not meant to be executed in this order.

Try all following commands from the non-privileged user account.

Table 1.16. List of basic Unix commands

command	description
pwd	display name of current/working directory
whoami	display current user name
id	display current user identity (name, uid, gid, and associated groups)
file <foo>	display a type of file for the file "**<foo>**"
type -p <commandname>	display a file location of command "**<commandname>**"
which <commandname>	' '
type <commandname>	display information on command "**<commandname>**"
apropos <key-word>	find commands related to "**<key-word>**"
man -k <key-word>	' '
whatis <commandname>	display one line explanation on command "**<commandname>**"
man -a <commandname>	display explanation on command "**<commandname>**" (Unix style)
info <commandname>	display rather long explanation on command "**<commandname>**" (GNU style)
ls	list contents of directory (non-dot files and directories)
ls -a	list contents of directory (all files and directories)
ls -A	list contents of directory (almost all files and directories, i.e., skip "**..**" and "**.**")
ls -la	list all contents of directory with detail information
ls -lai	list all contents of directory with inode number and detail information
ls -d	list all directories under the current directory
tree	display file tree contents
lsof <foo>	list open status of file "**<foo>**"
lsof -p <pid>	list files opened by the process ID: "**<pid>**"
mkdir <foo>	make a new directory "**<foo>**" in the current directory
rmdir <foo>	remove a directory "**<foo>**" in the current directory

command	description
cd <foo>	change directory to the directory "**<foo>**" in the current directory or in the directory listed in the variable "**$CDPATH**"
cd /	change directory to the root directory
cd	change directory to the current user's home directory
cd /<foo>	change directory to the absolute path directory "**/<foo>**"
cd ..	change directory to the parent directory
cd ~<foo>	change directory to the home directory of the user "**<foo>**"
cd -	change directory to the previous directory
</etc/motd pager	display contents of "**/etc/motd**" using the default pager
touch <junkfile>	create a empty file "**<junkfile>**"
cp <foo> <bar>	copy a existing file "**<foo>**" to a new file "**<bar>**"
rm <junkfile>	remove a file "**<junkfile>**"
mv <foo> <bar>	rename an existing file "**<foo>**" to a new name "**<bar>**" ("**<bar>**" must not exist)
mv <foo> <bar>	move an existing file "**<foo>**" to a new location "**<bar>/<foo>**" (the directory "**<bar>**" must exist)
mv <foo> <bar>/<baz>	move an existing file "**<foo>**" to a new location with a new name "**<bar>/<baz>**" (the directory "**<bar>**" must exist but the directory "**<bar>/<baz>**" must not exist)
chmod 600 <foo>	make an existing file "**<foo>**" to be non-readable and non-writable by the other people (non-executable for all)
chmod 644 <foo>	make an existing file "**<foo>**" to be readable but non-writable by the other people (non-executable for all)
chmod 755 <foo>	make an existing file "**<foo>**" to be readable but non-writable by the other people (executable for all)
find . -name <pattern>	find matching filenames using shell "**<pattern>**" (slower)
locate -d . <pattern>	find matching filenames using shell "**<pattern>**" (quicker using regularly generated database)
grep -e "<pattern>" *.html	find a "**<pattern>**" in all files ending with "**.html**" in current directory and display them all
top	display process information using full screen, type "**q**" to quit

command	description
ps aux | pager	display information on all the running processes using BSD style output
ps -ef | pager	display information on all the running processes using Unix system-V style output
ps aux | grep -e "[e]xim4*"	display all processes running "**exim**" and "**exim4**"
ps axf | pager	display information on all the running processes with ASCII art output
kill <1234>	kill a process identified by the process ID: "<1234>"
gzip <foo>	compress "**<foo>**" to create "**<foo>.gz**" using the Lempel-Ziv coding (LZ77)
gunzip <foo>.gz	decompress "**<foo>.gz**" to create "**<foo>**"
bzip2 <foo>	compress "**<foo>**" to create "**<foo>.bz2**" using the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding (better compression than **gzip**)
bunzip2 <foo>.bz2	decompress "**<foo>.bz2**" to create "**<foo>**"
xz <foo>	compress "**<foo>**" to create "**<foo>.xz**" using the Lempel–Ziv–Markov chain algorithm (better compression than **bzip2**)
unxz <foo>.xz	decompress "**<foo>.xz**" to create "**<foo>**"
tar -xvf <foo>.tar	extract files from "**<foo>.tar**" archive
tar -xvzf <foo>.tar.gz	extract files from gzipped "**<foo>.tar.gz**" archive
tar -xvjf <foo>.tar.bz2	extract files from "**<foo>.tar.bz2**" archive
tar -xvJf <foo>.tar.xz	extract files from "**<foo>.tar.xz**" archive
tar -cvf <foo>.tar <bar>/	archive contents of folder "**<bar>/**" in "**<foo>.tar**" archive
tar -cvzf <foo>.tar.gz <bar>/	archive contents of folder "**<bar>/**" in compressed "**<foo>.tar.gz**" archive
tar -cvjf <foo>.tar.bz2 <bar>/	archive contents of folder "**<bar>/**" in "**<foo>.tar.bz2**" archive
tar -cvJf <foo>.tar.xz <bar>/	archive contents of folder "**<bar>/**" in "**<foo>.tar.xz**" archive
zcat README.gz | pager	display contents of compressed "**README.gz**" using the default pager

command	description
zcat README.gz > foo	create a file "**foo**" with the decompressed content of "**README.gz**"
zcat README.gz >> foo	append the decompressed content of "**README.gz**" to the end of the file "**foo**" (if it does not exist, create it first)

Note

Unix has a tradition to hide filenames which start with "**.**". They are traditionally files that contain configuration information and user preferences.

Note

For **cd** command, see builtins(7).

Note

The default pager of the bare bone Debian system is more(1) which cannot scroll back. By installing the **less** package using command line "**apt-get install less**", less(1) becomes default pager and you can scroll back with cursor keys.

Note

The "**[**" and "**]**" in the regular expression of the "**ps aux | grep -e "[e]xim4*"**" command above enable **grep** to avoid matching itself. The "**4***" in the regular expression means 0 or more repeats of character "**4**" thus enables **grep** to match both "**exim**" and "**exim4**". Although "*" is used in the shell filename glob and the regular expression, their meanings are different. Learn the regular expression from grep(1).

Please traverse directories and peek into the system using the above commands as training. If you have questions on any of console commands, please make sure to read the manual page.

For example, try the following

```
$ man man
$ man bash
$ man builtins
$ man grep
$ man ls
```

The style of man pages may be a little hard to get used to, because they are rather terse, particularly the older, very traditional ones. But once you

get used to it, you come to appreciate their succinctness.

Please note that many Unix-like commands including ones from GNU and BSD display brief help information if you invoke them in one of the following ways (or without any arguments in some cases).

```
$ <commandname> --help
$ <commandname> -h
```

1.5. The simple shell command

Now you have some feel on how to use the Debian system. Let's look deep into the mechanism of the command execution in the Debian system. Here, I have simplified reality for the newbie. See bash(1) for the exact explanation.

A simple command is a sequence of components.

1. Variable assignments (optional)
2. Command name
3. Arguments (optional)
4. Redirections (optional: **>** , **>>** , **<** , **<<** , etc.)
5. Control operator (optional: **&&** , **||** , <newline> , **;** , **&** , **(** , **)**)

1.5.1. Command execution and environment variable

The values of some environment variables change the behavior of some Unix commands.

Default values of environment variables are initially set by the PAM system and then some of them may be reset by some application programs.

- The display manager such as **gdm3** resets environment variables.
- The shell in its start up codes resets environment variables in "**~/.bash_profile**" and "**~/.bashrc**".

1.5.2. The "$LANG" variable

The full locale value given to "**$LANG**" variable consists of 3 parts: "**xx_YY.ZZZZ**".

Table 1.17. The 3 parts of locale value

locale value	meaning
xx	ISO 639 language codes (lower case) such as "en"
YY	ISO 3166 country codes (upper case) such as "US"
ZZZZ	codeset, always set to "UTF-8"

For language codes and country codes, see pertinent description in the "**info gettext**".

For the codeset on the modern Debian system, you should always set it to **UTF-8** unless you specifically want to use the historic one with good reason and background knowledge.

For fine details of the locale configuration, see Section 8.3, “The locale”.

Note

The "**LANG=en_US**" is not "**LANG=C**" nor "**LANG=en_US.UTF-8**". It is "**LANG=en_US.ISO-8859-1**" (see Section 8.3.1, “Basics of encoding”).

Table 1.18. List of locale recommendations

locale recommendation	Language (area)
en_US.UTF-8	English (USA)
en_GB.UTF-8	English (Great Britain)
fr_FR.UTF-8	French (France)
de_DE.UTF-8	German (Germany)
it_IT.UTF-8	Italian (Italy)
es_ES.UTF-8	Spanish (Spain)
ca_ES.UTF-8	Catalan (Spain)
sv_SE.UTF-8	Swedish (Sweden)
pt_BR.UTF-8	Portuguese (Brazil)
ru_RU.UTF-8	Russian (Russia)
zh_CN.UTF-8	Chinese (P.R. of China)
zh_TW.UTF-8	Chinese (Taiwan R.O.C.)
ja_JP.UTF-8	Japanese (Japan)
ko_KR.UTF-8	Korean (Republic of Korea)
vi_VN.UTF-8	Vietnamese (Vietnam)

Typical command execution uses a shell line sequence as the following.

```
$ date
Sun Jun  3 10:27:39 JST 2007
$ LANG=fr_FR.UTF-8 date
dimanche 3 juin 2007, 10:27:33 (UTC+0900)
```

Here, the program date(1) is executed with different values of the environment variable "**$LANG**".

-

For the first command, "**$LANG**" is set to the system default locale value "**en_US.UTF-8**".

- For the second command, "**$LANG**" is set to the French UTF-8 locale value "**fr_FR.UTF-8**".

Most command executions usually do not have preceding environment variable definition. For the above example, you can alternatively execute as the following.

```
$ LANG=fr_FR.UTF-8
$ date
dimanche 3 juin 2007, 10:27:33 (UTC+0900)
```

As you can see here, the output of command is affected by the environment variable to produce French output. If you want the environment variable to be inherited to subprocesses (e.g., when calling shell script), you need to **export** it instead by the following.

```
$ export LANG
```

Note

When you use a typical console terminal, the "**$LANG**" environment variable is usually set to be **exported** by the desktop environment. So the above is not really a good example to test the effect of **export**.

Tip

When filing a bug report, running and checking the command under "**LANG=en_US.UTF-8**" is a good idea if you use non-English environment.

See locale(5) and locale(7) for "**$LANG**" and related environment variables.

Note

I recommend you to configure the system environment just by the "**$LANG**" variable and to stay away from "**$LC_***" variables unless it is absolutely needed.

1.5.3. The "$PATH" variable

When you type a command into the shell, the shell searches the command in the list of directories contained in the "**$PATH**" environment variable. The value of the "**$PATH**" environment variable is also called the shell's search path.

In the default Debian installation, the "**$PATH**" environment variable of

user accounts may not include "**/sbin**" and "**/usr/sbin**". For example, the **ifconfig** command needs to be issued with full path as "**/sbin/ifconfig**". (Similar **ip** command is located in "**/bin**".)

You can change the "**$PATH**" environment variable of Bash shell by "**~/.bash_profile**" or "**~/.bashrc**" files.

1.5.4. The "$HOME" variable

Many commands stores user specific configuration in the home directory and changes their behavior by their contents. The home directory is identified by the environment variable "**$HOME**".

Table 1.19. List of "$HOME" values

value of "$HOME"	program execution situation
/	program run by the init process (daemon)
/root	program run from the normal root shell
/home/<normal_user>	program run from the normal user shell
/home/<normal_user>	program run from the normal user GUI desktop menu
/home/<normal_user>	program run as root with "**sudo program**"
/root	program run as root with "**sudo -H program**"

Tip

Shell expands "**~/**" to current user's home directory, i.e., "**$HOME/**". Shell expands "**~foo/**" to **foo**'s home directory, i.e., "**/home/foo/**".

1.5.5. Command line options

Some commands take arguments. Arguments starting with "**-**" or "**--**" are called options and control the behavior of the command.

```
$ date
Mon Oct 27 23:02:09 CET 2003
$ date -R
Mon, 27 Oct 2003 23:02:40 +0100
```

Here the command-line argument "**-R**" changes date(1) behavior to output RFC2822 compliant date string.

1.5.6. Shell glob

Often you want a command to work with a group of files without typing all of them. The filename expansion pattern using the shell **glob**, (sometimes referred as **wildcards**), facilitate this need.

Table 1.20. Shell glob patterns

shell glob pattern	description of match rule
*****	filename (segment) not started with "**.**"
.*	filename (segment) started with "**.**"
?	exactly one character
[…]	exactly one character with any character enclosed in brackets
[a-z]	exactly one character with any character between "**a**" and "**z**"
[^…]	exactly one character other than any character enclosed in brackets (excluding "**^**")

For example, try the following

```
$ mkdir junk; cd junk; touch 1.txt 2.txt 3.c 4.h .5.txt
..6.txt
$ echo *.txt
1.txt 2.txt
$ echo *
1.txt 2.txt 3.c 4.h
$ echo *.[hc]
3.c 4.h
$ echo .*
. .. .5.txt ..6.txt
$ echo .*[^.]*
.5.txt ..6.txt
$ echo [^1-3]*
4.h
$ cd ..; rm -rf junk
```

See glob(7).

Note

Unlike normal filename expansion by the shell, the shell pattern "*****" tested in find(1) with "**-name**" test etc., matches the initial "**.**" of the filename. (New POSIX feature)

Note

BASH can be tweaked to change its glob behavior with its shopt builtin options such as "**dotglob**", "**noglob**", "**nocaseglob**", "**nullglob**", "**extglob**", etc. See bash(1).

1.5.7. Return value of the command

Each command returns its exit status (variable: "**$?**") as the return value.

Table 1.21. Command exit codes

command exit status	numeric return value	logical return value
success	zero, 0	**TRUE**
error	non-zero, -1	**FALSE**

For example, try the following.

```
$ [ 1 = 1 ] ; echo $?
0
$ [ 1 = 2 ] ; echo $?
1
```

Note

Please note that, in the logical context for the shell, **success** is treated as the logical **TRUE** which has 0 (zero) as its value. This is somewhat non-intuitive and needs to be reminded here.

1.5.8. Typical command sequences and shell redirection

Let's try to remember following shell command idioms typed in one line as a part of shell command.

Table 1.22. Shell command idioms

command idiom	description
command &	**background** execution of **command** in the subshell
command1 | command2	**pipe** the standard output of **command1** to the standard input of **command2** (**concurrent** execution)
command1 2>&1 | command2	**pipe** both standard output and standard error of **command1** to the standard input of **command2** (**concurrent** execution)
command1 ; command2	execute **command1** and **command2 sequentially**
command1 && command2	execute **command1**; if successful, execute **command2 sequentially** (return success if both **command1 and command2** are successful)
command1 || command2	execute **command1**; if not successful, execute **command2 sequentially** (return success if **command1 or command2** are successful)
command > foo	redirect standard output of **command** to a file **foo** (overwrite)

command idiom	description
command 2> foo	redirect standard error of **command** to a file **foo** (overwrite)
command >> foo	redirect standard output of **command** to a file **foo** (append)
command 2>> foo	redirect standard error of **command** to a file **foo** (append)
command > foo 2>&1	redirect both standard output and standard error of **command** to a file **foo**
command < foo	redirect standard input of **command** to a file **foo**
command << delimiter	redirect standard input of **command** to the following lines until "**delimiter**" is met (here document)
command <<- delimiter	redirect standard input of **command** to the following lines until "**delimiter**" is met (here document, the leading tab characters are stripped from input lines)

The Debian system is a multi-tasking system. Background jobs allow users to run multiple programs in a single shell. The management of the background process involves the shell builtins: **jobs**, **fg**, **bg**, and **kill**. Please read sections of bash(1) under "SIGNALS", and "JOB CONTROL", and builtins(1).

For example, try the following

```
$ </etc/motd pager
```

```
$ pager </etc/motd
```

```
$ pager /etc/motd
```

```
$ cat /etc/motd | pager
```

Although all 4 examples of shell redirections display the same thing, the last example runs an extra **cat** command and wastes resources with no reason.

The shell allows you to open files using the **exec** builtin with an arbitrary file descriptor.

```
$ echo Hello >foo
$ exec 3<foo 4>bar  # open files
$ cat <&3 >&4       # redirect stdin to 3, stdout to 4
$ exec 3<&- 4>&-    # close files
$ cat bar
Hello
```

The file descriptor 0-2 are predefined.

Table 1.23. Predefined file descriptors

device	description	file descriptor
stdin	standard input	0
stdout	standard output	1
stderr	standard error	2

1.5.9. Command alias

You can set an alias for the frequently used command.

For example, try the following

```
$ alias la='ls -la'
```

Now, "**la**" works as a short hand for "**ls -la**" which lists all files in the long listing format.

You can list any existing aliases by **alias** (see bash(1) under "SHELL BUILTIN COMMANDS").

```
$ alias
...
alias la='ls -la'
```

You can identity exact path or identity of the command by **type** (see bash(1) under "SHELL BUILTIN COMMANDS").

For example, try the following

```
$ type ls
ls is hashed (/bin/ls)
$ type la
la is aliased to ls -la
$ type echo
echo is a shell builtin
$ type file
file is /usr/bin/file
```

Here **ls** was recently searched while "**file**" was not, thus "**ls**" is "hashed", i.e., the shell has an internal record for the quick access to the location of the "**ls**" command.

Tip

See Section 9.2.7, “Colorized commands”.

1.6. Unix-like text processing

In Unix-like work environment, text processing is done by piping text through chains of standard text processing tools. This was another crucial Unix innovation.

1.6.1. Unix text tools

There are few standard text processing tools which are used very often on the Unix-like system.

- No regular expression is used:
  - cat(1) concatenates files and outputs the whole content.
  - tac(1) concatenates files and outputs in reverse.
  - cut(1) selects parts of lines and outputs.
  - head(1) outputs the first part of files.
  - tail(1) outputs the last part of files.
  - sort(1) sorts lines of text files.
  - uniq(1) removes duplicate lines from a sorted file.
  - tr(1) translates or deletes characters.
  - diff(1) compares files line by line.
- Basic regular expression (**BRE**) is used:
  - grep(1) matches text with patterns.
  - ed(1) is a primitive line editor.
  - sed(1) is a stream editor.
  - vim(1) is a screen editor.
  - emacs(1) is a screen editor. (somewhat extended **BRE**)
- Extended regular expression (**ERE**) is used:
  - egrep(1) matches text with patterns.
  - awk(1) does simple text processing.
  - tcl(3tcl) can do every conceivable text processing: See re_syntax(3). Often used with tk(3tk).
  - perl(1) can do every conceivable text processing. See perlre(1).
  - pcregrep(1) from the **pcregrep** package matches text with Perl Compatible Regular Expressions (PCRE) pattern.

- python(1) with the **re** module can do every conceivable text processing. See "**/usr/share/doc/python/html/index.html**".

If you are not sure what exactly these commands do, please use "**man command**" to figure it out by yourself.

Note

Sort order and range expression are locale dependent. If you wish to obtain traditional behavior for a command, use **C** locale instead of **UTF-8** ones by prepending command with "**LANG=C**" (see Section 1.5.2, “The "**$LANG**" variable” and Section 8.3, “The locale”).

Note

Perl regular expressions (perlre(1)), Perl Compatible Regular Expressions (PCRE), and Python regular expressions offered by the **re** module have many common extensions to the normal **ERE**.

1.6.2. Regular expressions

Regular expressions are used in many text processing tools. They are analogous to the shell globs, but they are more complicated and powerful.

The regular expression describes the matching pattern and is made up of text characters and **metacharacters**.

A **metacharacter** is just a character with a special meaning. There are 2 major styles, **BRE** and **ERE**, depending on the text tools as described above.

Table 1.24. Metacharacters for BRE and ERE

BRE	ERE	description of the regular expression
`\ . [ ] ^ $ *`	`\ . [ ] ^ $ *`	common **metacharacters**
`\+ \? \( \) \{ \} |`		BRE only "\" escaped **metacharacters**
	`+ ? ( ) { } |`	ERE only non-"\" escaped **metacharacters**
c	**c**	match **non-metacharacter** "**c**"
\c	**\c**	match a literal character "**c**" even if "**c**" is **metacharacter** by itself
.	**.**	match any character including newline
^	**^**	position at the beginning of a string
$	**$**	position at the end of a string
\<	**\<**	position at the beginning of a word

BRE	ERE	description of the regular expression
\>	**\>**	position at the end of a word
[abc…]	**[abc…]**	match any characters in "**abc…**"
[^abc…]	**[^abc…]**	match any characters except in "**abc…**"
r*	**r***	match zero or more regular expressions identified by "**r**"
r\+	**r+**	match one or more regular expressions identified by "**r**"
r\?	**r?**	match zero or one regular expressions identified by "**r**"
r1|r2	**r1|r2**	match one of the regular expressions identified by "**r1**" or "**r2**"
\(r1|r2\)	**(r1|r2)**	match one of the regular expressions identified by "**r1**" or "**r2**" and treat it as a **bracketed** regular expression

The regular expression of **emacs** is basically **BRE** but has been extended to treat "**+**"and "**?**" as the **metacharacters** as in **ERE**. Thus, there are no needs to escape them with "\" in the regular expression of **emacs**.

grep(1) can be used to perform the text search using the regular expression.

For example, try the following

```
$ egrep 'GNU.*LICENSE|Yoyodyne' /usr/share/common-licenses
/GPL
GNU GENERAL PUBLIC LICENSE
GNU GENERAL PUBLIC LICENSE
Yoyodyne, Inc., hereby disclaims all copyright interest in
the program
```

Tip

See Section 9.2.7, “Colorized commands”.

1.6.3. Replacement expressions

For the replacement expression, some characters have special meanings.

Table 1.25. The replacement expression

replacement expression	description of the text to replace the replacement expression
&	what the regular expression matched (use **\&** in **emacs**)
\n	what the n-th **bracketed** regular expression matched ("n" being number)

For Perl replacement string, "**$n**" is used instead of "**\n**" and "**&**" has no special meaning.

For example, try the following

```
$ echo zzz1abc2efg3hij4 | \
sed -e 's/\(1[a-z]*\)[0-9]*\(.*\)$/=&=/'
zzz=1abc2efg3hij4=
$ echo zzz1abc2efg3hij4 | \
sed -e 's/\(1[a-z]*\)[0-9]*\(.*\)$/\2===\1/'
zzzefg3hij4===1abc
$ echo zzz1abc2efg3hij4 | \
perl -pe 's/(1[a-z]*)[0-9]*(.*)$/$2===$1/'
zzzefg3hij4===1abc
$ echo zzz1abc2efg3hij4 | \
perl -pe 's/(1[a-z]*)[0-9]*(.*)$/=&=/'
zzz=&=
```

Here please pay extra attention to the style of the **bracketed** regular expression and how the matched strings are used in the text replacement process on different tools.

These regular expressions can be used for cursor movements and text replacement actions in some editors too.

The back slash "\" at the end of line in the shell commandline escapes newline as a white space character and continues shell command line input to the next line.

Please read all the related manual pages to learn these commands.

1.6.4. Global substitution with regular expressions

The ed(1) command can replace all instances of "**FROM_REGEX**" with "**TO_TEXT**" in "**file**".

```
$ ed file <<EOF
,s/FROM_REGEX/TO_TEXT/g
w
q
EOF
```

The sed(1) command can replace all instances of "**FROM_REGEX**" with "**TO_TEXT**" in "**file**".

```
$ sed -ie 's/FROM_REGEX/TO_TEXT/g' file
```

The vim(1) command can replace all instances of "**FROM_REGEX**" with "**TO_TEXT**" in "**file**" by using ex(1) commands.

```
$ vim '+%s/FROM_REGEX/TO_TEXT/gc' '+w' '+q' file
```

Tip

The "**c**" flag in the above ensures interactive confirmation for each substitution.

Multiple files ("**file1**", "**file2**", and "**file3**") can be processed with regular expressions similarly with vim(1) or perl(1).

```
$ vim '+argdo %s/FROM_REGEX/TO_TEXT/ge|update' '+q' file1
file2 file3
```

Tip

The "**e**" flag in the above prevents the "No match" error from breaking a mapping.

```
$ perl -i -p -e 's/FROM_REGEX/TO_TEXT/g;' file1 file2 file3
```

In the perl(1) example, "**-i**" is for the in-place editing of each target file, and "**-p**" is for the implicit loop over all given files.

Tip

Use of argument "**-i.bak**" instead of "**-i**" keeps each original file by adding "**.bak**" to its filename. This makes recovery from errors easier for complex substitutions.

Note

ed(1) and vim(1) are **BRE**; perl(1) is **ERE**.

1.6.5. Extracting data from text file table

Let's consider a text file called "**DPL**" in which some pre-2004 Debian project leader's names and their initiation date are listed in a space-separated format.

```
Ian     Murdock   August  1993
Bruce   Perens    April   1996
Ian     Jackson   January 1998
Wichert Akkerman  January 1999
Ben     Collins   April   2001
Bdale   Garbee    April   2002
Martin  Michlmayr March   2003
```

Tip

See "A Brief History of Debian" for the latest Debian leadership history.

Awk is frequently used to extract data from these types of files.

For example, try the following

```
$ awk '{ print $3 }' <DPL                    # month started
August
April
January
January
April
April
March
$ awk '($1=="Ian") { print }' <DPL           # DPL called Ian
Ian     Murdock   August  1993
Ian     Jackson   January 1998
$ awk '($2=="Perens") { print $3,$4 }' <DPL # When Perens
started
April 1996
```

Shells such as Bash can be also used to parse this kind of file.

For example, try the following

```
$ while read first last month year; do
    echo $month
  done <DPL
... same output as the first Awk example
```

Here, the **read** builtin command uses characters in "**$IFS**" (internal field separators) to split lines into words.

If you change "**$IFS**" to "**:**", you can parse "**/etc/passwd**" with shell nicely.

```
$ oldIFS="$IFS"   # save old value
$ IFS=':'
$ while read user password uid gid rest_of_line; do
    if [ "$user" = "bozo" ]; then
      echo "$user's ID is $uid"
    fi
  done < /etc/passwd
bozo's ID is 1000
$ IFS="$oldIFS"   # restore old value
```

(If Awk is used to do the equivalent, use "**FS=':'**" to set the field separator.)

IFS is also used by the shell to split results of parameter expansion, command substitution, and arithmetic expansion. These do not occur within double or single quoted words. The default value of IFS is <space>, <tab>, and <newline> combined.

Be careful about using this shell IFS tricks. Strange things may happen, when shell interprets some parts of the script as its **input**.

```
$ IFS=":,"                          # use ":" and "," as IFS
$ echo IFS=$IFS,   IFS="$IFS"       # echo is a Bash builtin
IFS=  , IFS=:,
$ date -R                           # just a command output
Sat, 23 Aug 2003 08:30:15 +0200
$ echo $(date -R)                   # sub shell --> input to
main shell
Sat  23 Aug 2003 08 30 36 +0200
$ unset IFS                         # reset IFS to the default
$ echo $(date -R)
Sat, 23 Aug 2003 08:30:50 +0200
```

1.6.6. Script snippets for piping commands

The following scripts do nice things as a part of a pipe.

Table 1.26. List of script snippets for piping commands

script snippet (type in one line)	effect of command
`find /usr -print`	find all files under "`/usr`"
`seq 1 100`	print 1 to 100
`| xargs -n 1 <command>`	run command repeatedly with each item from pipe as its argument
`| xargs -n 1 echo`	split white-space-separated items from pipe into lines
`| xargs echo`	merge all lines from pipe into a line
`| grep -e <regex_pattern>`	extract lines from pipe containing <regex_pattern>
`| grep -v -e <regex_pattern>`	extract lines from pipe not containing <regex_pattern>
`| cut -d: -f3 -`	extract third field from pipe separated by "`:`" (passwd file etc.)
`| awk '{ print $3 }'`	extract third field from pipe separated by whitespaces
`| awk -F'\t' '{ print $3 }'`	extract third field from pipe separated by tab
`| col -bx`	remove backspace and expand tabs to spaces
`| expand -`	expand tabs
`| sort| uniq`	sort and remove duplicates

script snippet (type in one line)	effect of command
`| tr 'A-Z' 'a-z'`	convert uppercase to lowercase
`| tr -d '\n'`	concatenate lines into one line
`| tr -d '\r'`	remove CR
`| sed 's/^/# /'`	add "#" to the start of each line
`| sed 's/\.ext//g'`	remove "`.ext`"
`| sed -n -e 2p`	print the second line
`| head -n 2 -`	print the first 2 lines
`| tail -n 2 -`	print the last 2 lines

A one-line shell script can loop over many files using find(1) and xargs(1) to perform quite complicated tasks. See Section 10.1.5, “Idioms for the selection of files” and Section 9.3.9, “Repeating a command looping over files”.

When using the shell interactive mode becomes too complicated, please consider to write a shell script (see Section 12.1, “The shell script”).