Efficient use of the Linux command line in the Bash shell

Marc van der Sluys

Nikhef/GRASP, Utrecht University
The Netherlands

September 2, 2023

http://eubs.sf.net
http://pub.vandersluys.nl

This document was typeset in LaTeX by the author.
The official GNU Bash logo was created by the Free Software Foundation.

Contents

1 Introduction

Using a shell or 'the command line', as opposed to a graphical program, can be extremely fast, albeit Spartan.[1] The drawback is that you will have to know very well what you are doing (which of course adds to the speed). I also notice that a key combination is often pressed by my fingers before my brain is aware what is going on.[2] To me, using a GUI rather than the command line is often like writing an email by selecting the words from a dictionary rather than typing them using the alphabet and keyboard. Also, I feel switching from keyboard to mouse or *vice versa* forms a significant overhead, and the command line allows you to stick to the keyboard in most cases.

In addition, I often have to repeat a set of commands. For example, I might need to select certain lines from a file (using `grep`), make some changes in those lines (`sed`), convert them to a different format (`awk`) and save them in a file. And then I need to do the same four things to another 50 files. The advantage of shell commands is that you can copy

script them into a text file, make the file executable, and you have created a *script*, a lightweight program, that does the job for you. Add a `for` loop over all files, and your script replaces those 200 commands with only one.

shell When using the command line, one uses an *interactive shell*, which forms the layer or interface between the user and the operating system. Commands given in the command-line shell will be interpreted by the system and carried out if possible. A default shell on

bash many systems is the Bourne Again SHell, or *bash* [1].[3] While the use of a command-line shell may be inherently faster than using a GUI, the use of bash itself can be sped up significantly as well, since in many cases commands or file names do not have to be typed out in full, but can be completed semiautomatically, or reused. Many shortcuts exist to perform these actions quickly. These are not bash per se, but many are provided by the

readline `readline` [2] package (by the same authors) — see `man readline` for more information. The tips and tricks in this document can improve your experience when using the bash command-line shell dramatically.

2 Directories

2.1 Moving around in the directory tree

cd Use the command `cd` (**change directory**) to move around the directories in your file system. Note that the directory name may or may not have a trailing forward slash. They are equivalent: `cd dir` does the same as `cd dir/`. The slash denotes the separation between directory names, and comes in handy when you cd into multiple directories at once: `cd dir1/dir2`. By issuing `cd` without an argument, you will jump to your **home directory** (/home/<user> or ~).

cd .. The command `cd ..` moves you to the **parent directory** of the current directory, often called going 'up' one directory (although this moves you one step closer to the **root** directory (/) in your directory *tree*, which suggests it ought to be called 'down' in this analogy). You can combine this with other cd commands, for example by moving 'up' from the current directory into the parent directory and from there into another subdirectory using *e.g.* `cd ../dir1`.

cd - `cd -` returns to the **previously visited** directory. Issuing the command repeatedly without any other cd call will have you jump back and forth between the last two directories.

If you want to specify an **absolute path** to cd into, you specify it from the **root directory**

cd / (/), *e.g.* `cd /usr/bin/`. If my root directory contains the two subdirectories dir1 and dir2,

[1] In my experience, 'Spartan' often turns out to mean 'efficient'.

[2] To be fair, this may be an issue with my brain.

[3] For alternative shells, search for `sh, csh, tcsh, fish, ksh, zsh, ...`

and I am currently in dir1, I can cd into dir2 in two ways: using the absolute path: `cd /dir2` and using the **relative path**: `cd ../dir2`.

If the absolute path lies somewhere in your home directory, you can replace the '/home/<user>' part with '~', *e.g.* `cd ~/Documents/`. The command `cd ~` is equivalent to `cd` (without argument). **cd ~**

If you want to see where you can cd into, or whether the current directory contains the file you are looking for, use the `ls` command to **list** the directory contents. See Sect. 3.1 for more details. **ls**

Another way to check which directories are available to change into is to use *Tab completion* by typing `cd` and pressing Space Tab Tab (see Section 6.1). The shell will show the solutions that can be cd-ed into, *i.e.* directories. **Tab**

To see where in your file system you are currently located, issue the command `pwd` (print working directory). This will show the absolute path of your current directory. **pwd**

2.2 Creating and removing directories

To **create** a new directory as a subdirectory of the current directory, you can use the `mkdir` command, *e.g.* `mkdir newdir`. **Removing** a directory, if it is empty, can be done using `rmdir`. In order to remove a directory and all its content (files, subdirectories) recursively, use the `rm` command (with care!): `rm -r olddir/`. **mkdir** **rmdir** **rm -r**

3 Using files

3.1 Managing files

You can **list** the files in the current directory by typing `ls`. To see the properties of those files, like the size, ownership and permissions, use the long format: `ls -l`. The very first character in the line shows whether the file is 'special': d for directory, l for symbolic link, c and b for a character and block device, p for a pipe and s for a socket. The next three blocks of 'rwx' indicate whether the user, group members or others are allowed to read, write and/or execute the file. Note that directories must be executable in order to `cd` into them. See also `chmod` below. If you are only interested in the most recent files, sort them reversely by time using `ls -lrt` and look at the bottom of the list. `ls -a` shows all files, including hidden files starting with a dot. **ls** **ls -l**

More information about the **file type** can be obtained using the `file` command, *e.g.*: `file file1`. **file**

You can **copy** a file elsewhere, *e.g.* into a subdirectory using the `cp` command: `cp file dir/`. This duplicates the existing file, so that you have two independent copies. The target can also be in the current directory, *e.g.* `cp file1 file2`, which results in two copies with different names. **cp**

Moving a file using the `mv` command puts it elsewhere, removing the original. To move file1 'up' one directory, do `mv file1 ../`. You can rename a file by 'moving' into a different name in the current directory: `mv file1 file2`. By default, the `mv` command overwrites the destination file if it exists. Hence, it is a good idea to use `mv -i`, which will prompt you before overwriting. In fact, I have an alias set that makes this the default behaviour (see Sects. 8.2 and 8.4). This behaviour can be overridden by specifying `mv -f` to force overwriting existing files. **mv** **mv -i**

To **remove** a file, use the `rm` command: `rm file`. By default, no confirmation is asked before deleting the file, so be careful. Again, I use the alias `rm -i`, which will ask for confirmation (see Sects. 8.2 and 8.4). Override this with `rm -f`. **rm** **rm -i**

chmod Finally, you can change the **permissions** (read, write execute) of a file using `chmod`, either using mode bits or octal mode. For instance, `chmod u+x file` gives the user (u) permission (+) to execute (x) the file, while `chmod go-rwx file` denies (-) read, write and execute (rwx) permissions to group and others (go). See `man chmod` for more information and `man 2 chmod` for extra details on the octal mode. In a similar way, the owner and
chown group of a file can be changed using `chown`.

3.2 Reading text files

cat You can print the contents of a text file to the screen by issuing the command `cat file`. For large files, this may take some time and make it difficult to scroll back to earlier
head output. If you are interested in the first part of the file, use `head`: the command `head`
tail `-20 file` shows the first 20 lines. The command `tail -30 file` shows the last 30 lines of a file. All three commands and more are included in the command `less` (see the next section).

3.2.1 Using the less program

less To have more control and a clean screen afterwards, use `less file` [3]. This will show you the first screen of text. You can jump screen pages by pressing Space, jump to the
g, G, q beginning or end of the file by pressing g or G respectively, and quit by pressing q.
/ Searching forward is done by pressing a forward slash (/) followed by the search term and
n Enter. Search for the next match by pressing n. Reverse searching can be done using
? the question mark (?), followed by the search term and Enter. Pressing n searches for the previous match *before* the current one. Note that searching is case sensitive by default. You can open the text in your default editor (see the `$EDITOR` variable in Section 8.1) by
v pressing v.

pager The command `less` can also be used as a *pager*, *i.e.* to paginate long texts, as in `ls -l | less`. I have set `less` as my default pager using the `$PAGER` environment variable (see Sect. 8.1).

3.3 Text editors

When selecting a text editor, consider using one that can run in a terminal. The great advantage is that it uses key combinations to perform the mostly used tasks, which makes them fast in use. Another pro is that these editors are easy to use when logged in to a remote machine over a slow connection. Some editors may have a terminal and a graphical mode, which makes them very flexible and useful.

If you'll be using a text editor under Linux (or other UNIX-like system) a lot, for example because you're a student of physics or computer science, it may well be worth your while
vi to look into `emacs` [4] or `vi` [5]. While `vi` is installed on basically all GNU/Linux systems
emacs by default, `emacs` is installed on most and easily installable on virtually all other Linux systems. Both editors can run in a terminal as well as with a GUI (`vi` clone `vim` has a GUI version `gvim`) and are available for a number of other operating systems as well. They support many languages, and are extensible, so that they will probably also support languages that do not yet exist.[4] They have been around for nearly 50 years, and may well be around for another 50. Which of the two you choose is largely a matter of taste. In my opinion, the main difference is that with `vi` you can perform the basic tasks with a few key strokes, while in `emacs` you will need a few more key strokes but can do virtually anything with them. The additional advantage of emacs is that many of these keystrokes can also be used in the bash shell. `vi` is even more Spartan[5] than `emacs` and has a steeper

[4] I use about 20 such languages or modes. I would consider learning a different editor or IDE for each one of them about as cumbersome as learning a different natural language for every person I regularly talk to.

[5] Efficient?

learning curve. Emacs's menu can be accessed with a mouse in the GUI and F10 in the console version. To get started with emacs, see [6].

If you need a text editor in a terminal, and have never used one before, you can try `nano` [7], which is relatively user friendly. The command `nano file.txt` will open the selected file and you can close the editor again by pressing `Ctrl-X` and following the instructions at the bottom of your screen. Examine the status lines for more basic nano commands, or press `Ctrl-G` for help. Nano provides some basic syntax highlighting for *e.g.* Bash, C, Python, Fortran and LaTeX. Nano settings can be found using `man nanorc` and saved in ∼/.nanorc. You can quickly get up to speed with the nano basics using [8]. **nano**

A list of text editors, sorted by type, is provided by Wikipedia [9].

3.4 Bash scripts

The details of bash scripting are beyond the scope of this article, but the basics are simple. A bash script is little else than a text file with executable permission that contains a list of bash commands, which are usually executed in the order in which they occur. The file name is arbitrary, but often has the extension `.sh`. The first line of the script should read `#!/bin/bash`, to indicate that it is to be interpreted by bash. Hence, a simple example of a trivial bash script that prints the current directory, changes to the root directory and shows its contents would look like this:

Listing 1: Script to cd to the root directory and list its contents.

```
#!/bin/bash
pwd
cd /
ls
```

Create it in your favourite text editor (`emacs script.sh`), save it, make it executable (`chmod u+x script.sh`) and execute it (`./script.sh`).

For all the details on bash scripting, using variables, conditional statements, loops, functions and much more, see [10].

4 Users and groups

4.1 Users

In order to use the GNU/Linux system, you need a **user account**. You log in as a user with a user name and password, given to you by the system administrator. You can change the password by issuing the `passwd` command. The system recognises you by your user identity (uid), which is a number (usually 1000 or higher), and unique for each user. You can see your uid by typing `id`. Using your uid, the system can handle the different file permissions (read, write and execute) on a per-user basis, using the `chmod` command (see Sect. 3.1). For example, you could allow yourself to write to a given file while other users are denied that permission. If you would like to temporarily log in as a different user, you can use the command `su`. For instance, `su - joe` would change your identity to the user joe (the `-` is recommended if you want to keep your environment). You can see who you are (as which user you are currently logged in) by issuing the command `whoami`. File ownership can be transferred to a different user using `chown`. **passwd** **id** **chmod** **su** **whoami** **chown**

4.2 Groups

Each user belongs to a group. By default this is often the group *users* or a one-user group with the same name as the user. The purpose of groups is to allow a finer tuning of permissions. For example, on a school server, all teachers could belong to a group called *teachers*, while all students could be part of the group *students*. Directories containing lecture notes could have the read permission set to 'allow' for both groups, but write

permission to the group of teachers only. The directories containing exams, on the other hand, would have no read permission for students. You can check your group issuing `id`. The group ownership of a file can be changed using `chgrp file` (or `chown`). Read, write and execute permissions (rwx) for the user (owner), group and others (not belonging to the group) for a given file, as well as the user and group that own the file, are shown by `ls -l file` (see Sect. 3.1).

id
chgrp

4.3 The superuser

The superuser is the **administrator** of the system, usually called `root`. The superuser has permission to read and write any file (unless she denies it to herself, which she can change using `chmod`). In particular, root has access to the system files. Since the root account exists on all systems, it must be guarded by a strong password. In addition, many systems do not allow root access over ssh, since the user name is already known and only the password must be guessed. Apart from that, the superuser account provides unlimited power and should only be used for system administration. Hence, most administrators have a normal user account for normal use of the system. To administer the system, they log in as a normal user, before switching over to the root identity using the root password. This can be done by issuing the `su -` command without a user id.

root
su -

Normal users can obtain (partial) root privileges from the superuser to perform specified tasks. In such a case, the user can prefix the command `sudo` to their actual command, upon which the system will ask the user's password and execute the provided command with root privileges.

sudo

5 Job control

In order to start a program, you can simply type its name followed by Enter. The job will run in the **foreground** until it exits, after which the control of the terminal is returned to bash, which will show an empty command prompt.

foreground

If you want to run a program in the current directory, *e.g.* one you have just compiled, you need to prefix `./`, as in `./prog`. The reason is that the current directory (. or ./) is not in your PATH by default for security reasons.

./

If you want to benchmark a program, you can use the `time` command, which will display the real (clock), user (program) and system (overhead) run times of your program: `time ./prog`.

time

5.1 Issuing multiple commands

In order to run a job in the **background**, you can put a single ampersand after the command, for example `./prog &`. This has the effect of returning the control of the terminal back to the user as soon as the program has started, so that you can do something else while the program runs. If the program produces output, that may frustrate what you are doing. The ampersand can be used to start two programs to run at the same time as well: `./prog1 & ./prog2`. In this example, prog1 is started, control is handed back to the shell, which launches prog2 to run concurrently (and in the foreground in this example — a second ampersand would be needed to run prog2 in the background as well).

background
&

When I'm writing or debugging a program, I often compile, run; compile, run; *et cetera* to see if the program's output makes sense. Hence, I'm repeating the same pair of commands very often. In order to save typing, I put the two commands on one line. The default way to do this is by using a semicolon (;), as in `gcc prog.c -o prog; ./prog`. This will *always* issue both commands.[6]

;

[6] Unless the first command *e.g.* wipes your disc.

If the program doesn't compile in the previous example, there is no point in running the code. Hence, I actually use a logical AND by typing a double ampersand (&&): `gcc prog.c -o prog && ./prog`. This will execute prog only if compilation succeeded.

&&

Analogously, the opposite can be achieved using the logical OR: `gcc prog.c -o prog || echo 'Compilation failed'`.

||

5.2 Foreground and background jobs

5.2.1 The jobs, fg, bg and kill commands

While there can be only one foreground job running at any time, several jobs may run in the background concurrently, or be **suspended** (also called stopped). Jobs that need user interaction and are run in the background, will be suspended as soon as they are started. A job running in the foreground can usually be suspended by pressing `Ctrl-Z`. Suspension of a job means that the job is *temporarily* stopped, or rather paused, but can be continued later. Note that this is very different from **killing** a job by pressing `Ctrl-C`, after which the job cannot be continued.

suspend

Ctrl-Z

Ctrl-C

The command `jobs` lists all jobs that were started from the current shell and are suspended or running in the background. The jobs are labelled by a number between square brackets, *e.g.* `[1]`, and the label is defined in the current shell only. You can refer to that job by prefixing a percent sign to the label, `%1` in this example. A plus behind the label indicates the default job that will be referred to if the `%1` indicator is omitted.

jobs

[1]

%1

When a job is suspended, typing `fg` will allow it to continue to run in the foreground. Typing `bg` will allow it to continue to run in the background. You can specify a particular job by typing *e.g.* `fg %3` or `bg %1`. If this indicator is omitted, the default job will be continued. A running background job can be brought to the foreground directly using the `fg` command.

fg

bg

If a job is running in the background, it can be suspended or killed using the `kill` command.[7] The command `kill %1` will kill the selected job using the TERMINATE signal (unless that signal is caught and handled differently by the job). In order to kill a process using a signal that cannot be caught, use `kill -KILL %1`. On Linux systems this is equivalent to `kill -9 %1`. In order to suspend a running background job, use `kill -TSTP %1`. This can be useful if it is using too much CPU time while you quickly need to do something. The job can later be continued using `bg`, or `kill -CONT %1`. For a list with signals and their numbers, type `kill -l` (lower-case ell).

kill

kill -TSTP

kill -CONT

kill -l

In order to have a long job run in the background without using all CPU time, you can lower its priority using the `nice` command, for example `nice -n ./prog &`, where `n` is a number between 0 and 19. The default nice value with which a job is started is 0, and higher values indicate *lower* priorities. Hence, to be able to use your system normally and only allow a job to run when the system is not doing anything else, use `nice -19 ./prog &`.[8] If you forgot to set the niceness of your job when starting it, you can change (increase only) it using the `renice` command. Only root can assign negative nice values, or decrease the nice value.

nice

renice

Note that a job needs standard output and error to exist in order to run. Hence, if you close your terminal after launching a background job, the job will usually be killed soon after. In order to ensure that the program continues running, you should run it in the background and redirect its standard output and error to a file (*e.g.* using `./prog &> output.txt &`, see Sect. 5.3). This allows you to ssh into a machine, start a long job and log out again, while it also provides you with a log file to analyse later. If you don't want

[7] The kill command is named unfortunately, since it can send many types of *signal* to a process, not just kill.

[8] Note that the nice value in this example is +19 — the `-` is a *dash*, not a minus.

to save possible output (or error) messages, redirect to the special file `/dev/null`, which acts like a black hole (see Sect. 5.3).

5.2.2 Processes and the ps, kill and top commands

ps

While *jobs* are defined in the shell they were started from, these programs also have a unique *process* identifier (PID), which can be shown by the command `ps`. While the job ID is defined in the local shell only (and hence `fg` and `bg` work only in that scope), the PID is unique in the system. This allows you to send a signal (using `kill`) to a process that was not started in the current shell. In order to list all your processes, issue `ps x`. The first column usually contains the PID. The command `ps -l` also lists the process state, its parent's PID (PPID), and the priority and nice values of the processes. The `kill` command can be used to kill a process with a given PID (without a percent symbol), *e.g.* `kill -9 12345` to kill the process with PID 12345. This way, processes can also be suspended and continued, using the `TSTP` and `CONT` signals, as we saw above.

ps x

ps -l

kill

top

If a certain process is using up a lot of CPU time, you can identify that process using the `top` command. If installed on your system, `htop` and `atop` are useful alternatives. Top programs sort all processes to CPU usage by default, and list their PIDs, so that it is straightforward to kill the process that is causing the mayhem. Most top programs can kill processes too, usually by typing 'k' and specifying the desired PID.

5.3 Redirection

redirection

You can use **redirection** to use other sources or destinations for standard input, output and error. If you want to send standard output to a file rather than to the screen, you can use the greater-than symbol > (short for 1>): `ls > filelist.txt`. To run a job in the background without the cluttering output, redirect it to /dev/null (which is a special file that works like a black hole[9]): `./prog > /dev/null &`.

>, 1>

/dev/null

While > redirects standard output, it doesn't affect standard error, which is still sent to the screen. In order to redirect both standard output and error, use `&>`: `./prog &> /dev/null`. To redirect only standard error, use 2>: `./prog 2> /dev/null`. If you want to redirect standard error into standard output (*e.g.* because you want to use it later in a pipe), you can use `./prog 2>&1` (without the ampersand, a file named '1' would be created).

&>

2>

2>&1

Redirection of standard input can be useful if you know which commands you should type in a program to do what you want. For example, the following plotting program creates the graph I want from columns 1 and 3 of data.txt and then quits by pressing 'p', '1', '3' and 'q'. When I create the text file input.txt, containing just the line 'p 1 3 q', this can be used as input rather than standard input using the smaller-than symbol <: `./plot data.txt < input.txt`.

<

5.4 Pipes

Pipes can be used on the command line to use the output of one command as input for the next. The symbol used for a pipe is |: `ls -l | less` will list the contents of the current directory and use less as a pager. `ls -1 | wc -l` will display the number of entries in the current directory (the number of lines produced by `ls -1` (using the number one)).

|

6 Editing the command line

Many of the hot keys described in this section are `readline` features. Detailed information can often be found in the `SEARCHING` and `EDITING COMMANDS` sections of the `man readline` page.

man readline

[9] Information goes in and is never heard of again.

6.1 Tab completion

Bash allows you to complete a command, file or directory name by pressing Tab. If the solution is unique, that solution will be used. If not, a partial completion will occur, and the system will print the remaining options by pressing Tab two or three times. This saves you the trouble of removing the command and issuing the `ls` command to see what the options are. **Tab**

Depending on the packages installed on your system, bash completion can be rather smart. For example, if several files exist, but only one subdirectory, `cd` Space Tab will select that directory, since you cannot cd into a file. If the current directory only contains the files file.bak and file.odt, typing `libreoffice` Space Tab will select file.odt, since the extension suggests it is the only solution that can be opened by that program.

6.2 Reusing a previous command

The easiest way to move around the command-line history is by using the **up and down** arrow keys (↑, ↓). Once you arrive at the desired command, you can either use it directly by pressing Enter, or edit it first, and then press Enter. There is no need to move to the end of the line before hitting Enter. **↑,↓**

If the command you intend to reuse was issued more than a few commands ago, it is more useful to do a reverse search by pressing Ctrl R and typing a (unique) part of the command you are looking for. You will see the most recent match appear. If that is not the desired command, press Ctrl R again, or type additional characters to do a more specific search. Once you have found the desired line, either press `Enter` to execute it, or press `Esc` to exit search mode and edit the line before issuing `Enter`. Pressing Ctrl R twice (without typing a search string) will search for the last search string you used. If you want to execute the line found, you can press Ctrl O instead of Enter, which will execute the line and jump to the next line in your history, rather than returning to the end of your history list. If you skipped past the line you were looking for, Ctrl S can search forward again.[10] **Ctrl-R** **Ctrl-O** **Ctrl-S**

6.3 Moving around on the command line

In order to edit the current command-line text, you will need to move around to the position of interest. Using the **left and right** arrow keys (←, →) is the simplest solution. However, in particular if the line is long, it may not be the most efficient way. **←,→**

On most systems, you can use the `Ctrl` or `Alt` key in combination with the **left and right** arrow keys to jump words rather than single characters. **Ctrl-←, -→**

Jumping to the beginning or end of the line can be done quickly using Ctrl A or Ctrl E respectively. **Ctrl-A, -E**

Finally, Ctrl S and Ctrl R can be used to search forward and backward in the current line, as well as in other lines in your history. **Ctrl-S, -R**

6.4 Editing the current command line

If you mistyped or want to adapt a previously issued command to your wishes, you will need to edit the current contents of the command line. The most straightforward way of doing this is by removing content using the `Backspace` key and typing new text. **Backspace**

You can delete entire words using Alt Backspace. **Alt-BS**

[10] On some systems, Ctrl S may freeze the shell. In that case, issue Ctrl Q to continue. You can disable XON/XOFF flow control by typing `stty -ixon` [11].

Ctrl-U, -K Cutting everything to the beginning or end of the line can be done using Ctrl U and Ctrl K respectively. The last text cut this way can be yanked back in at the position
Ctrl-Y of the cursor by pressing Ctrl Y. Directly after that, you can paste earlier cuts with
Alt-Y Alt Y.

Ctrl-T Many typos can be fixed with Ctrl T, which swaps the character under the cursor with
Alt-T the previous one. Pressing Alt T swaps the word under the cursor with the previous one.

If you make a mistake whilst editing the command line, you can undo your last edits by
Ctrl-/ pressing Ctrl /. If you made many mistakes, you can revert to the original command
Alt-R line by pressing Alt R.

6.5 Resetting your terminal

If your terminal starts producing garbage as you type, or doesn't echo the characters you
reset type at all, you can reset it using the `reset` command (which comes with the ncurses package [12]). If it isn't installed, type `echo -e \\033c` instead.

7 Information about your system

7.1 Man pages

One of the most useful, albeit Spartan[11] ways to obtain information on your system are the manual pages or **man pages** [13]. They do not only provide you with the syntax of system commands on the command line, but also with information on system-library calls, standard C library functions and more, which is very useful when working on Linux systems or Linux system programming.

man Man pages are displayed using the `man` command, followed by a command or function name. For example, in order to see all the options of the `ls` command, I simply type `man ls`. Information on C header files can also be found in the man pages, *e.g.* `man stdio.h`.

man -k In order to find the man page you are looking for, you can search by keyword using `man -k`. For instance, to get a list of man pages that deal with semaphores, issue `man -k semaphore`.[12]

7.1.1 Sections

Man pages are categorised into numbered sections. The most important ones are:

1 User commands: information on command-line commands;

2 System calls: information on Linux system calls;

3 C library functions: information on the C standard library;

7 Miscellanea: background information;

9 Kernel Hackers Manual: information on the Linux-kernel API [14] and Linux device drivers [15] (unofficial[13]).

man man A complete list of sections can be found by typing `man man`.

If a command or function is only listed in one section, `man` will automatically list the information from that section. However, in many cases an entry occurs in multiple sections.

[11] Efficient!

[12] This command searches a database with short descriptions of the available man pages. `man -K` searches the actual man pages, which is much slower.

[13] The Kernel Hackers Manual comes with the Linux-kernel source, and can be generated with the command `make mandoc` in the directory `/usr/src/linux`.

For example, `kill` is both a command and a Linux system call. Hence, I need to specify the section number as an option: `man 1 kill` will open the information on the command, while `man 2 kill` shows the Linux programmer's manual. **man #**

Note that apart from syntax, the library manuals also include information such as the header file that needs to be included in order to use a function, and compiler options that are necessary. Two advantages of the man pages is that they are specific to your system and available without a network connection.

7.1.2 Contents of a man page

Each man page is itself subdivided into sections. The following sections often occur, but are not mandatory. They are usually written in capitals.

NAME Name of the command or function;

SYNOPSIS A brief description of syntax, header files and/or compiler options;

DESCRIPTION A description of the command or function;

OPTIONS A detailed description of each command-line option or interface variable;

FILES Files that affect the program or function (*e.g.* settings);

ENVIRONMENT Environment variables that affect the program or function;

EXAMPLES Example usage;

BUGS Known bugs or issues;

AUTHOR The author of the program or function;

SEE ALSO Related programs or functions (with their sections between brackets).

7.1.3 Navigating a man page

A man page usually spits out a lot of text, paginated by a pager. By default, `man` uses the `less` command, but this may be altered by setting the `$PAGER` environment variable (see Sect. 8.1). Section 8.4 shows how to add colours to your man pages when using `less`, which can make the information much easier to read. Section 3.2.1 shows you how to move around and search in a man page using `less` keystrokes.

7.2 Bash help

While the *man pages* give detailed information on *external* commands (*i.e.*, executables that sit in *e.g.* `/usr/bin/`), bash's *built-in commands* are not included. They can be consulted using the `help` command. Without an argument, `help` lists all internal commands with a brief list of options, while `help <command>` gives more information on the command of interest. If you feel that `help` output ought to look more like man pages, use the `-m` option (see `help -m help`). Information on the internal commands can also be found when searching for the `SHELL BUILTIN COMMANDS` section of `man bash`. **help** **man bash**

8 Setting up your bash environment

You can design your bash environment to suit your needs using *e.g.* environment variables and aliases. Unfortunately, once you close your shell, your settings will be lost. This can be solved by saving your definitions in the files `.bash_profile` and `.bashrc` in your home directory, so that they will be executed each time you log in. Below I describe some useful variables and aliases, and give some example content for `~/.bash_profile` and `~/.bashrc`.

8.1 Environment variables

Environment variables are the 'global variables' of your system. They can contain your preferences, such as your favourite editor, or how your command prompt looks. Of course, each time a program wants to open an editor, the system could ask you which editor to use. However, since your preferred editor is likely to remain constant over longer periods of time, it is more useful to define it in the variable `$EDITOR` and have the program check that. You can see the current value of the variable by typing `echo $EDITOR`.[14]

$EDITOR
echo

Other environment variables that are often set include `$PATH`, which contains a list of directories where your system searches for a binary executable with the desired name whenever you type a command,[15] `$PAGER`, which sets the default pager for *e.g.* man pages that are longer than one screen, and `$PS1`, which defines the command prompt at the beginning of each new command line (in the example in Listing 3 it takes the format `[user@machine currentdir]$` using colours). Using a coloured prompt helps you to find the first error in several screenfulls of gcc output following the output from the previous gcc command. (Instead, or in addition, you can `Ctrl-L` to clear your screen between compilations, or start your command line with `clear && gcc ...`.) Also, I have a very different and even more conspicuous prompt for the superuser, constantly reminding me that I am root and should be careful.

$PATH
$PAGER
$PS1
Ctrl-L
clear

export Exporting an environment variable causes it to be available in the environment of a command that is executed later.

8.2 Aliases

An alias provides a new name for a command or set of commands. For example, since I am lazy, I don't bother with typing `exit` each time I want to quit a shell — I use 'lo' (short for 'log off') instead. Also, I often want to list the contents of a directory in coloured, long format. Rather than typing `ls -lGh --color=auto` each time, I have defined the alias `lls` for it using `alias lls='ls -lGh --color=auto'`. From that moment on, typing `lls` as if it were an existing command does exactly what I want. More examples of aliases can be found in Listing 3.

alias

8.3 .bash_profile

The file `.bash_profile` in your home directory is sourced by bash for login shells, *e.g.* when logging in into X, into a text console, or through ssh. It is *not* sourced when starting a new shell from X (though this may have changed recently). In that case, only `~/.bashrc` is sourced and adding the line `[[ -f ~/.bashrc ]] && . ~/.bashrc`, which sources `.bashrc` in your home directory if present, would be a good idea.

The file `~/.bash_profile` is mainly used to set the `$PATH` environment variables, *e.g.* (an electronic version of this file can be found at [16]):

Listing 2: Example content for `~/.bash_profile`.

```
# This file is sourced by bash for login shells, e.g. when logging into X, logging
#   into a text console or login in using e.g. ssh.  This file is NOT sourced when
#   starting a new shell from X.  In that case, only .bashrc is sourced.
# See also https://stackoverflow.com/a/415444/1386750
#
# The following line runs your .bashrc when logging in through e.g. ssh:
[[ -f ~/.bashrc ]] && . ~/.bashrc

# Set and export PATH:
PATH="$PATH:/sbin:/usr/sbin:/usr/local/sbin"
export PATH
```

[14] If not set, the default editor is likely to be vi.

[15] *e.g.* ls will usually be found in /bin/ls. With an empty `$PATH` variable, bash will return `ls: No such file or directory`.

Other variables you may want to set here are `$LIBRARY_PATH`, `$LD_LIBRARY_PATH`, etc. The reason I define `$PATH` here is because it adds to the existing content of that variable. If placed in `~/.bashrc`, many directories could appear twice (*e.g.* if the server started X, and I logged in through ssh). Check the content of your `$PATH` variable by typing `echo $PATH`.

8.4 .bashrc

The file `.bashrc` in your home directory is sourced when starting a shell after logging in (*e.g.* from X), but not when logging in via ssh or a text console. In that case *only* `.bash_profile` is sourced.[16] An electronic version of the example file below can be found at [16].

Listing 3: Example content for `~/.bashrc`.

```
# .bashrc is only read by a shell that's both interactive and non-login (e.g. when
#   starting a terminal from X).  See https://stackoverflow.com/a/415444/1386750

# Coloured command prompt for user (/root):
export PS1="[\[\033[1;31m\]\u\[\033[0m\]@\[\033[1;34m\]\h\[\033[0m\] \W]\$ "
# export PS1="[\[\033[1;41m\]\u\[\033[0;1;7m\]@\[\033[0;1;44m\]\h\[\033[0m\] \W]# "

# Favourite editor and pager:
export EDITOR='emacs'
export PAGER='less'

# History control:
export HISTCONTROL="ignoreboth"   # Ignore repeat commands, cmds starting w/ space
export HISTSIZE=10000             # Make a history file of 10k lines (def: 500)
export HISTFILESIZE=100000        # Make a history file of 100k lines (def: 500)
shopt -s histappend               # Append to history rather than overwrite

# Shell options:
shopt -s cdspell                  # Correct minor typos in dir names on cd command
shopt -s dirspell                 # Correct minor typos in dir names on tab compl.
shopt -s checkjobs                # Do not exit if shell has running/suspended jobs

# Colour in man pages (when using less as a pager - see man termcap):
export LESS_TERMCAP_mb=$'\E[01;34m'  # Blinking -> bold blue
export LESS_TERMCAP_md=$'\E[01;34m'  # Bold (sect. names, cl options) -> bold blue
export LESS_TERMCAP_me=$'\E[0m'      # End bold/blinking
export LESS_TERMCAP_so=$'\E[01;44m'  # Standout mode - pager -> bold white on blue
export LESS_TERMCAP_se=$'\E[0m'      # End standout
export LESS_TERMCAP_us=$'\E[01;31m'  # Underline - variables -> bold red
export LESS_TERMCAP_ue=$'\E[0m'      # End underline
export GROFF_NO_SGR=1

# My aliases for frequently used commands:
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -ip'
alias ls='ls --color=auto'
alias lls='ls -lGh'
alias lo='exit'
alias less='less -Si'
alias du='du -h'
alias ssh='ssh -Y'
# etc...

# Lazy cd'ing:
alias ml='cd ~/work/UU/Teaching/MachineLearning'
# ... and many, many more...
```

9 Using a terminal in a graphical environment

In many cases, you will be using a terminal or console in a graphical environment. The default environment on GNU/Linux systems has been the X Window system x.org, which is currently being replaced by `Wayland` on many systems. The window system in turn allows desktop environments (DEs) like Plasma (KDE), Gnome, Xfce, LXDE, etc. [17]

x.org
Wayland
DE

[16] Which can in turn source `.bashrc`, see Sect. 8.3.

to run. Here are some tips and tricks to facilitate the interaction between your terminal and the rest of your graphical system. I use Plasma in x.org as my desktop environment, but some of the features described below are part of X, while others can probably be configured in other DEs as well.

9.1 Switching between windows

When I write a computer program, I typically use two windows: one with a bash command line that I use to compile my program, run it, and check the (screen) output, and one with my editor. Hence, I want to switch between the two often. Moving my hands from keyboard to mouse and back takes a lot of time, and hence I use a keyboard shortcut for this. The default shortcut to switch between windows in KDE is `Alt` `Tab`. I have configured it to switch from the current to the previously active window (on my current virtual desktop), so that one `Alt` `Tab` takes me from my editor to my command line, and the next `Alt` `Tab` takes me back. Holding `Alt` and pressing `Tab` more than once allows me to reach the other windows on my desktop.

Alt-Tab

9.2 Using virtual desktops

VD

Many desktop environments use *virtual desktops* (VDs). Each VD, or simply desktop for short, acts as a different monitor, and contain windows that are only active if that particular VD is active. This is useful to separate the different things you may be doing at any given time. For example, I am typing this in an editor and have a terminal to run LaTeX in one desktop, while my browser and email program sit in another, and in yet another desktop I have another console to test some of the commands I describe here to see whether I'm not mistaken. I switch between VDs in a similar way to switching windows, using the `Ctrl` `Tab` shortcut.

Ctrl-Tab

9.3 Copy and paste between windows

copy
paste

In the X Window system, selecting a text with the mouse also copies it into the clipboard. Selecting and copying a single word can be easily done by double-clicking it. Pasting in X can be done simply by clicking the middle mouse button or scroll wheel. Note that this copy/paste clipboard is different from that used by `Ctrl` `U`, `Ctrl` `K` and `Ctrl` `Y` for the command line, as described in Sect. 6.4.[17]

xclip

If you want to copy output from a command to the X clipboard, you can use `xclip` [18], typically used with a pipe. For example, `ls |xclip` will copy a list of files in the current directory to the clipboard. Pasting can be done (apart from clicking the middle mouse button) with `xclip -o`. `xclip -o > file.txt` saves the contents of the clipboard to a file.

References

[1] **Fox, B. & Ramey, C.** *Bash.* URL http://tiswww.case.edu/php/chet/bash/bashtop.html. Visited 2016-03-20.

[2] ——. *Readline.* URL https://tiswww.case.edu/php/chet/readline/rltop.html. Visited 2022-04-21.

[3] **Greenwood Software.** *Less.* URL http://www.greenwoodsoftware.com/less/. Visited 2016-03-20.

[4] **Stallman, R.** *Emacs.* URL https://www.gnu.org/software/emacs/. Visited 2016-03-20.

[5] **Moolenaar, B. *et al.*** *Vim.* URL http://www.vim.org. Visited 2016-03-20.

[17] Some configuration of your DE may be needed to allow different clipboards/buffers (*e.g.* emacs and Plasma) to cooperate.

[6] **van der Sluys, M.** *Getting started with Emacs*. URL http://pub.vandersluys.nl. Visited 2022-04-13.

[7] **Allegretta, C. *et al.*** *Nano*. URL http://www.nano-editor.org. Visited 2016-03-20.

[8] **Boyd, S. & Vermeulen, S.** *Nano basics guide*. URL https://wiki.gentoo.org/wiki/Nano/Basics_Guide. Visited 2016-03-20.

[9] **Wikipedia**. *List of text editors*. URL https://en.wikipedia.org/wiki/List_of_text_editors. Visited 2016-03-20.

[10] **Cooper, M.** *Advanced Bash-Scripting Guide*, 2014. URL http://tldp.org/LDP/abs/html/. Visited 2016-08-10.

[11] **Linux, A.** *Arch Wiki: Readline*. URL https://wiki.archlinux.org/index.php/Readline. Visited 2016-08-02.

[12] **Ben-Halim, Z., Raymond, E. & Dickey, T.** *Ncurses*. URL https://www.gnu.org/software/ncurses/. Visited 2016-03-20.

[13] **The Linux man-pages project**. *Man pages*. URL https://www.kernel.org/doc/man-pages/. Visited 2016-03-20.

[14] **The Linux Kernel Organization**. *The Linux Kernel API*. URL https://www.kernel.org/doc/htmldocs/kernel-api/. Visited 2016-05-04.

[15] ——. *Linux device drivers*. URL https://www.kernel.org/doc/htmldocs/device-drivers/. Visited 2016-05-04.

[16] **van der Sluys, M.** *Environment settings (bash, emacs, git)*. URL https://github.com/MarcvdSluys/han-ese-ops-env. Visited 2019-02-11.

[17] **Wikipedia**. *Desktop environment*. URL https://en.wikipedia.org/wiki/Desktop_environment#Examples_of_desktop_environments. Visited 2016-03-20.

[18] **Saunders, K. & Ãstrand, P.** *Xclip*. URL https://github.com/astrand/xclip. Visited 2016-03-20.