UNIX/Linux Quick Start

This tutorial list some of the most common unix/linux commands. To learn more about each command and its options, read the associated man page, e.g. **man** *command*

Getting Help

man man	*Display how to use the online manual pages*
man *subject*	Looks up *subject* in the online manual pages
man -k *keyword*	Search all manual entries for a particular keyword in the title
apropos *keyword*	Search all manual entries for a particular keyword in the title (same as **man -k**)
whereis *command*	Locate the binary, source, and manual page files for a *command*
which *command*	Locate a *command* path
help	Display information about bash built-in commands
whatis *command*	Display the title line of command's man page

Managing the file system

cd *dir*	Go to the directory called dir
cd ..	Go to the parent directory (the directory above the current directory)
.	Current directory
..	Parent directory
~	Home directory
~*user*	Home directory of *user*
cd	Go to your home directory
ls	List the files in the current directory
ls -F	List the files in the current directory, and indicate subdirectories by appending a backslash (/) to their name
ls -a	List all the files in the current directory, including the hidden ones. Hidden files are prefixed by a dot
ls -A	List almost all the files in the current directory (list the hidden files, but do not list implied **.** and **..)**

ls -l List all files in the current directory using the long format (the long format shows a lot of miscellaneous info about each file)

ls -lt Sort files by modification time (the most recently changed file is listed on the top)

cp *file1 file2* Make a copy of *file1* and call the copy *file2* (if *file2* already exists it will be overwritten)

ln -s *file target* Link symbolically *file* to *target*

mv *file1 file2* Rename a file from old name *file1* to new name *file2* (if *file2* already exists it will be overwritten).

mv *file dir***/** Move *file* from its present directory into another directory (*dir*)

mv *dir1 dir2* Move or rename a directory from old name *dir1* to new name *dir2*

rm *file* Remove *file*

rm -r *dir* Remove directory *dir* (**-r** recursively remove *dir* contents; if you also would like to be prompted before removing files add **-i**)

pwd Print name of current/working directory

passwd Set or change your local login password

mkdir *dir* Create a new directory *dir*

rmdir *dir* Remove empty directory *dir* (if the directory is not empty use**: rm -r** *dir)*

find . -name *file* **-print** Finds all paths containing *file* in the current directory or below it

wc *file* Counts the lines, words, and characters in *file*

du Displays disk usage in Kbytes by directory, starting in the current directory and working down

du *file* Display disk usage in Kbytes for *file* (or *directory*)

du -s Summarize total disk usage

gzip *options files* Compress or expand *files.* Compressed files are stored with **.gz** ending

gunzip *options files* Compress or expand *files.* Compressed files are stored with **.gz** ending

tar options files Create or extract archives of *files.* Archived files are stored with **.tar** ending

`tee` *file* — Split the output of a program so that it can be seen on the display and also be saved in a *file. (**`tee -a`** file* appends the output to the end of *file* instead of writing over it).
Example: **`echo "hello" | tee logfile`**

`uniq` *file* — Remove repeated lines in *file*

`touch` *file* — Create an empty *file* or update the time stamp of an existing *file*

`sort` *options file* — Sort the lines of *file* according to the *options* chosen

`hostname` — *Display the name of the current machine*

When specifying a path, use the **/** character to separate directories and file names.

Viewing files

less *file* — Shows the contents of *file*
type **q** to quit the listing of text and return to the shell prompt
type **ctrl-u** to scroll up half screen size
type **ctrl-d** to scroll down half screen size
type **j** to scroll down one line
type **k** to scroll up one line

`file` *file* — Determine the type of a file

`cat` *files* — Concatenate and display files

`spell` *file* — Reports possible misspelled words in *file*

`diff` *file1 file2* — Display differences between two text files

`cmp` *file1 file2* — Compare two files (text or binary) and list where differences occurs

`grep` *options expr files* — Searches *files* for a specified string or expression

`head` *file* — Shows the first few lines (by default 10 lines) of *file*

`tail` *file* — Shows the last few lines (by default 10 lines) of *file*

Editing files

`vim` *file* — advanced text editor (widely used)

gvim *file* graphical implementation of vim

emacs *file* extensible, customizable, self-documenting, real-time display text editor (widely used)

nano *file* small, friendly to use text editor

gedit *file* official text editor for the GNOME desktop environment

Managing Processes

ps List the status of all processes started during your login session

jobs List all background jobs started during your login session

top Display system summary information as well as a dynamic real time view of running tasks currently being managed by the OS kernel

kill Kill an unwanted process. The process to kill is specified through the process id number (*pid#*) or the job control number (**%***n*):
kill *pid#*
OR
kill *%n*

Useful Tips and Commands

history	List the most recent commands executed
!!	Repeat the last command
!*n*	Repeat command *n* from the history list
!*PATTERN*	Repeat last command beginning with *PATTERN*
Ctrl+p	Scroll backward through the previous shell commands
up-arrow	Scroll backward through the previous shell commands
down-arrow	Scroll forward through the previous shell commands
Ctrl+u	Delete the last line you typed
Tab	While typing, complete file/path name as much as possible
Delete or **Backspace**	Erase the last character you typed

clear	Clears the screen
sftp *user@host*	Secure file transfer session
ssh –CY *user@host*	Secure access to a remote computer
exit	Cause normal process termination
Ctrl+c	Terminate the active program.
Ctrl+z	Suspend the current active program. To bring it back, use %*jobnumber*, e.g. %1 (the number comes from the jobs list)
%	Continue last job suspended. Alternatively type **fg** (foreground)
*	Wildcard. Any number of characters (not "**.**"). Can be used to express patterns matching multiple file names (e.g. typing **ls –l** *dir/*.sp* will list all *dir* files ending with .sp)
?	Any single character (not "**.**").
command **&**	If you put & at the end of a *command,* the process runs in the background, letting you type more commands on the shell (e.g. firefox &)
fgrep –i *pattern file*	Display all lines in *file* that contain *pattern* (case insensitive)
whoami	Displays the logged-on user's name
w	Report who is logged in and what they are doing
date	Display the current date and time
chmod *options file*	Change the permissions on a *file* (or *directory*)
chgrp *group file*	Change the *group* of a *file* (or *directory*)
chgrp *owner file*	Change the *ownership* of a *file* (or *directory*)
source *file*	Execute the lines in *file* as if they where typed to the shell
script	Start saving everything that happens in a file, Type **exit** when done.
dmesg	Print the system's boot up messages
echo "*some message***"**	Print the string *some message* on the screen

diff and grep utilities

`diff -c file1 file2`

Compare files line by line. The output put **+** in front of lines that are added in file2, **–** in front of the lines that are deleted in file2 and **!** in front of the lines that were changed in file2.

`grep` [option(s)] pattern [file(s)]

grep (general regular expression print) print lines matching a pattern. It searches for the specified pattern in file(s), or if you choose to omit file(s) it searches the terminal's standard input.

`grep -i` *pattern files* — makes the search case-insensitive

`grep -v` *pattern files* — negate the search (the search will return everything that doesn't match the pattern)

Examples:

grep "area()" program.c — looks for all the lines where the string "area()" occurs in program.c

grep "area(.*)" program.c — grep can match not only exact strings, but also general patterns. These patterns are called "regular expressions". A **.** in a grep pattern matches any character(including no character), and following it with a ***** means the character can be repeated any number of times (including zero times). Consider the case where area() takes a number of arguments.

grep Exception logfile.txt | grep -v ERROR — Search logfile.txt for occurrences of Exception but exclude all occurrences that contain ERROR

grep -c "ERROR" logfile.txt — Print how many times patterns containing ERROR occurred in logfile.txt

egrep 'ERROR|Exception' logfile.txt — Search for either pattern containing ERROR or Exception in logfile.txt

grep -w ERROR logfile.txt — Searches only for instances of ERROR that are entire words. For example it does not match SysERROR

grep 'ERROR\>' logfile.txt — Searches only for patterns ending in ERROR. For example it matches the word SysERROR

grep '\<ERROR' logfile.txt — Searches only for patterns starting in ERROR. For example it matches the word ERRORSys, but not SysERROR.

grep '\<ERROR\>' logfile.txt Equivalent to **grep –w 'ERROR' logfile.txt**

grep -l 'main' *.java	List the names of all java files in the current directory whose contents mention main.
grep -n ERROR logfile.txt	Shows on which lines the pattern ERROR has appeared

Note: the use of single quote ' or double quote " for delimiting regular expressions is primarily meant to deal with white spaces. In Unix/Linux the exact behavior of the single quotes and the double quotes is actually dependent on the shell. In our case we assume the use of the bash shell. The choice between single or double quotes is only important if the search string contains variables or items that you expect to be "evaluated". With single quotes, the string is taken literally, no expansion takes place. With double quotes, variables are expanded.
Example:

```
VAR="Kameamea"
grep '$VAR' logfile.txt
grep "$VAR" logfile.txt
```

The first grep will look for the literal string $VAR. The second will expand the $VAR variable and look for the string `Kameamea`.
In doubt, a simple tip to find out what we should expect is to use the echo command.

```
echo '$VAR'
```

will print $VAR

```
echo "$VAR"
```

will print `Kameamea`.

There are three version of grep in UNIX: grep, fgrep, egrep. fgrep stands for "fixed grep", and egrep stands for "extended grep". The difference between grep and fgrep is that while grep matches regular expressions, fgrep matches literal strings. When you want to search for an ordinary string, if you use fgrep, there is no need to precede special characters with \. The difference between grep and egrep is that egrep supports an extended set of regular expressions and allows for a few more useful features

Regular Expressions by Example

A regular expression, also referred to as regex or regexp, is a sequence of characters that provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters.

grep -w 't[a-i]e'

The brackets have a special meaning. They mean to match one character that can be anything from a to i. It matches words like tee, the, and tie.

grep -w 'cr[a-m]*t'

The * means to match any number (including zero) of the previous character, which in this case is any character from a through m. It matches words like craft, credit, and cricket. Also note that since

the * means to match any number of characters including no characters, for example: cr[a-m]*t could match the letter sequence crt that is cr and a t with zero characters between them.

grep -w 'kr.*n'

The . matches any character and the * means to match the dot any number of times. It matches words like kremlin and krypton, but also krn.

egrep -w '(th|sh).*rt'

The | means to match either the th or the sh. egrep is just like grep but supports extended regular expressions that allow for the | feature. Note how the square brackets mean one-of-several-characters and the round brackets with |'s mean one-of-several-words. It matches words like shirt, short, and thwart.

grep -w 'thr[aeiou]*t'

A list of possible characters can be placed inside the square brackets. It matches words like threat and throat.

grep -w 'thr[^a-f]*t'

The ^ after the first bracket means to match any character except the characters listed. It matches words like throughput and thrust, but the word thrift is not matched because it contains an f.

grep –w '$VAR'

It searches for the word $VAR

grep –w "$VAR"

It searches for the value of the variable $VAR (e.g. Kameamea)

egrep 'colou?r'

It might match both colour and color

egrep 'co{3}'

It might match cool

egrep 'co{1,}'

It might match color and coool

Usually, you will use regular expressions to search for whole lines that match, and sometimes you would like to match a line that begins or ends with a certain string. The ^ character specifies the beginning of a line, and the $ character the end of the line. For example, `'^The'` matches all lines

that start with a `The`, and `'hack$'` matches all lines that end with `hack`, and `'^ *The.*hack *$'` matches all lines that begin with `The` and end with `hack`, even if there is whitespace at the beginning or end of the line.

Because regular expressions use certain characters in a special way (these are . \ [ ] * + ?), these characters cannot be used to match characters. This restriction would severely limits you from trying to match, for example, file names, which often use the . character. To match a . you can use the sequence \. which forces interpretation as an actual . and not as a wildcard (wildcard or meta-character is an alternative term for special character). The regular expression myfile.txt might match the letter sequence myfileqtxt or myfile.txt, but the regular expression myfile\.txt will match only myfile.txt.

Summary of basic special characters

`^`	Match the empty string at the beginning of a line
`$`	Match the empty string at the end of a line.
`\`	*Turn off the special meaning of the next character*
`[ ]`	*Match any one of the enclosed characters, as in [aeiou]. or use Hyphen "-" for a range, as in [0-9].*
`[^ ]`	*Match any one character except those enclosed in [ ],.*
`.`	*Match any single character, except new line*
`?`	*The preceding item is optional and matched at most once.*
`*`	*The preceding item will be matched zero or more times.*
`+`	*The preceding item will be matched one or more times.*
`{n}`	*The preceding item is matched exactly n times.*
`{n,}`	*The preceding item is matched n or more times.*
`{,m}`	*The preceding item is matched at most m times.*
`{n,m}`	*The preceding item is matched at least n times, but not more than m times.*

Standard Input and Output

All standard Unix/Linux commands make use of 3 standard I/O file descriptors:

fd (file descriptor)	Description	Name	Default
0	Standard input	Stdin	Keyboard
1	Standard output	Stdout	Screen
2	Standard error	Stderr	Screen

I/O Redirections

command **>** *filename*	Write the stdout of *command* to *filename*. If *filename* already exists it will be overwritten.

command **>>** *filename*	Append the stdout of *command* to the end of an existing *filename*
command **2>** filename	Write the stderr of *command* to *filename*. If *filename* already exists it will be overwritten.
command **2>>** filename	Append the stderr of *command* to the end of an existing *filename*
command **<** *filename*	Read the *command*'s stdin from *filename*
command **<<** *word*	Read the following lines until a line with only *word* and use these as stdin
command1 **|** *command2*	Redirect the output of *command1* to the input of *command2*. This is called "piping" the output of the first command into the input of the second *command*.
*n***>&***N*	Set the file descriptor *n* to whatever the file descriptor *N* points to. If n is missing, the default used is stdout.

Examples:

`date > datefile`	Store the output of the date command in a file named datefile
`gcc sample.c 2>> errfile`	Compile the file sample.c and append the error messages in a file named errfile
`ls existingfile nonexistingfile > logfile 2>&1`	Save both stdout and stderr in a file named logfile
`ls existingfile nonexistingfile 2>&1 > log`	The rule is that any redirection sets the file descriptor (a.k.a. handle) to the output stream independently. **2>&1** set handle 2 (stderr) to handle 1 (stdout); then **>** set handle 1 to the file log, but it does not change handle 2 (stderr). The result is that the standard output is written to the file log, but errors are sent to the screen.
`echo "hello" > /dev/null`	Discard stdout
`echo "hello" >&2`	Redirect stdout to stderr

`ls existingfile nonexistingfile > /dev/null 2> errlog`	Discard stdout but write stderr in a file named errlog
`ls -lrt | tee listfile`	Direct the output of the ls command to both stdout and the file listfile

References

[1.] Sobell, *A Practical Guide to Linux Commands, Editors, and Shell Programming*, 2/e, Prentice Hall, 2009, pp.257-263

[2.] Mike Lamasney, *Introduction to Unix Usage*, UC Berkeley, 1996

[3.] CS 107: Computer Organization and Systems (Spring 2011), *Stanford University*, Julie Zelenski
https://courseware.stanford.edu/pg/courses/169631/cs107-spring-2011

[4.] UNIX Documentation at Stanford, Stanford University Information Technology Services
https://itservices.stanford.edu/service/unixcomputing/unix

[5.] UNIX Command Summary, Stanford University Information Technology Services
https://itservices.stanford.edu/service/unixcomputing/unix/unixcomm

[6.] Common unix commands and utilities, Stanford University School of Earth Sciences
https://itservices.stanford.edu/service/unixcomputing/unix/unixcomm
http://pangea.stanford.edu/computing/unix/shell/commands.php

[7.] Redirection
http://en.wikipedia.org/wiki/Redirection_(computing)

[8.] Paul Sheer, *LINUX*, 2002
http://rute.2038bug.com/rute.html.gz