Vlaanderen
is supercomputing

Command line tools

A short introduction to the UNIX shell

Thomas Danckaert

VLAAMS
SUPERCOMPUTER
CENTRUM

vscentrum.be

1 Introduction

This is a short reference, written for the course "Introduction to Linux" for new users of the UAntwerpen HPC clusters. The purpose of this document is to explain some of the most commonly used shell commands with their basic options in an accessible way, so new users can quickly gain some practical experience.

These instructions should provide you with all the information you need in order to solve the exercises, but only serve as an introduction to the many options and features of these commands. To see all available options for a command, you can read the man page:

```
    $ man [COMMAND]
```

Man pages are often written in a very succinct style. For more elaborate explanations and detailed discussion of every feature of a command, the manuals provided by the GNU project are very useful.

- Most commands discussed here are part of a package called 'coreutils', for which you can find the manual at

  https://www.gnu.org/software/coreutils/manual/html_node/index.html.

- For information on Bash and built-in shell commands specifically, we refer to the Bash manual at

  https://www.gnu.org/software/bash/manual/html_node/index.html.

- `sed` is an extremely versatile tool with a separate manual of its own:

  https://www.gnu.org/software/sed/manual/html_node/index.html.

On systems where these manuals are installed, you can read them from the terminal as well. To read the manual section on a command, typing

```
    $ info [COMMAND]
```

usually takes you straight to the manual section about that command. For instructions on how to use the `info` viewer itself, you can run

```
    $ info info "Getting Started" Help
```

Note: there are many different shells (i.e. `bash`, `sh`, `ksh`, ...), and many different implementations of these command line tools (GNU, BSD, commercial implementations, ...). This document assumes we are using the Bash shell and the GNU command line tools, which are the default on the VSC systems and probably most HPC clusters worldwide. Nevertheless, a lot of the material applies to all versions and implementations.

2 The file system

`ls` – List directory contents

```
$ ls [OPTION]... [FILE]...
```

The `ls` command lists files and directories on the file system. If if no **`FILE`** arguments are given, it lists the contents of the current directory. With **`FILE`** arguments, `ls` will list each file or – if **`FILE`** is the name of a directory – its contents.

Many options are available to make `ls` display additional information about files, or change the output order and formatting. Some of the most commonly used options are

`-a, --all` Do not ignore entries starting with '.'.

`-d, --directory` List directories themselves, not their contents.

`-l` Use a long listing format. This displays additional information such as file size, modification date and time, ownership and permissions.

`-h, --human-readable` If used together with **`-l`**, print file sizes in human readable format.

`--sort` By default, `ls` displays entries in alphabetical order. You can use the options **`--sort=time`** or **`--sort=size`** – or the respective short options **`-t`** and **`-S`** – to sort by modification time or file size. Other sorting options are **`extension`** (**`-X`**), **`version`** (**`-v`**), or **`none`** (**`-U`**)[1].

`-r, --reverse` Reverse the order while sorting.

[1] With sorting order **none**, files are displayed in the order in which they are stored in the directory (often, this is the order in which they were created, but this depends on the details of the file system).

locate – Look up files by name

```
$ locate [OPTION]... PATTERN ...
```

locate takes one or more patterns and looks for matching files in a periodically updated database of file names. The **PATTERN** arguments can be simple text strings (which match any filename that contains them), shell wildcard patterns, or – when combined with the **--regex** option – regular expressions.

If you are looking for a certain file, locate is a fast and simple tool, with the following restrictions:

- locate can only find files which existed when the database of filenames was last updated.
- Not all files are included in the database. For example, personal directories of users are not indexed on the UAntwerpen HPC systems.

Example:

```
$ locate */hosts
```

find – Search for files in a directory hierarchy

Like locate, find can search for files, but it offers many more options and works according to a different principle:

- find does not rely on databases of file names, but actively searches the file system while it is running instead. Therefore, find can see any file that exists at the time it is called (if you have reading permission for the directory where it is located).
- Because find searches the file system instead of a fixed database, it can be much slower than locate.

- `find` can use arbitrary combinations of many different search criteria besides the file name, including but not limited to file ownership, permissions, creation, access or modification times.

- Using “actions” such as **`-exec`** or **`-ok`**, you can immediately run arbitrary commands on the matched files.

Because of its many features, `find` is a powerful, but complex, command. It is called according to the following convention:

```
$ find [PATH...] [EXPRESSION]
```

`find` recursively searches for files that match the search criteria in **`EXPRESSION`**, starting from every location in the list of **`PATH`** arguments. The list of search **`PATH`**s must appear first, the **`EXPRESSION`** must appear last. Arguments are optional: without **`PATH`**, `find` searches from the current working directory; without **`EXPRESSION`**, `find` matches every file it encounters, and prints the file name.

Search criteria and actions are given in **`EXPRESSION`**. Many properties of a file can be used as search criteria, but we only list a few of the most common **`tests`**. The manual for `find` has a complete list.

`-name PATTERN` Test if the file name matches **`PATTERN`**, a shell wildcard expression. Example: **`-name '*.txt'`** matches files ending in **`.txt`**.

`-type [dfl...]` Test if the file is a **d**irectory, regular **f**ile or symbolic **l**ink. Refer to the manual for more exotic file types.

`-size n[bcwkMG]` Test the size of the file, in units of **b**locks, bytes (**c**), words, **k**ilobytes, **M**egabytes or **G**igabytes. To find files bigger than or smaller than a given size, use a prefix of **+** or **-** respectively. Example: **`-size +1M`** matches files larger than 1 Megabyte.

`-readable, -writable, -executable` Test if you can read, write or execute the file.

If an expression contains multiple tests in a row, files must satisfy all of them to be considered a match. Alternatively, you can combine tests with **`-or`**, or negate the result of a test with **'!'** or **-not**. For complex combinations of tests, you may need to use parentheses **'(' ')'** to enforce the correct precedence.

EXPRESSION can also contain **actions** to be performed on matching files. If no action is given, `find` prints the name of each matching file.

-exec COMMAND {} ';' Run an arbitrary **COMMAND** on the matching files. The name of each matching file is inserted in place of the **{}** braces. '**;**' indicates the end of the command (you should quote or escape it, because the shell will interpret an unquoted **;** as the end of your `find` command, instead of passing it on to `find`).

-ok COMMAND {} ';' Identical to **-exec**, but asks for confirmation before executing a **COMMAND**.

Example: find all files in the present directory and its subdirectories:

```
$ find
```

Example: find all pictures (with extension `.jpg`) under the present directory:

```
$ find . -name '*.jpg'
```

Note the `''` quotes surrounding the pattern `*.jpg`[2]. The '.' is optional here, as `find` searches the present directory by default if no **PATH** arguments are given.

We can add an action to the command to move the matching files to another location:

```
$ find . -name '*.jpg' -exec mv  $HOME/Pictures/ ';'
```

We can use **-or** and parentheses to include files with extension `.png` as well:

```
$ find . '(' -name '*.jpg' -or -name '*.png' ')' \
-exec mv {} $HOME/Pictures/ ';'
```

[2] Without quotes, the shell could expand `*.jpg` into a list of file names instead of passing it on as an argument to `find`.

`tar` – Create or unpack tar archives

```
$ tar [OPTION]... [FILE]...
```

`tar` – originally an abbreviation for "tape archive" – can create and extract `.tar` archive files. `tar` files can contain arbitrary sets of files and directories, including their ownership and permission data. Plain `tar` files do not compress the data, therefore `tar` is usually combined with `gzip`, `xzip` or `bzip` to compress the archive file.

First, choose one of the following options to tell `tar` to **c**reate, e**x**tract or lis**t** the contents of an archive, and which file to work with:

`-c` Create an archive file.

`-x` Extract the contents of an archive file.

`-t` List the contents of the archive.

`-f FILE` Provides the name of the archive file you want to create, extract or list. When you create a new archive, and a file with the same name already exists, it is replaced. If no **`-f`** argument is given, GNU `tar` reads archive files from standard input, or writes files to standard output.

`-v` Enable verbose output. If you use **`-v`** together with options **`-c`** or **`-x`**, `tar` prints the name of every file it archives or extracts. If you combine **`-v`** with option **`-t`**, `tar` prints a verbose listing of the archive contents, showing ownership and permissions, size and modification date.

To use compressed archive files, provide one of the following options:

`-z` `gzip`

`-J` `xzip`

`-j` `bzip`

Apart from a number of options, `tar` accepts a list of **`FILE`** arguments, which can be files or directories. When you create an archive (**`-c`**), this is the list of files and/or directories that should be included. When you extract or list the contents of an archive, the **FILE** arguments are optional: if a list is given, only those files or directories will be extracted or listed; if no list is given, `tar` extracts or lists all of the archive's contents.

Examples

Extract the contents of a file:

```
    $ tar -xf archive.tar.gz
```

Archive all `txt` files in the current directory, with `gzip` compression:

```
    $ tar -czf textfiles.tar.gz *.txt
```

List the contents of the previously created file:

```
    $ tar -tf textfiles.tar.gz
```

Archive the contents of a directory, with `xzip` compression:

```
    $ tar -cJf my_directory.tar.zs ./my_directory
```

`zip` – Create and modify zip archives

```
    $ zip [OPTION]... [ZIPFILE] [FILES]...
```

The `zip` commands creates archive files in the familiar .zip format. At the command line, specify options first, followed by the name of the .zip archive you want to create or modify, and the list of `FILE`s you want to include in the archive. If the provided .zip archive already exists, `zip` will add the `FILE`s to the existing archive, or update them if the archive already contains files with the same name.

`-r, --recurse-paths` The list of `FILE`s to be added to the archive can contain directory names as well. By default, the resulting .zip archive will only contain the (empty) directory. Use the option **`-r`**, to add the contents of these directories – and their subdirectories – as well.

-0, -1, -2, ..., -9 A number from **-0** to **-9** sets the compression level. The default level is **-6**. Higher levels will result in smaller archive files, but might take more time to compress or extract.

We refer to the command's man page for many more advanced options to add files to an archive, update or remove existing files, or select files using on pattern matching.

Examples

Compress all `txt` files in the current directory:

```
$ zip txtfiles.zip *.txt
```

Compress a directory with all contents:

```
$ zip -r my_directory.zip ./my_directory
```

`unzip` – Extract files from zip archives

```
$ unzip [OPTION]... [ZIPFILE] [FILES]...
```

Use `unzip` to extract contents from .zip archives. To extract all contents of the archive `myfile.zip` into the current working directory, simply call `unzip` with a single argument:

```
$ unzip myfile.zip
```

Optionally, you can provide a list of `FILE`s to extract only those files from the archive.

To inspect the contents of an archive **without extracting** them, you can use the options `-l` or `-v`.

-l List archive contents (short format): print names, uncompressed file sizes and modification dates.

-v List archive contents (verbose format): print additional information about the files contained in the archive, such as their compressed file size and the compression method used.

3 Processes

ps – List processes

```
$ ps [OPTIONS]
```

`ps` provides information on processes currently running on the system. Without any options, only processes running in your current shell session are listed. For each process, `ps` prints the process id (`PID`), name of the `COMMAND` and used processor `TIME`, as well as the `TTY` interface[3] it is connected to.

Using various options, you can control which information is shown, and which processes are listed. Note that, for historical reasons, `ps` accepts options with and without leading '-'.

x Also show processes not connected to a terminal.

a Also show processes from other users. If combined with **x**, `ps ax` shows all processes by all users.

u displays processes in "user-oriented" format: displays some additional information, such as user name, processor and memory usage, and the full command line that started the process.

o format display information according to a chosen **format**, a list of names of fields such as **pid**, **user**, **command**, ... See the `man` page for `ps` for a full description of all formatting fields and options.

k keys sort output according to the given list of **keys**. **keys** is a comma-separated list of fields, and the output is hierarchically sorted according to the chosen keys. For example, the command `ps au --sort user,pid` sorts processes by user id, and then sorts processes for each user by their process id. Add '-' in

[3] `tty` originally stands for TeleTYpewriter, and is the Linux system interface for keyboard terminal sessions.

front of the field name to sort in reverse order. Again, we refer to the `man` page for a full list of sorting options.

`U userlist` Given the option **`U`** and an argument **`userlist`**, a comma-separated list of one or more user names, `ps` only displays processors that belong to one of those users

`-T` Display a separate line of information for each of a process' threads.

Examples

List all processes, in user-oriented format:

```
$ ps aux
```

List all processes, sorted by ascending CPU usage:

```
$ ps aux k%cpu
```

`top` and `htop` – List processes interactively

`top` – and its younger cousin `htop` – can be regarded as an interactive, "live" version of `ps`.

```
$ top [OPTIONS]
```

Usually, you run `top` without any options to see a continuously updated list of processes, and some system usage statistics. By default, processes are listed in order of decreasing processor usage ('`CPU`'). Not all processes will fit in the terminal, but you can use the arrow keys to scroll up or down. `top` can also show the threads of each process if you switch to thread view by pressing **`H`**.

Some of the fields shown by default are:

`PID` The process id.

`USER` The user id.

RES Resident memory, a good measure for the amount of physical memory currently occupied by the process.

SHR Shared memory, the amount of resident memory that is shared with other processes.

%CPU The percentage of available processor time currently used by the process, measured in percents of one processor core – for multi-threaded processes, this can be more than 100%.

%MEM Memory usage: resident memory as a percentage of available memory.

TIME Total processor time used by the process so far.

`top` doesn't show all available information by default. To show further data (or disable fields that don't interest you), press **f** ("Fields Management"). This takes you to a menu with a list of all available fields, where you can select new fields using the arrow keys, and mark them for display with the **d** key. A very useful field is **P**, the processor core a thread or process last ran on. This allows you to verify whether multiple threads and processes are running on the same core, which can slow down parallel calculations, sometimes by a big factor.

`htop` is a more modern version of `top` which presents similar information in a slightly more advanced interface. It displays usage bars for every processor core on the system, as well as the memory, which gives you a quick impression of the system's total load. Hit function key **F1** or **h**, to get help; **F2** or **C** to configure which fields are displayed.

jobs – Controlling jobs in the shell

```
$ jobs
```

Every active pipeline (consisting of one or more processes) in the current shell session is known as a 'job'. If you move jobs to the background, either by terminating a pipeline with **&**, or by interrupting them with **Ctrl + z**, multiple active jobs can exist at the same time.

Jobs are numbered starting from 1. The `jobs` command prints a list of all currently active jobs, with their job number, status ("Running" or "Stopped") and the pipeline used to start them.

Commands that act on jobs, such as `kill`, `fg` (run job in foreground) and `bg` (run job in the background), identify jobs by a **`jobspec`** argument. This is the number of the job as reported by `jobs`, preceded by `%`. For example, the following command lets job 2 continue running in the background:

```
$ bg %2
```

`kill` – Terminate or pause processes

```
$ kill [-sigspec] pid | jobspec ...
```

`kill` can terminate or interrupt processes by sending them a signal. You can terminate a single process by providing its **`pid`**, the process id which you can obtain from `ps` or `top`. For example, the following command terminates process 54399:

```
$ kill 54399
```

You can also terminate a shell job instead of a simple process, by providing a **`jobspec`** instead of a **`pid`**. If this job is a pipeline consisting of multiple processes, all those processes are terminated by the same `kill` command. For example, to terminate job 2, use the following command:

```
$ kill %2
```

By default, `kill` sends processes the **TERM** signal, asking them to clean up and exit. However, there are other signals you can send, using the optional argument **`-sigspec`**. There are a number of different signals, each of which has a different meaning. Every signal has a name and a number which can be used interchangeably. For example, the **STOP** signal has number **19**, so the following commands are equivalent:

```
$ kill -STOP 54399
```

```
$ kill -19 54399
```

We list a few of the most common signals, you can run `kill -l` to see a full list.

`-TERM, -15` Processes receiving a **`TERM`** signal are not immediately stopped, but get a chance to shut down gracefully (for example by saving intermediate data). This is the clean way to stop a process, and also the signal sent by default if you do not provide a **`-sigspec`** argument.

`-KILL, -9` The **`KILL`** signal immediately ends the process. If you want to end a process, the recommended way is to try the **`TERM`** signal first (i.e. use `kill` without **`-sigspec`** argument), and try **`-KILL`** if a process does not respond to **`TERM`**.

`-STOP, -19` The **`STOP`** signal stops (pauses) the process. All calculations are stopped, but the process is not destroyed and may be resumed later.

`-CONT, -18` The **`CONT`** signal continues a previously stopped process.

`time` – Measure the time used by a command

```
$ time COMMAND
```

If you precede a command (or pipeline) by ‘`time`’, the shell will report the total time taken by that command. `time` reports three values:

real “real” time is the actual time that has passed in the physical world (sometimes referred to as “walltime” – the time that has gone by as measured by a clock on the wall).

user This is the time spent by processor cores (“CPU time”) executing the code of the command itself. If the command uses multiple processor cores, this value can be larger than the real time. Conversely, user time can be much less than real time if the command doesn’t do a lot of work (for an example, try `time sleep 5`).

sys The CPU time spent by the operating system (kernel) on behalf of your command. Examples of tasks that count towards “system” time are: input/output, thread management and allocation of memory.

4 Working with text

wc – count lines, words or characters

```
$ wc [OPTION]... [FILE]...
```

wc prints the number of lines, words and bytes (or characters) for each file, and – if more than one file is given – the total counts. If no file is given, wc reads from standard input. You can select what is printed by providing one or more of the following options:

-l, --lines

-w, --words

-c, --bytes

-m, --chars

less – Scroll through text

```
$ less [FILE]
```

less displays the content of a file, or – if no **FILE** is provided – standard input.

While less is running, you can navigate the text by pressing the appropriate keys. Some useful commands:

j	Scroll forward one line (you can also use the down arrow key).
k	Scroll backward one line (you can also use the up arrow key).
f	Scroll forward one window.
b	Scroll backward one window.

q Quit.

h Read help, where you can learn about further commands not explained here. Press **q** once to leave the help screen and go back to the text screen.

A very useful feature is the **search** function. To search for a word in the text you are viewing:

1. Press the **/** ("slash") key. A cursor appears at the bottom of the terminal.

2. Type the word you want so search for, and press **`enter`**. `less` jumps ahead to the next occurrence of that word (if it appears anywhere below your current position in the text).

3. Press **n** to go to the next occurrence, press **`Shift + n`** to go back to the previous occurrence.

head – Print the first part of files

```
$ head [OPTION]... [FILE]...
```

`head` prints the first 10 lines of each **`FILE`** on standard output. If no **`FILE`** arguments are provided (or when **`FILE`** is **`-`**), `head` reads from standard input instead. Using the option **`-n`** or **`--lines=`** with an integer value **k**, you change the number of printed lines. With a negative number **`--lines=-k`**, `head` prints all but the last **k** lines.

Example: print the first 5 files with the extension `.txt` in the current directory.

```
$ ls *.txt | head -n 5
```

`tail` – Print the last part of files

```
$ tail [OPTION]... [FILE]...
```

`tail` prints the last 10 lines of each **`FILE`** on standard output (it is thus a mirror version of `head`). If no **`FILE`** arguments are provided (or when **`FILE`** is **`-`**), `tail` reads standard input. Using the option **-n** or **--lines=** with an integer value **k**, print the last **k** lines. If you provide the option **--lines=+k**, head prints everything starting from the **k**th line.

cat – Concatenate files and print

```
$ cat [OPTION]... [FILE]...
```

cat concatenates the **FILE**s passed as arguments, and prints the result on standard output. cat with a single **FILE** argument is a quick way to print the entire content of that file on standard output:

```
$ cat file1
```

Often, cat is used with an output redirection to combine a number of files into a new file:

```
$ cat file1 file2 > file_combined.txt
```

Without **FILE** arguments, or with the argument '**-**', cat reads from standard input. This can be used to combine the content of existing files with the output of a command. For example, the following pipeline inserts the output of ls between two existing files prologue.txt and epilogue.txt:

```
$ ls | cat prologue.txt - epilogue.txt > my_directory.txt
```

cat provides a number of options to decorate the output in various ways, e.g. by adding line numbers, for which we refer to the man page.

sort

```
$ sort [OPTION]... [FILE]...
```

The sort command sorts and then prints the lines of all input files (or standard input). By default, lines are sorted alphabetically, but other options are available.

-r, --reverse Reverse the sorting order.

-n, --numeric-sort Sort according to the numerical value.

-f, --ignore-case Treat lower case characters as upper case when comparing.

`sort` can also work with tabular data, i.e. text data where each line consists of different fields. With the option **-k f1,f2** (or **--key=f1,f2**), for numbers **f1** and **f2**, `sort` uses the contents of fields **f1** up to **f2** to sort the lines (**f2** can be omitted, in which case the contents of all fields starting from **f1** up to the end of the line is used). We refer to the official documentation for more complex key specifications and options.

By default, fields are separated by space or tab characters. With the option **-t SEP** (or **--field-separator=SEP**), `sort` uses the character **SEP** to delimit the fields.

Example: an alternative way to sort the output of `ls` by file size, using `sort` on numeric value of the fifth field:

```
$ ls -l | sort -k 5,5 -n
```

This produces similar[4] output to the following command, which uses the built-in **--sort** option of `ls`:

```
$ ls -l --sort=size --reverse
```

.

uniq – Detect repeated lines

```
$ uniq [OPTION]... [INPUT [OUTPUT]]
```

`uniq` reads lines from the file '**INPUT**', and prints to the file '**OUTPUT**'. If two or more consecutive lines are identical, only one repetition is printed. **INPUT** or **OUTPUT** are optional arguments, and may be replaced by standard input or output instead. Some extra options are:

[4] Small differences arise from the combination of alphabetical and numerical sorting. To exactly reproduce the output of `ls`'s built-in **--sort** option, use the following slightly more complex `sort` pipeline: `ls -l | sort -k 5n,5 -k 9r,9`

-c Print the number of repetitions of each line.

-d Only print duplicated lines.

-u Only print those lines which are not repeated.

-i Ignore case when comparing lines.

Note that uniq only detects **consecutive** identical lines. Therefore, a classic technique is to first move identical lines together using sort, and then filter the output with uniq, as in the following example:

```
$ sort somefile.txt | uniq
```

or, if you want to see unique lines of the output of a command:

```
$ COMMAND | sort | uniq
```

cut – Extract parts of text lines

cut is a useful tool to extract parts of structured text.

```
$ cut OPTION... [FILE]...
```

cut takes its input from (a list of) files, or standard input. Depending on the chosen **OPTION**, cut extracts either fixed ranges of characters, or fields:

-c LIST Print the character ranges specified in **LIST**.

-f LIST Treat lines of text as table rows with different fields, and print the fields from the given **LIST**. By default, the TAB character marks the end of a field, but you can choose a different field separator using the option **-d**.

-d CHAR In combination with the **-f** option: set the field separator to the right character of choice, if fields are not separated by TAB.

The format of the **LIST** argument for options **-c** and **-f** is the same: a list of one or more integers **i** or integer ranges **i-j**, separated by commas. For example, the following command prints characters 1, 3, 4 and 5 of each line of the file input.txt:

```
$ cut -c 1,3-5 input.txt
```

The following command prints columns 3 and 5 of a file with columns separated by a single space:

```
$ cut -d ' ' -f 3,5 columns.txt
```

grep – print matching lines

```
$ grep [OPTION]... PATTERN [FILE...]
```

`grep` searches given input files (or standard input if no files are given) for a pattern and prints the matching lines. By default **`PATTERN`** is interpreted as a basic regular expression, the option **`-E`** enables extended regular expressions. Typical uses are

- searching for files that contain certain words, or
- filtering the output of commands (or long files) to print only lines containing specific information.

`grep` provides many options to configure what is matched and what gets printed. Some of the most commonly used options are:

`-E, --extended-regexp` Interpret **`PATTERN`** as an extended regular expression.

`-i, --ignore-case` Allow matches with characters that have a different (upper/lower) case.

`-v, --invert-match` Select those lines which **do not** match the pattern.

`-r, --recursive` Recursively search all files under each subdirectory, if one or more of the **`FILE`** arguments is a directory.

Example: list files in the current directory which were last modified in March or April.

```
$ ls -l | grep -E 'Mar|Apr'
```

It is often a good idea to enclose the regular expression **`PATTERN`** in single quotes. Otherwise, the shell might interpret special characters (such as parentheses, the asterisk or the pipe symbol), instead of passing the literal **`PATTERN`** to `grep`.

5 Variables

read – Read variables from standard input

```
$ read [NAME ...]
```

The built-in shell command `read` takes a list of variable **NAME**s as arguments. It reads a line of text from standard input, splits it into a list of words, and assigns each word to the corresponding variable (if the line contains more words than there are variable arguments, the entire remaining part of the input line is assigned to the last variable name).

`read` can be used to get user input in interactive scripts, but it can also be used to process text output. A common idiom is to combine `read` with a `while` loop. For example, the following code successively assigns each line of output from `ls -la` to the variable `LINE`, and runs further commands to process each line.

```
ls -la | while read LINE
do
    # sequence of commands that work on $LINE
    ...
done
```

export – Create environment variables

```
$ export name[=value]
```

The `export` command creates an environment variable with the given **name**. If **name** is the name of an existing shell variable, its value is preserved. To create the variable and set its value with a single command, include the optional **=value**.

`set` – List shell variables

If you run `set` without any further arguments, it prints a list of all variables and functions defined in the current shell. `set` can also change various options that affect the behaviour of the shell. We refer to the Bash manual for these advanced options.

`printenv` – List environment variables

The command `printenv` prints all current environment variables and their values.

6 Index