A Unix Cook-Booklet

Pekka Janhunen, v1.0, 24 April 1999

This is a short beginner's guide to practical using of Unix. I have used the principle that if I don't remember something without checking, I don't write it here. The man pages and other documention is there for definitive information.

Contents

1 Finding information

1.1 Moving around and listing files (cd,ls)

By typing `ls` the files in the current directory are listed. Useful variants are `ls -a` (list also files whose name starts with a dot, since these are normally hidden), `ls -l` (list in long format) and `ls -sF` (show file size and type).

The command `pwd` prints the current directory. The directory can be changed by `cd` (change directory). For example, `cd /usr/bin` changes to directory `/usr/bin`. Just typing `cd` returns to the user home directory. The home directory can be found by typing `pwd` in the home directory, or by `echo $HOME`. In C-shell the tilde character also denotes the home directory.

Filenames that start with `/` are absolute filenames, such as `/bin/sh`. Other names are relative to the current directory. The dot denotes the current directory and two dots the parent directory (the directory one level up). For example, to cd one level up and then down in directory `tex`, one can do `cd ../tex`.

1.2 Locating files by name

The command `locate` (or `glocate`) finds files with a given name, or part of a name. The command relies on a database and therefore works quickly. Very recent files are not found, since the database is typically updated every night.

If not sure about `locate`, one can use `find`, which actually goes through the directory structure:

```
find . -type f -name '*myfile*' -print
```

The first argument is the directory to go through, the current directory `.` in this case. The search is limited to proper files only (usually not necessary). The filename pattern can contain wildcards such as * and ?. Quoting is necessary to prevent expansion by the shell. `-print` is necessary, otherwise nothing is shown.

On some systems commands analogous to `locate` exist and are named `whereis` or `rfind`.

1.3 Looking inside files

To search for a string in a single file, or a group of files that can be named by wildcards, one uses `grep`, for example

```
grep Janhunen *.tex
grep -i janhunen *.tex
grep -in '^c.*mrqmin' *.f *.h
```

In the first case all lines containing Janhunen are printed in all TeX/LaTeX source files in the current directory. In the next one, lower and uppercase letters are treated the same (ignore case, `-i`). The last example prints all Fortran comment lines (starting with `c`) that contain `mrqmin`, ignoring case and printing out line numbers (`-n`). The hat denotes the beginning of a line and `.*` stands for any number (`*`) of any characters (`.`) in between.

`Grep` uses regular expressions, thus characters such as ., *, [and] and some others have special meaning. `Fgrep` is a variant that does not use regular expressions but a plain string. Use `fgrep` if you don't need regular expressions.

The GNU/Linux version of `grep` recognizes options such as `-3` which print out 3 lines surrounding the match also. This is often very useful.

To search in multiple directories one can do e.g.

```
grep -in janhunen *.tex */*.tex */*/*.tex
find . -type f -name '*.tex' -exec grep -in janhunen {} ";" -print
```

After `-exec`, `find` takes arguments to the executed command (`grep`), until a semicolon appears. The semicolon must be quoted to prevent interpretation by the shell. The {} passes the current filename to `grep`.

Often one would like to see or say something about the contents of an unknown file. If the file is a text file, one can do `more myfile` or `less myfile` to view it page by page, or to do `head -20 myfile` to display the first 20 lines, or `tail -20 myfile` to show the last 20 lines, or `cat myfile` to show it all, if the file is short. (To find the size one does `ls -l myfile`.)

A binary file is more difficult, but there is a command `strings` which prints out all printable substrings from the file's binary byte stream. This is very useful for finding out e.g. which binary program produces a given error message, for example.

The command `file` tries to guess the 'type' of a binary file the best it can. For example, it can be used to find out whether a byte stream read from a tape is in `tar` format or perhaps gzipped tar format.

Finally, one can take the file in the GNU `emacs` editor for viewing. One exits from `emacs` by typing `Control-x Control-c`.

1.4 The man-pages

The system manual pages are read by `man`, for example `man man` or `man ls`. `man -k keyword` lists those man-pages whose header contains the given keyword. The man-pages are grouped in numbered sections, for example all user commands are in section 1 and C-callable system calls in section 2. If the same name appears in multiple sections, `man` may show only the first one. In such a case one has to give `man` the section number also, for example, `man 3 printf` shows the C-function printf, while (on my Linux system) `man printf` would show the GNU command line version of printf. `apropos` is the same as `man -k` and `xman` is a version with graphical user interface.

To turn man-pages into normal text files (printable form) pipe the output to `col -b`. For example,

```
man awk | col -b | grep -i -3 aho
```

searches for the substring 'aho' in the `awk` man page, printing also 3 surrounding lines around the match (requires GNU version of `grep`).

1.5 Which command is executed

The command `which`, which is present in most modern shells, prints out which command is actually executed. For example, `which ls` might print out `ls is /bin/ls`. The commands are searched for in $PATH environment variable. To find its value, do `echo $PATH`. To add your own directories in PATH, do

```
setenv PATH ${PATH}:$HOME/bin          # in C-shell (csh,tcsh)
PATH=${PATH}:$HOME/bin; export PATH    # in Bourhe-shell (sh,ksh,bash)
export PATH=${PATH}:$HOME/bin          # simpler, works in ksh and bash
```

To find out which shell you are using, do `echo $SHELL`. You can set such definitions in your startup files. Csh and tcsh read `~/.login` once when you log in and `~/.cshrc` every time a new shell is started. Environment variables settings should in general be gathered in `.login` while aliases and shell variables (the `set` command) should rather be put in `.cshrc`. This is a rough guideline only and depends on personal taste. The Bourne-shell derivatives (sh,ksh,bash) read `~/.profile`.

2 Manipulating files

A Unix file is a sequence of 8-bit bytes. Files have names and they are organized in directories. In this section we show how to manipulate files without looking at their contents.

2.1 Putting files in directories

`mkdir dirname` makes a new directory and `rmdir dirname` removes it. `Rmdir` agrees to remove an an empty directory only. `cp` copies files, `mv` moves them and `rm` removes:

```
cp fn fnnew               # copy a file, new name is fnnew
cp fn ../tex/fnnew        # copy, placing the copy in a different directory
cp fn1 fn2 fn3 ~/dir      # copy fn1,fn2,fn3 in dir, which is under home directory
cp -r dir1 dir2           # copy whole directory recursively
cp -r -p dir1 dir2        # preserve permissions etc.
mv fn fnnew               # rename fn to be fnnew
mv fn1 fn2 ... dir        # move files to directory
mv dir1 dir2              # move directory (SAME FILE SYSTEM)
mv dir1 dir2 ... dirN     # move directories (SAME FILE SYSTEM)
rm fn                     # remove a file
rm fn1 fn2                # remove the named files
rm -r dir                 # remove directory recursively (all its contents)
```

Thus, if `cp` and `mv` are given more than 2 arguments, the last one must be an existing directory. Generally `mv` refuses to move entire directories across file systems, because this would involve physically copying them and then removing.

The `cp, mv, rm` commands can be given the `-i` option to inquire the user about any overwrite. On some systems this is the default (`alias cp 'cp -i'` has been done in C-shell; on Bourne-Again Shell `bash`, `alias cp='cp -i'`). The `-f` flag can be used to force the commands through without asking questions.

Any of the commands `cp, mv, rm` can remove or overwrite data permanently. Either use the `-i` option or think twice before hitting Return.

2.2 Symbolic links

Symbolic links are created by `ln -s truefile linkname`. Symbolic links can point to across file systems without restrictions, and they can point to directories as well as files. Symbolic links are handy for example if you use LaTeX to create your papers and seminar talks and want to reuse the same figures. LaTeX finds the figure files easily if they are in the current directory, and typically you keep files with a single talk in its own directory. Without symbolic links you would have to copy all the figure files in each talk directory, wasting disk space. With symbolic links, only one copy is retained, the others are just links pointing to it.

A symbolic link is removed by `rm`. It removes the link, not the file it points to.

Symbolic links pointing to directories are handy for "emulating" different directory structures, to support software that expects to find some files in a certain place which does not exist on this system.

Hard links are generated by `ln` without the `-s` option. They are not much used nowadays because they can point only to files, not directories, and not across file systems. Usually one is better off using symbolic links instead of hard links.

2.3 Tarfiles and gzipped files

To create a tarfile of a directory, do

```
tar cf tarfile.tar dir            # tar create file
```

All the files in `dir` are recursively put in the tarfile. To extract the contents of an existing tarfile in the current directory, do

```
tar xf tarfile.tar                # tar extract file
```

To view the contents of a tarfile without actually extraing it, do

```
tar tvf tarfile.tar               # see contents only
```

`Tar` is also used to copy directory recursively. One could also do this by `cp -r` (see 2.1 (cp -r)above), but then the permissions, modification times etc. are reset to new values (though `cp -p` should prevent this, but it may not prevent it in every aspect). Anyway, `tar` is a foolproof way to make an exact copy of a directory in every aspect:

```
cd                                               # cd to home directory
cd ..                                            # go one level up
tar cf - pjanhune | (cd /work/pjanhune; tar xf -)
```

This copies the directory `pjanhune` (the home directory of user pjanhune) to a new place, under `/work/pjanhune`. It thus creates `/work/pjanhune/pjanhune`. The `-` stands for standard output in the first `tar`: the tarfile is written to stdout. The stdout is piped to the subshell command in parentheses. This subshell first changes to `/work/pjanhune`, then unpacks the tarfile read from stdin. The result is a copy of directory without using temporary disk space.

The option `v` makes `tar` verbose, it lists what it is doing. Sometimes the `B` option may be necessary when `tar` is reading from a pipe (on IRIX system at least); it may also help when reading from a tape tarfile written using a different system.

The program `gzip` gzips (compresses) a file:

```
gzip file              # compress file, producing file.gz (and delete old)
gzip -9 file           # try heavily, use longer time
gunzip file.gz         # uncompress file.gz, produce file (and delete file.gz)
gunzip <file >newfile  # uncompress from stdin to stdout, not deleting anything
```

Gzipped tarfiles are common and typically end with `.tar.gz`. A shorter convention `.tgz` is common in Linux world. The GNU/Linux version of `tar` can do the gzip transparently by using the `z` option:

```
tar czf tarfile.tgz dir     # create a gzipped tarfile
tar xzf tarfile.tgz         # extract from a gzipped tarfile
```

2.4 Backups and such

To make a backup on a local tape device, one normally uses

```
cd ~/..                    # cd to one level up from home dir
tar cf /dev/tape pjanhune
```

The tape device name may vary, it might also be `/dev/rmt0`, for instance. You need write permission to it, which is typically done by adding yourself to group “tape” or “disk” in `/etc/group`, or by giving enough permissions to everyone for `/dev/tape`: `chmod uog+rw /dev/tape`.

A backup is restored using `tar xf`. Be extremely careful NOT to use `tar xf` inadvertently. If you have a tape in your drive, you could easily anti-backup your files (i.e., lose your recent changes) by doing this! Remember that `c=Create New Tarfile`, `x=eXtract Old Tarfile`.

If the tape drive exists on a different machine, you can do one of the following:

```
(do cd ~/.. first)
tar cf - pjanhune | ssh -l pjanhune rem.fmi.fi 'dd of=/dev/tape'
tar cf - pjanhune | rsh -l pjanhune rem.fmi.fi 'dd of=/dev/tape'
tar cf rem.fmi.fi:/dev/tape pjanhune
```

In the first case the tarfile is piped to a Secure Shell command, which logs in as pjanhune on rem.fmi.fi, invokes the Direct Data transfer (`dd`) on rem.fmi.fi, writing output file to `/dev/tape`. In the second case one uses the old `rsh` (remote shell) instead of `ssh`. `Rsh` does not encrypt the traffic. In the third case one relies on the `rmt` (remote magnetic tape) daemon running on rem.fmi.fi.

One may have to set the buffer size for `tar`. See `man tar` to find out how. One may also have to set the buffer size for `dd` using `bs=8192` or something similar.

On same systems, writing to tape using `dd` does not work properly (then probably neither does `cat`) but one has to use `tar`. If also the `rmt` daemon is not running, probably for security reasons, then one has to use NFS (network file system) to `mount` the directories to rem.fmi.fi before the backup can succeed. Using NFS for backups probably uses more network resources than using the other methods.

It is not usually a good idea to use gzip with tape backups. If a single bit error occurs, it can corrupt the whole backup. Plain `tar` format is more tolerant to errors, in the best case only one file is damaged by a tape bit error.

2.5 Remote copying

To copy files or directories from machine to machine one can use `ssh, scp` (or the old, less safe `rsh, rcp`). To use `scp`, the `sshd` daemon must be running on the remote machine. `sshd` typically resides in `/usr/local/sbin` or `/usr/sbin`, use `locate` or `find` to find out. If typing the password each time is no bother, this is all that is needed. For example,

```
scp -r dir rem.fmi.fi:/tmp/pjanhune    # copy dir to rem.fmi.fi
scp -r dir pj@rem:/tmp/pj              # log in as user 'pj'
```

If too lazy to type the password each time, one has to add the local machine's `~/.ssh/public_key` to the remote machine's `~/.ssh/authorized_keys` file. Or, one can add the local machine's name and username in the remote machine's `~/.shosts` file, which is less safe. One can also add it to `~/.rhosts`, which enables `rcp` to work and is even less safe. Check `man ssh` for details.

2.6 File systems

Directories are divided in file systems, which usually correspond to physical disk units, but nowadays may also be completely virtual. To see the file systems use `df -k` or `mount`.

The command `mount` is used to mount local and remote (NFS) file systems. The configuration file is `/etc/fstab`. `mount -a` examines `/etc/fstab` and tries to mount everything defined there. `umount` is used to unmount file systems. For example, a CD-ROM must be mounted/umounted this way.

2.7 Permissions and ownership

A Unix file and directory has read, write and execute (rwx) permission bits for user (u), group (g) and others (o). To publish your `~/public_html`directory for the Web server or other people to see, do

```
cd
chmod -R og+r public_html
```

The `-R` means recursive, and others (o) and group (g) are given r access to everything under `public_html`. This is not enough, you must also add execute permission to every subdirectory:

```
find public_html -type d -exec chmod og+x {} ";"
```

If you do `chmod -R og+rx public_html`, the execute permission is also added to every file, which is not dangerous but a bit ugly.

If you are the root, you can change the owner and group of files using `chown` and `chgrp`. Modern versions of `chown` allow to set the group simultaneously, like

```
chown -R pjanhune.luser .
```

changes all files owner to be pjanhune and group to by luser ("local user") under current directory and all its subdirectories.

3 Processing files

3.1 Textual processing (awk, sed, tr)

The command `awk` (Aho, Weinberger, Kernighan) contains pattern matching an C-like language to process text files line by line and field by field. For example, to print the second column (columns are by default any whitespace-separated substrings), do

```
awk <inputfile '{print $2}' >outputfile
```

. The string passed to `awk` usually consists of pattern-program pairs. To sum the numbers in the first column, ignoring lines starting with '#', do

```
awk 'BEGIN {s=0} /^#/ {next} /.*/ {s+=$1} END {print s}' <inputfile >outputfile
```

The special patterns BEGIN and END are associated before starting and after ending the processing itself. The patterns are regular expressions, see `man regexp` on most systems.

The `sed` is a stream editor which can substitute strings. For example, to rename each .cpp file into .C file in the current directory, do

```
for i in *.cpp                # (Bourne-shell)
do
mv $i `echo $i | sed -e 's/\.cpp/.C/' | tr -d '\012'`
done
```

`Echo` writes the current filename.cpp to stdout for further processing by `sed` and `tr`. The `sed` substitute command is `s/oldstring/newstring/g`, however, `oldstring` is a regular expression and thus the dot is preferrably quoted. If the `/g` is given, multiple replacements are allowed (not needed in the above example).

The final character translate (`tr`) command removes the newline characters. This is perhaps not absolutely needed in most systems because the shell may take care of it, but is a good idiom to learn.

The above example works in Bourne-type shells (sh, ksh, bash). In C-shell (csh, tcsh) there is a simpler way, and the syntax of the for-statement is different:

```
foreach i (*.cpp)              # (C-shell)
mv $i $i:r.C
end
```

The `:r` "goes backward" one dot-separated word, thus if $i is `myfile.cpp`, $i:r becomes `myfile` and $i:r.C is `myfile.C`.

One can also use `basename` to extract the 'type' field out of filenames.

4 Using shells

4.1 Bourne-type shells

There are two main flavours of shells, the Bourne-shell (sh, ksh, bash) and the C-shell (csh, tcsh). An example of a Bourne-shell script is `tofloppy`:

```
#!/bin/sh

gzipped="false"
dev="/dev/fd0"
if [ $# -eq 1 ]; then
    case ".$1" in
    .*.gz|.*.tgz)
        gzipped=true ;;
    .*) ;;
    esac
fi

if [ ".$1" = ".-9" ]; then
        gzarg='-9'
        shift
else
        gzarg='-5'
fi

if [ "$gzipped" = "true" ]; then

echo "This will put the gzipped item $1 to the floppy disk."
echo 'Insert the floppy disk and press Return when done.'
read ans
cat $1 | dd bs=8192 of="$dev"

else

echo "This will put items $* to the floppy disk."
echo 'Insert the floppy disk and press Return when done.'
read ans
tar cfv - $* | gzip $gzarg | dd bs=8192 of="$dev"
```

```
fi

echo 'tofloppy ok. See fromfloppy [-t] for reverse operation.'
```

Put this in directory `bin` under your home directory (do `mkdir bin` if it does not yet exist), naming the file `tofloppy`, and give execution permission: `chmod +x tofloppy`. Then ensure that `$HOME/bin` is in your PATH (see 1.5 (PATH)). After this, tofloppy is available as a command (after doing `rehash` if using csh or tcsh). `Tofloppy` is given a list of file or directory names; it makes a gzipped tarfile out of those and writes it to your floppy disk. The option `-9` is recognized, it tries to compress harder. For user convenience, if only one argument is given and if it ends with `.gz` or `.tgz`, it will be written directly to disk, without tarring and compressing it first, since it is already in compressed form.

The used shell is specified in the first line reading `#!/bin/sh`. Since this line starts with #, the shell treats it as a comment, but the Unix kernel uses the information to execute the file using the program `/bin/sh`. The program name must be a full, explicit pathname here. This is sometimes a nuisance for portability, if the pathname is different on different system. One can trust that `sh` is always found in `/bin`, this is one reason why I prefer Bourne shell scripts.

A corresponding program `fromfloppy` looks the following:

```
#!/bin/sh

opts="xvfo"
showingOnly=false
if [ $# -eq 1 ]; then
    if [ "$1" = "-t" ]; then
       opts="tvf"
       showingOnly=true
    fi
fi

if [ "$showingOnly" = "true" ]; then
    echo "This will show the directory on your floppy disk."
else
    echo "This will extract the files from your floppy disk."
    echo '(Use fromfloppy -t to see the directory only.)'
fi

echo 'Insert the floppy disk, press Return when done.'
read ans
dd bs=8192 if=/dev/fd0 | gunzip | tar $opts -
echo 'fromfloppy ok. See tofloppy for reverse operation.'
```

`fromfloppy` reads a compressed tarfile stream from the floppy disk (it prompts the user to press Return first) and unpacks the files in the current directory. `fromfloppy -t` only shows what files are there without actually extracting them.

A summary of the shell syntax can be found in the man pages. The only problem is that the man pages are very long. Thus it is better to grab them in a text file first, by doing e.g. `man sh | col -b >mansh` and viewing `mansh` in your favourite text editor. In `bash`, a quick help is available by typing `help`.

The oldest of the shells is the original Bourne-shell, `sh`. It does not provide interactive sugar such as command line editing or TAB completion (command and filename completion by pressing TAB). The Korn-shell `ksh` is a somewhat-extended version which is the default on HP workstations, for instance. The GNU version is called Bourne-again shell (`bash`). It provides many nicety features while being compatible with the original Bourne-shell and is often the default shell on Linux systems. On Linux, `/bin/sh` is a symbolic link to `/bin/bash`.

4.2 C-shell

The C-shell has a different syntax for if-statements, loop statements and so on. The original C-shell does not have command line editing and is thus not suitable for interactive work. The `tcsh` contains these facilities. I usually use `tcsh` as my interactive shell but `/bin/sh` for my scripts.

4.3 Redirection

A summary of the input/output direction commands in different shells:

```
prg <input >output          # (all shells) stdin from input, stdout to output
prg 2>/dev/null | less      # (Bourne-shell) ignore error output, pipe stdout to less
prg 2>&1 >x                 # (Bourne-shell) stdout and stderr to x
prg >&x                     # (csh,tcsh) stdout and stderr to x
(prg >x) >&y                # (csh,tcsh) stdout to x, stderr to y
```

4.4 Job control in `tcsh` and `bash`

A program can be running in the foreground, in the background, or be in suspended state (stopped but restartable). To run the program in the background one ends the command with &. It is also possible to put a foreground job to the background by first suspending it by pressing `Control-z` and then typing `bg`. Inversely, a background job can be brought to foreground by typing `fg`. Thus it is not necessary to restart the program just because one wants it to run in the background. On modern systems, background jobs are usually not killed if the shell that created them exits.

5 Publishing in the World Wide Web

The HTTP daemon `httpd` uses ~public_html/ as the magic directory which is visible. Each subdirectory there should by good practice contain `index.html`, in particular, `~/public_html/index.html` is the user home page that is referred to as `http://machine-name/~username`, for example `http://www.geo.fmi.fi/~pjanhune`. To limit access in a subdirectory, create a file named `.htaccess` there, with example contents as:

```
<Limit GET>
order deny,allow
deny from all
allow from .fmi.fi
</Limit>
```

The file `.htaccess` must be world readable, otherwise it has no effect! Thus you must do `chmod og+r .htaccess`.