Scientific Computing I

Linux and the Commandline

Jens Saak

Computational Methods in Systems and Control Theory (CSC) Max Planck Institute for Dynamics of Complex Technical Systems

Winter Term 2016/2017

A short History of an Accidental Revolution

First announcement

A short History of an Accidental Revolution

First announcement

Linus Torvalds news posting August 26, 1991

"Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).
I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torv...@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(. "

After this the (r)evolution has been fast as the following time-line shows.

Unix/Linux history (following Wikipedia)

A short History of an Accidental Revolution

Unix/Linux history (following Wikipedia)

http://en.wikipedia.org/wiki/History_of_Linux

1983 Richard Stallman creates the GNU project with the goal of creating a free operating system.

1989 Richard Stallman writes the first version of the GNU General Public License.

1991 The Linux kernel is publicly announced by the 21 year old Finnish student Linus Benedict Torvalds.

1992 The Linux kernel is relicensed under the GNU GPL. The first so called "Linux distributions" are created.

1993 Over 100 developers work on the Linux kernel. With their assistance the kernel is adapted to the GNU environment, which creates a large spectrum of application types for Linux. The oldest currently existing Linux distribution, Slackware, is released for the first time. Later in the same year, the Debian project is established. Today it is the largest community distribution.

A short History of an Accidental Revolution

Unix/Linux history (following Wikipedia)

http://en.wikipedia.org/wiki/History_of_Linux

1994 In March Torvalds judges all components of the kernel to be fully matured: he releases version 1.0 of Linux. The XFree86 project contributes a graphic user interface (GUI). In this year the companies Red Hat and SUSE publish version 1.0 of their Linux distributions.

1995 Linux is ported to the DEC Alpha and to the Sun SPARC. Over the following years it is ported to an ever greater number of platforms.

1996 Version 2.0 of the Linux kernel is released. The kernel can now serve several processors at the same time, and thereby becomes a serious alternative for many companies.

1998 Many major companies such as IBM, Compaq and Oracle announce their support for Linux. In addition a group of programmers begins developing the graphic user interface KDE.

A short History of an Accidental Revolution

Unix/Linux history (following Wikipedia)

http://en.wikipedia.org/wiki/History_of_Linux

1999 A group of developers begin work on the graphic environment GNOME, which should become a free replacement for KDE, which depended on the then proprietary Qt toolkit. During the year IBM announces an extensive project for the support of Linux.

2004 The XFree86 team splits up and joins with the existing X Window standards body to form the X.Org Foundation, which results in a substantially faster development of the X Window Server for Linux.

2005 The project openSUSE begins a free distribution from Novell's community. Also the project OpenOffice.org introduces version 2.0 that now supports OASIS OpenDocument standards in October.

2006 Oracle releases its own distribution of Red Hat. Novell and Microsoft announce a cooperation for a better interoperability.

A short History of an Accidental Revolution

Unix/Linux history (following Wikipedia) http://en.wikipedia.org/wiki/History_of_Linux

- 2007 Dell starts distributing laptops with Ubuntu pre-installed in them.
- 2011 Version 3.0 of the Linux kernel is released.
- 2012 2012: the aggregate Linux server market revenue exceeds that of the rest of the Unix market.
- 2013 Google's Linux-based Android claims 75% of the smartphone market share, in terms of the number of phones shipped.
- 2014 Ubuntu claims 22,000,000 users.

`bash` and its Basic Helpers

bash

bash and its Basic Helpers

bash (Bourne Again Shell)...

...is a Unix command interpreter that usually runs in a text window where the user can type in commands. It can also be used to read commands from files, the so called *scripts*. It extends the capabilities and command set of the classic Bourne shell sh.

bash and its Basic Helpers

bash (Bourne Again Shell)...

...is a Unix command interpreter that usually runs in a text window where the user can type in commands. It can also be used to read commands from files, the so called *scripts*. It extends the capabilities and command set of the classic Bourne shell sh.

Stephen Bourne and sh

- Inventor of the sh shell in 1977
- sh used to be the standard shell for Unix-7 systems
- sh is still available on most Unix(-like) systems today

Special Characters

bash and its Basic Helpers

Special Characters

*	serves as a placeholder for arbitrarily many characters
?	a placeholder for a single character
/	directory separator
\	escape character for quoting special characters and to mark line-breaks
~	abbreviation for your home directory
|	the pipe operator: connects two simple commands to a new one by redirecting the output of the one on the left to the other one on the right. || represents a logic OR.
<	fetches the input for a command (on the left) from a file or device (on the right)
>	redirects the output of a command (on the left) to a file or device (on the right)

bash and its Basic Helpers

Special Characters

2>	same as above for the error output only, can be used to redirect the standard error messages to standard output so it is recognized by the > and | as well via 2>&1
1>	same as above for the standard output without the errors
>>	as > but appends the output instead of overwriting the file
$	used in command substitution and for referring to shell and environment variables
&	a single & after a command name sends the execution to the background. Double && stand for the logic AND.

bash and its Basic Helpers

Special Characters

`	accent grave is used for command substitution
'	single quotes removes the special meaning of all special characters enclosed by them.
"	double quotes act the same as single quotes with the exception of the $, `, \ (and sometimes !) characters keeping their special properties.
blank	the simple blank is used to separate words and thus needs to be escaped when, e.g., a file name contains it.
#	comment character; everything following this character on the same line will be dropped

Basic Command for Managing Files

`bash` and its Basic Helpers

Basic Command for Managing Files

`pwd` short for print working directory, and printing the name of the directory you are currently working in is exactly what it does.

`cd` change directory, switches the current working directory to the directory given as the argument. If no argument is given `cd` takes you home, i.e., switches to your users home directory.

`mkdir` creates a new directory in the current working directory

`rmdir` removes the directories specified as arguments if they are empty.

`touch` creates an empty file or sets the access date of the file to the current time and date if the file exists

`rm` removes files. It can also be used to remove directories with the `-r` (recursive) option. This is especially useful when `rmdir` does not work since the directory is not empty. The `-f` (force) option can be used to remove even protected files.

`bash` and its Basic Helpers

Basic Command for Managing Files

`ls` lists all files in the directory specified. If none is specified the current working directory is used. If the argument is a file or a list of files only those files are listed. Useful options are `-l` for a full listing including access rights and ownership information, `-a` for a listing including also hidden files. The `-h` option in combination with the two previous ones makes file sizes human readable, i.e., displayed as multiples of kB, MB, GB, TB, where all of these are representing powers of 1024. If a 1000 based presentation is desired `--si` needs to be used instead.

`cp` takes two or more arguments and copies the n-1 first arguments to the last. If more than 2 arguments are given the last one must be a directory. Absolute and relative paths are allowed.

`mv` Same as above but moves the files, i.e., the originals are removed after the copy is successful.

`bash` and its Basic Helpers

Basic Command for Managing Files

`ln` links files to new names. By default a hardlink is created. Then the new name serves as a new entry in the file system associated to the same data and the data is only removed if all hardlinks are removed. When used with the -s option a softlink is created that only points to the original. When the original data is removed the link becomes orphaned.

`find` find is a powerful search tool that can hardly be fully described in a few words. We refer to the man and info pages for details.

`locate` Another search tool that uses a pregenerated database for the searches. The database may be restricted to parts of the filesystem only, or even not exist. Also it is frequently updated but may be outdated when the actual search is request. However, for directories that do not change very frequently this is a good alternative since it is a lot faster than find usually.

Working with Files

Access Permissions, Disk Usage, and Quotas

Working with Files

Access Permissions, Disk Usage, and Quotas

`chmod` change the file permissions, i.e., access permission for reading, writing, and executing a file based on user, group and world privileges.

`chown` change the associated owner of a file (usually require administrator privileges).

`chgrp` change the Unix-group associated to a file

`df` generates a list of all available filesystems in the machine and their occupation statistics.

`du` prints the disk usage of the argument, which can be either a (list of) single file(s), or directory(ies), where the later will be searched recursively.

`quota` tells the user how much of his/her quota, i.e., the maximum allowed disk usage is used already.

Viewing, Compressing, and Identifying Files

Working with Files

Viewing, Compressing, and Identifying Files

`less`	print the content of a text file to the screen.
`cat`	redirect the content of a file to another one.
`watch`	keep track of the changes in a file.
`tail`	view the end of a file (-n number of lines to show)
`head`	view the beginning of a file (-n number of lines to show)
`diff`	compare 2 files
`zip/unzip`	compress and uncompress files into or from zip-archives.
`tar`	create tape archives of files by gluing them into one large entity. Can use additional compression (gzip, bzip2)
`file`	identify the file type by examining the file instead of relying on file-extensions.

Manipulation of Simple Commands

The Pipe Operator

Manipulation of Simple Commands

The Pipe Operator

The pipe operator `|` in the Linux shell can be employed to directly use the output of one program as the input for another one. A statement of the form

```
program1 | program2
```

can be used if `program1` writes its output to the standard output and `program2` reads its input from the standard input.

Manipulation of Simple Commands

The Pipe Operator

The pipe operator `|` in the Linux shell can be employed to directly use the output of one program as the input for another one. A statement of the form

```
program1 | program2
```

can be used if `program1` writes its output to the standard output and `program2` reads its input from the standard input.

Problem

Often programs expect a number of input arguments to be supplied and inputs are not read from user interactions via the standard input device.

Example

Remove all `.pdf` files from the working directory and its subdirectories.

- `find` produces the list of `.pdf` files.
- `rm` expects a list of files as arguments, not from user input.

Manipulation of Simple Commands

The Pipe Operatorand the `xargs` utility

Example

Remove all `.pdf` files from the working directory and its subdirectories.

- `find` produces the list of `.pdf`s.
- `rm` expects a list of files as arguments not from user input.

Solution

`xargs` can be used to split a list from standard input into several arguments for another program.

```
find . -name '*.pdf' | xargs -n 1 -P 4 -d '\n' rm
```

Manipulation of Simple Commands

The Pipe Operatorand the `xargs` utility

Example

Remove all `.pdf` files from the working directory and its subdirectories.

- `find` produces the list of `.pdf`s.
- `rm` expects a list of files as arguments not from user input.

Solution

`xargs` can be used to split a list from standard input into several arguments for another program.

```
find . -name '*.pdf' | xargs -n 1 -P 4 -d '\n' rm
```

`-n` number arguments passed in a singe call

Manipulation of Simple Commands

The Pipe Operatorand the `xargs` utility

Example

Remove all `.pdf` files from the working directory and its subdirectories.

- `find` produces the list of `.pdf`s.
- `rm` expects a list of files as arguments not from user input.

Solution

`xargs` can be used to split a list from standard input into several arguments for another program.

```
find . -name '*.pdf' | xargs -n 1 -P 4 -d '\n' rm
```

- `-n` number arguments passed in a singe call
- `-P` maximum number of parallel executions

Manipulation of Simple Commands

The Pipe Operatorand the `xargs` utility

Example

Remove all `.pdf` files from the working directory and its subdirectories.

- `find` produces the list of `.pdf`s.
- `rm` expects a list of files as arguments not from user input.

Solution

`xargs` can be used to split a list from standard input into several arguments for another program.

```
find . -name '*.pdf' | xargs -n 1 -P 4 -d '\n' rm
```

- `-n` number arguments passed in a singe call
- `-P` maximum number of parallel executions
- `-d` set delimiter used for spliting (default: blank)

Redirection Operators

Manipulation of Simple Commands

Redirection Operators: > and >>

Output Redirection

```
program1 > output.txt
```

creates a file `output.txt` in the current working directory and writes all outputs to that file. If `output.txt` already exists it is replaced.

Manipulation of Simple Commands

Redirection Operators: > and >>

Output Redirection

```
program1 > output.txt
```

creates a file `output.txt` in the current working directory and writes all outputs to that file. If `output.txt` already exists it is replaced.

Manipulation of Simple Commands

Redirection Operators: > and >>

Output Redirection

```
program1 > output.txt
```

creates a file `output.txt` in the current working directory and writes all outputs to that file. If `output.txt` already exists it is replaced.

If we want to preserve the existing content of the file

```
program1 >> output.txt
```

can be used to simply append the new data. If `output.txt` does not exist it is created.

Manipulation of Simple Commands

Redirection Operators: <

Input Redirection

To stream the input from a file `input.txt` instead of the standard input we write

```
program1 < input.txt
```

May be used in conjunction with output redirection

```
program1 <input.txt >output.txt
```

or

```
program1 <input.txt >>output.txt
```

Manipulation of Simple Commands

Redirection Operators: Splitting Messages and Errors

There are two special variants of the output operator that allow to separate between standard outputs and error messages.

```
program 1>output 2>errors
```

will create a file `output` containing the standard messages of the program and another file `errors` where all the error messages are stored.

A common application:

```
find / -name search_expression 2>/dev/null
```

redirects error messages due to missing permissions to the “data nirvana”.

Command Substitution

Manipulation of Simple Commands

Command Substitution

Command Substitution Operators

- `` ` ` `` accent grave pairs evaluate the expression enclosed and insert the result.
- `$( )` as above but avoids confusion with single quotes.

Example

Both

```
echo Yeah, today is `date`, the term is almost over!
echo Yeah, today is $(date), the term is almost over!
```

result in

```
Yeah, today is Thu Oct 11 14:33:35 CEST 2016, the term \
    → is almost over!
```

Script File Basics

Script files

Script File Basics

Script Files

If the shell is told to execute a text file it simply interprets the content as shell command unless the file starts with the expression `#!` followed by the full path to an executable.

A file `hellow.sh` with content:

```
echo "Hello␣World!␣"
```

writes

```
Hello World!
```

evaluated in the current shell.

Script File Basics

Replacing the content by

```
#!/bin/bash
echo "Hello␣World!␣"
```

provides the same output, but is evaluated in a newly started `bash` process.

The same is true for a file `hellow.py`

```
#!/usr/bin/python
print("Hello␣World!");
```

which is run in a python process.

Sophisticated `bash` Scripting

`bash` knows several standard control structures as loops and conditionals. See the man page for details.

Simple Automatic File Manipulation

Regular Expressions

Simple Automatic File Manipulation

Regular Expressions

Regular Expression

- strings that can be used to establish complex search and replace operations on other strings.
- consist of a combination of special and basic characters
- used to match (specify and recognize) substrings in a string.

The special characters

$$/, (,), *, ., |, +, ?, [,], \hat{}, \$, \backslash, \{, \}$$

and their actions are described in the following.

Simple Automatic File Manipulation

Regular Expressions

Basic Specifiers and Matches

.	matches any single character except linebreaks
^	matches the beginning of the string/line
$	matches the end of the string/line
[list]	any one character from list. Here list can be a single character, a number of characters, or a character range given with -
[^list]	any one character that is NOT in list.
( )	guarantees precedence of the enclosed expression. (optional)
(re)	matches the expression re
\	escapes, i.e., removes the special meaning of, the following special character.

Simple Automatic File Manipulation

Regular Expressions

Multiplicity Control and Combinations

re?	matches at most one appearance of re.
re+	matches one or more subsequent appearances of re
re*	matches none or arbitrarily many subsequent appearances of re
re{n,m}	matches at least n and at most m subsequent appearances of re. Both n and m can be omitted either with or without the comma. Then n means exactly n matches. n, stands for at least n matches and ,m for at most m matches.
(re1)(re2)	matches re1 followed by re2
re1 | re2	matches either the expression re1 or re2

Simple Automatic File Manipulation

Regular Expressions

Some Simple Examples

a?b	matches a string of one or two characters eventually starting with a but necessarily ending on b
^From	matches a line/string beginning with From
^$	matches an empty line/string
^X*YZ	matches any line/string starting with arbitrarily many X characters followed by YZ
linux	matches the string linux
[a-z]+	matches any string consisting of at least on but also more lower case letters
^[^aA]	any line/string that does not start with an a or A.

grep

Simple Automatic File Manipulation

grep

`grep` is basically used for printing lines in a number of input files matching a given pattern. That pattern can be a simple keyword but also an arbitrarily complicated regular expression.

Examples

```
grep Time logfile
```

If you are not sure whether Time was written with capital T you can use

```
grep -i Time logfile
```

which switches of case sensitivity, or

```
grep [tT]ime logfile
```

as an example for a simple regular expression.

Simple Automatic File Manipulation

grep (Extended Usage)

Recursive Operation

In the case you do not remember which file in your large software project contains the definition of a certain function you can have `grep` search a complete directory recursively

```
grep -r function-name *
```

returning all lines containing `function-name` preceded by the corresponding file name.

Output Negation

You can also negate the output of `grep` by the switch `-v` to suppress printing of all lines that match the pattern.

sed

Simple Automatic File Manipulation

`sed`

`sed` the **S**tream **Ed**itor is a basic text editor that in contrast to the usual text editors (like `vi`, `emacs`, `nano`, ...) is not interactive but uses certain command strings to manipulate the text file streamed into it automatically without user interaction.

Example Scenario

`sed` is especially useful when, e.g., a variable or function (or any other identifier) in a large software project is supposed to be renamed. Consider the name of variable called `complicatedname` is to be replaced by `simplename` for better readability of the code in a large C project.

Simple Automatic File Manipulation

sed

Search and Replace

The search and replace string in `sed` takes the form `s/foo/bar/`.

The incoming stream is searched line by line and every first match of the regular expression `foo` is replaced by `bar`. If we expect more than one possible matches we might want to use `s/foo/bar/g` to replace all of them.

In our example C project the call for the main file might be

```
sed -i 's/complicatedname/simplename/g' main.c
```

To complete the picture we can use `find` to search for all `.c` and `.h` files and execute the above line for every single one of them.

```
find . -name '*.[ch]' -exec sed -i 's/complicatedname/\
    → simplename/g' {};
```

Simple Automatic File Manipulation

sed

Search and Replace

The search and replace string in `sed` takes the form `s/foo/bar/`.

The incoming stream is searched line by line and every first match of the regular expression `foo` is replaced by `bar`. If we expect more than one possible matches we might want to use `s/foo/bar/g` to replace all of them.

In our example C project the call for the main file might be

```
sed -i 's/complicatedname/simplename/g' main.c
```

To complete the picture we can use `find` to search for all `.c` and `.h` files and execute the above line for every single one of them.

```
find . -name '*.[ch]' -exec sed -i 's/complicatedname/\
    → simplename/g' {};
```

Simple Automatic File Manipulation

sed

Inplace Backups

```
sed -i.orig 's/foo/bar/4' filename.txt
```

Copies `filename.txt` to `filename.txt.orig` prior to the manipulation.

Simple Automatic File Manipulation

sed

sed and head

For printing the first 10 lines of `file` like

```
head file
```

we can use

```
sed 10q file
```

as well.

Simple Automatic File Manipulation

sed and grep

We can make sed emulate grep by using a simple search string instead of the replace string.

```
grep foo file
```

can be written as

```
sed -n '/foo/p' file
```

in sed and grep -v is performed by replacing p with !p above.

The example above could as well be written as

```
cat file | sed -n '/foo/p'
```

awk

Simple Automatic File Manipulation

awk

The AWK utility is an interpreted programming language typically used as a data extraction and reporting tool. Its name is derived from the family names of its inventors – Alfred Aho, Peter Weinberger, and Brian Kernighan. The current specifications can be found in the standard IEEE 1003.1-2008[1].

Basic Calling Sequences

It is invoked using

```
awk 'awk-statements' filename
```

to analyze a file. It can also read its input from a pipe:

```
... | awk 'statements'
```

[1] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html

Simple Automatic File Manipulation

awk

awk programs

- Basic format of a rule:

```
Condition { Action }
```

- General awk program

```
Condition1 { Action1 }
Condition2 { Action2 }
...
```

awk knows four types of conditions:

- Expression Operator Expression
- Expression Operator /RegEx/
- BEGIN
- END

Simple Automatic File Manipulation

awk

Expressions

Expression can be one of

- column specifier
- numeric value
- a string enclosed by double quotes

Operators

Operators are the usual comparison or assignment operators

- ==, !=,
- <, >,
- +=, −=,
- ...

Simple Automatic File Manipulation

awk

Consider the following file containing some measured data

```
1 0.02 0.43
2 0.03 1.03
3 0.55 0.30
```

If we want to extract only the second column we invoke `awk` as

```
cat file | awk '{ print $2; }'
```

All rows where the third column is larger than one are returned by

```
cat file | awk '$3>1.0 { print $0; }'
```

Simple Automatic File Manipulation

awk

Consider the following file containing some measured data

```
1|0.02|0.43
2|0.03|1.03
3|0.55|0.30
```

The column separator here is not a space or a tabulator. It can be specified using `FS="Separator"` inside the `BEGIN` action.
Our previous calling sequence

```
cat file | awk '$3>1.0 { print $0;}'
```

changes to

```
cat file | awk 'BEGIN{FS="|";} $3>1.0 { print $0;}'
```