Linux in Science and Engineering

Working and programming in a Linux environment

Andreas Hirczy

February 12, 2026

Contents

1 Who am I?

1.1 And why do I feel qualified to talk about Linux?

- started studying "Technische Physik" in 1986

- started commercial programming in C and C++ in 1988
- got promoted to project lead by 1989 - and got unhappy
- went back to university, started working with Linux
- programming in a scientific setting - ÖAW, Institut für Weltraumforschung
  - ground station telemetry for Cassini Huygens
  - satellite tracking
- got hired bei TU Graz, Institute of Theoretical and Computational Physics to maintain and shepherd a middle sized UNIX and Linux environment of ≈ 130 PCs - UNIX has been retired in the meantime

2 Introduction to this mini course

2.1 Scope

Aim of this short series of talks is to provide an overview on **methods** and **tools** enabling you to **develop and run programs** of medium complexity **on our site** using **Linux**.

Notes are available at `https://itp.tugraz.at/~ahi/VO/Linux_Science_101.html` and as PDF - old notes at `https://itp.tugraz.at/~ahi/admin/Working_Linux.html`.

Examples might be written in *shell* language, *python* and *C*, although the principles should be applicable using most programming languages. Shell in this context always means Bash – there is a whole family of other command shells, usually with minor differences.

Teaching/Learning a specific programming language is explicitly out of scope.

2.2 Parts / Time slots

2.3 Available Topics

- shell, commands and scripting
- version control
- (semi-)automatic build
- debugging tools, debugging strategies and quality assurance
- documentation - when & how to write it

- batch system "HTCondor"
- permanent data storage
- optimization
- computer security

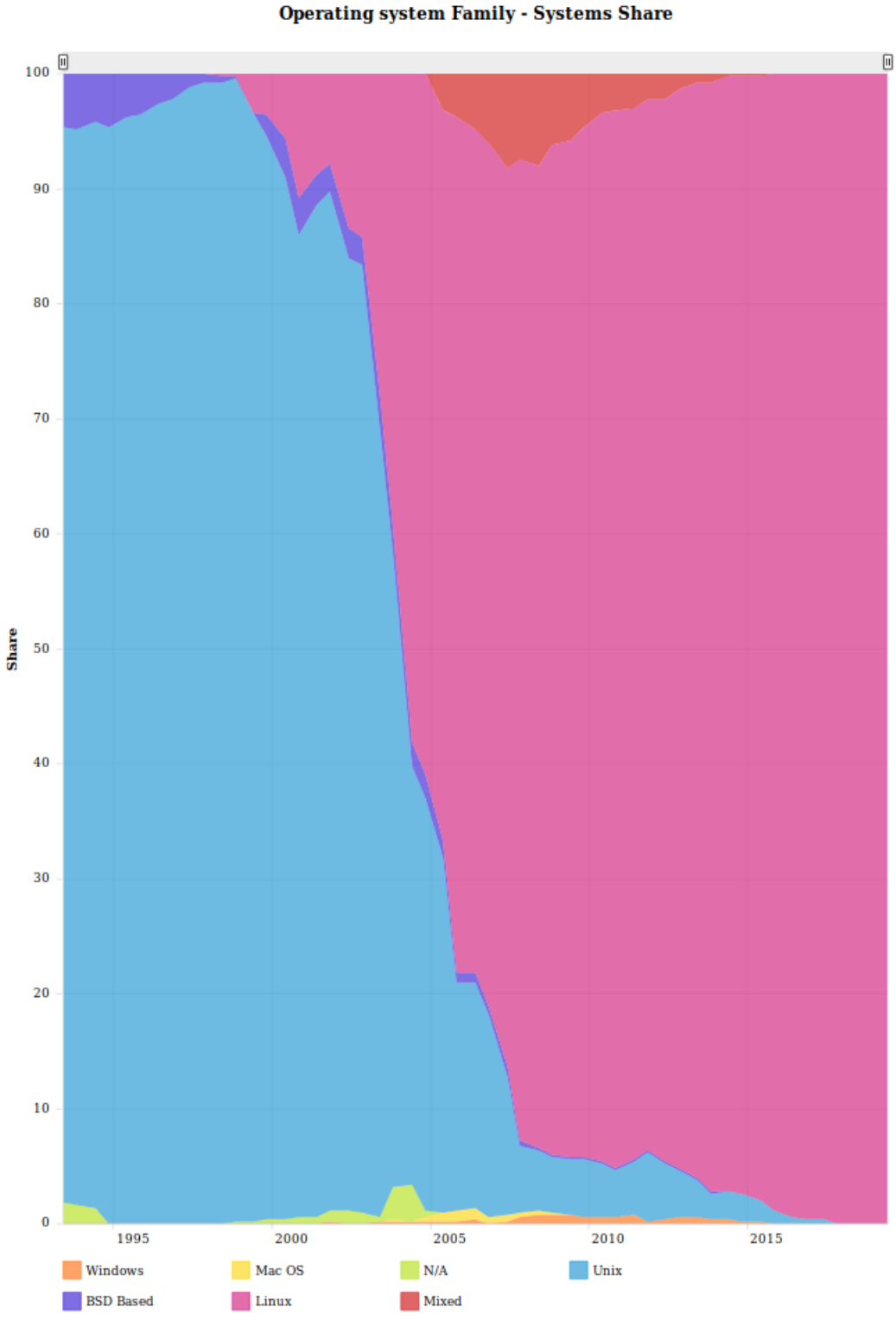
3 Who are You?

4 Introduction to Linux

4.1 Why Linux at all?

Two Reasons!

1. I like it!
2. Windows and MacOS are everywhere!
3. Since we care about **theoretical physics** with a specialisation towards **computational physics** one of our primary targets are supercomputers.
   Since November 2017 100% of the Top 500 supercomputers run Linux.


Operating system Family - Systems Share
Share
100
90
80
70
60
50
40
30
20
10
0
1995
2000
2005
2010
2015
Windows
Mac OS
N/A
Unix
BSD Based
Linux
Mixed

4.2 Unix (and Linux) Philosophy

4.2.1 KISS

KEEP IT SIMPLE, STUPID!

Every tool should fulfill one purpose - and fulfill it as good as possible, adding only as much complexity as absolutely necessary.

4.2.2 Combine orthogonal tools

```
man -t bash | grep ^%%Page: | wc -l
```

4.2.3 Everything is a file

In UNIX everything is a file: network, hard disks, USB sticks, keyboards & mice, ... even the main memory

```
ahi:~ $ LANG=C ls -lh /dev/input/by-id/
total 0
lrwxrwxrwx 1 root root 9 Okt   2 23:40 usb-CHICONY_HP_Basic_USB_Keyboard-event-kbd ->
lrwxrwxrwx 1 root root 9 Okt   2 23:40 usb-Microsoft_Microsoft_Basic_Optical_Mouse_v2
lrwxrwxrwx 1 root root 9 Okt   2 23:40 usb-Microsoft_Microsoft_Basic_Optical_Mouse_v2

ahi:~ $ LANG=C ls -lh /dev/input/event3
crw-rw---- 1 root input 13, 67 Okt   2 23:40 /dev/input/event3

ahi:~ $ LANG=C ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 Okt  2 23:40 /dev/sda
brw-rw---- 1 root disk 8, 1 Okt  2 23:40 /dev/sda1
brw-rw---- 1 root disk 8, 2 Okt  2 23:40 /dev/sda2

ahi:~ $ LANG=C sudo /sbin/fdisk -l  /dev/sda
Disk /dev/sda: 465,78 GiB, 500107862016 bytes, 976773168 sectors
Disk model: ST3500630AS
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x00084420

Device     Boot     Start       End   Sectors   Size Id Type
/dev/sda1  *         2048 960937983 960935936 458,2G 83 Linux
/dev/sda2       960937984 976771071  15833088   7,6G 82 Linux swap / Solaris
```

```
ahi:~ $ ls -lh /dev/mem
crw-r----- 1 root kmem 1, 1 Okt  2 23:40 /dev/mem

ahi:~ $ sudo od -t x1 -a /dev/mem | head -n 4
0000000  f3  ee  00  f0  f3  ee  00  f0  c3  e2  00  f0  f3  ee  00  f0
          s   n nul   p   s   n nul   p   C   b nul   p   s   n nul   p
0000020  f3  ee  00  f0  54  ff  00  f0  2f  20  00  f0  d7  1f  00  f0
          s   n nul   p   T del nul   p   /  sp nul   p   W  us nul   p
```

4.2.4 History

The history and technology of Unix is strongly tied to the programming language "C", which was largely developed to implement the operating system.

AT&T UNIX from 1969 onwards, source available free in the beginning

Berkeley Software Distribution (BSD) Unix from 1977 to 1995, a collection of free derivatives (FreeBSD, OpenBSD, NetBSD, ...) exist till now

commercial UNIX some workstation vendors adopted BSD for their hardware, some used evolved this system later to AT&T *system V* UNIX. The latest offspring is Apples macOS, whose kernel consists of parts of FreeBSD and the "Mach" microkernel.

Linux developed starting 1991 in the spirit of Unix, technically not UNIX since no code has been inherited and no fees have been paid for the UNIX trademark. *Linux* is just the operating system kernel, most of the user visible code comes from the GNU project → so you will often find the term **GNU/Linux**.

5 Using the shell

> I really hate this damn machine
> I wish that they would sell it
> It never does quite what I want
> But only what I tell it
> — Anonymous

The manual page for `bash(1)` is 81 printed pages of rather dense reference, we will just scratch the surface of most topics to demonstrate the possibilities.

Without knowing the following "tricks" you will find the shell complicated and heavy on your typing fingers – learning them will make you wonder, how long you could stand clumsy graphical user interfaces.

5.1 Getting HELP!

Try understanding the principles, you can usually find the details and peculiarities in the man pages:

```
man ls
man man
apropos interfaces
```

References to manual pages are marked as `man(1)`. The number is specifying the manual for the command `man` in section 1 (executable programs or shell commands) of the manual pages, instead `man(7)` of section 7 (Miscellaneous - including macro packages and conventions)

5.2 Working with files and directories

ls list storage

cp copy

mv move (or rename)

cd change directory

touch create file and update timestamp

stat show information on a file

file show type of file

```
ls
cd ~
pwd
cd /tmp/
touch TEST-$USER-$(date -I)
ls -l TEST-*
```

5.3 cat & tac & more & less & most

cat concatenate all provided files (sorted) to STDOUT, a single "-" in the parameter list represents STDIN.

tac similar to `cat`, but reverse line order

more & less & most similar to `cat`, but provide additional features like *pagination*, *text search* or *colour support*.

5.4 Tab-Completion

You can use the `Tab` key to complete partially entered commands - in a lot of cases also for parameters.

Life demonstration!

5.5 History!

The shell stores a history of recent commands - apart from using the `up` and `down` cursor keys to walk the command history there are some commands helping:

```
history
hgrep grep
```

5.6 Interesting keyboard shortcuts

`Ctrl-a` jump to beginnig of current line

`Ctrl-e` jump to end of current line

`Ctrl-t` "transopse" - change the positions of characters left of the cursor; usually these are the last characters typed.

`Ctrl-r` reverse history search

!<exp> repeat the last command matching <exp>*

`Ctrl-c` terminate current process

Life demonstration!

A remark: on keyboards with german layout the `Ctrl` key is labeled `Strg` (*Steuerung*).

5.7 Variables

```
TEST="Das ist ein Test!"
export TEST
echo $TEST
```

5.8 Pipes & IO redirection

```
man -t bash | grep ^%%Page: | wc -l
man -t man > man.ps
gv man.ps
```

6 More on "Using the shell"

6.1 What is a shell skript? How to write one?

Shell scripts are just collections of commands.

Open a file in your programming editor of choice and start with a line specifying the command interpreter (could also be "perl", "python", "awk", "m4", "make", ...).

```
#!/bin/bash
```

The remainder of this line will be supplied to the interpreter, `-x` is usually helpful to find errors. Write some commands as you would type them on your console. Save the file - a special extension is not necessary - and make the file executable:

```
chmod +x my_new_shell_script
```

In your skript you access command line parameters via variables `$1`, `$2` to `$n`; the scripts name is stored as `$0`.

6.2 I/O redirection

The system provides two output channels and one input channel to every process - as usual these are regarded files:

STDIN standard input, from the keyboard

STDOUT standard output, → screen

STDERR standard error output, → screen

Those channels can be redirected to other "files":

```
my_cool_software  <input  >output  2>>error.log
```

The programm `my_cool_software` will read anything it expects from the keyboard from file `input` instead, writing any output to `output` and append error messages to `error.log`.

A device (*special file*) often used for redirecting error messages is `/dev/null`, which discards any written data.

6.2.1 Here documents

Here documents are a special of *input redirection* with the operator `<<` allowing multiline input right inside a shell script without using an additional file.

```
software <<EOFlu  1st line of input  2nd line of inputEOF
```

The delimiters for text can bee choosen freely, EOF is just a common string reminding the ASCII character EOF – End Of File. Trailing blanks in *here documents* are ommited.

6.2.2 Here strings

A variant of *here documents*, allowing for shorter text snippets on a single line.

```
cat <<< "one line of input"
```

6.3 Pipelines

A programms STDOUT can be rewired to another programms STDIN via the pipe operator `|`:

```
ls | grep 2001
```

To connect STDOUT and STDERR combined to next STDIN, you can use `|&` or `2>&1 |`.

6.4 Return value

Every command in the shell returns a value. By convention, a value of "0" means succesfull operation, a value bigger than zero has to be interpreted as an error code.

You can have a peek at the last return value with `echo $?`.

```
/bin/true  ; echo $?
/bin/false ; echo $?
```

Unluckily not every executable honors this convention, but most standard utilities do.

6.5 Lists

Lists are sequences of one or more pipelines separated by one of the operators `;`, `&`, `&&`, `||` or `<newline>`.

`;` or `<newline>` commands delimited by `;` or `<newline>` are called in sequence

`&` start the previuos command in the background and immediately proceed with next command

`command1 && command2` control operator, standing for logical AND - `command2` is executed only if `command1` returns success - e.g. use for handling prerequisites

`command1 || command2` control operator, standing for logical OR - `command2` is executed only if `command1` returns an error code - e.g. use for error handling

6.6 Background jobs

& Commands terminated by `&` are executed in the background.

Ctrl-z Halt a job and bring the shell prompt to the foreground.

jobs Show background and halted jobs.

fg Bring background or halted jobs to the foreground again.

bg Run a halted job in background mode.

nohup Run a command immune to hangups - without a controlling terminal.

6.7 Expansion

6.7.1 Brace Expansion

```
mkdir -p projekt/{conf,doc,misc,src,test}
```

6.7.2 Pathname Expansion (aka. globbing)

If the shell finds any of the latter characters in the words of a command line, it tries to match those to files in the working directory and replaces them with an alphabetically sorted list of those files.

***** matches any sequence of characters, even the null string

? matches exactly one character

[...] matches any of the enclosed characters. It is possible to use range expressions like `[A-J]` meaning every capital character between "A" and "J" in the character set in use (see `locale(1)`).

6.7.3 Parameter Expansion

`${parameter}` evaluates to the value of the *paramter*

`${parameter:-text}` substitute *text* if *parameter* is not defined

`${parameter:offset:length}` extract a substring

`${parameter#text}` remove prefix matching *text*

`${parameter%text}` remove sufffix matching *text*

```
file="/tmp/tmp.TEST.jpeg"
touch $file
mv $file ${file%.jpeg}.jpg
ls -l /tmp/tmp.*.jpg
```

6.7.4 Command Substitution

Command substitution allows the output of a command to replace the command name.

`$(command)`
`` `command` ``

Both forms are roughly equivalent; the shell executes the command in a subshell and replaces *command substitution* with output from *command*.

6.7.5 Arithmetic Expansion

Arithmetic expansion evaluates an arithmetic expression and substites the result.

`$((expression))`

Evaluation is done in fixed-width integers with no check for overflow. Operators and their precedence, associativity, and values are the same as in the C language

6.8 Defining an *alias*

Aliases substitute a string for the *first word* of a command. This is usually used to define abreviations for short short sequences of commands or provide default arguments.

```
alias ..='cd ..'
alias ...='cd ../..'
alias kalender='pcal -F 1 -E -a de -m | lpr -Zsimplex'
```

The command `alias` by itself shows your currently defined alias definitiones. To define your own, write them in a file `~/.alias`, which is parsed by the shells init skripts on startup.

6.9 Conditional Expression

Conditional expressions are used by the `[[` compound command and the `test` and `[` builtin commands to test file attributes and perform string and arithmetic comparisons.

`-e file` file exists

`-d file` file exists and is a directory

`-r file` file exists and is readable

`-s file` file exists and has a size greater than zero

`-z string` length of string is zero

`-s string` length of string is not zero

`string1 = string2`=, `string1 ! string2`= strings1 and 2 are equal/not equal

6.10 Compound commands, control structures & loops

Usual forms of compound commands are

`( list )` `list` is executed in a subshell environment

`{ list; }` `list` is simply executed in the current shell environment and has to be terminated with a newline or semicolon. This is also known as a *group command*.

`((expression))` expression is evaluated as an integer (see Arithmetic Expansion).

`[[ expression ]]` expression returns 0 or 1 according to the rules of Conditional Expressions.

6.10.1 if ... then ... else

The `if` construct tests whether a evaluation of an expression returns zero (success!) and executes different compound commands:

```
if [ -d $HOME/Download/new ];
then
    DESTINATION=$HOME/Download/new/
else
    DESTINATION=/tmp/
fi
```

6.10.2 case

```
case $(hostname -s) in
    faepop?? )
        echo "Workstation for staff and diploma students."
        ;;
    faepcr?? )
        echo "Computer in students lab."
        ;;
    * )
        echo "I know nothing about $(hostname -f)"
        ;;
esac
```

- Each test line follows the globbing rules and ends with a right parenthesis `")"`.
- Each condition block ends with a double semicolon `;;`.

6.10.3 for loops

`for` loops come in iterative and arithmetic flavour:

```
for file in *.jpeg
do
    mv $file ${file%.jpeg}.jpg
done
```

```
LIMIT=10
for ((a=1; a <= LIMIT ; a++))  # Double parentheses, and naked "LIMIT"
do
        echo -n "$a "
done
```

6.10.4 while & until

`while` and `until` loops execute the compound statement `list-2` as long as the statemant `list-1` is evaluating to true or respectively false.

```
while list-1; do list-2; done
until list-1; do list-2; done
```

6.11 Defining a function

A shell function is called like a simple command and executes a *compound command* with a new set of parameters. This allows more flexibility with parameters than an *alias* and is faster and does not produce that much clutter in the file system than writing a *shell script*.

```
function YouTube_download() {
  youtube-dl "$1" \
             --download-archive ${CACHEFILE} \
             --prefer-free-formats \
             --playlist-end $2 \
             --restrict-filenames \
             --output
             ↪  "${MEDIADIR}/%(uploader)s/%(upload_date)s_-_%(title)s_-_%(id)s.%(ext)s"
}
export -f YouTube_download
```

6.12 some more useful commands

6.12.1 head & tail

Copy the leading or last n lines of input to STDOUT. An especially usefull paramter to `tail` is `-f` which allows monitoring for additional data.

6.12.2 cut & paste

`cut` allows the extraction of columns from file in tabular form.

```
head -n 1  /etc/passwd
echo
head -n 1  /ect/passwd | cut -d : -f 6,7
```

`paste` writes corresponding lines from several files side by side

```
paste /etc/passwd /etc/group | head -n 3
```

6.12.3 find (& fdfind)

The `find` command allows to search recursively for files meeting various constraints:

`-name` search for a file with a name pattern - globbing rules apply

`-iname` same as search, but *case insensitive*

`-type` search for directories, regular files, links, ...

`-ctime n` files last changed n days ago

`-empty` file or directory is empty

There is a simpler (and sometimes faster) alternative to find: `fdfind`.

6.12.4 xargs

`xargs` is an efficient way to collect output from find as parameters to another executable:

```
find /var/tmp/$USER -type d -empty -print0 \
    | xargs --null --no-run-if-empty \
            echo rmdir
```

6.12.5 grep

`grep` collects matching lines. There are several matching algorithms *regular expressions (re)* and *simple text match* to chose from - in our implementation there are three variants. Globbing rules do not apply :)

```
ls /afs/itp.tugraz.at/common/www/Lokales-TU/Arbeiten/Bakk/  \
    | grep "\-_200.-"
```

6.12.6 fzf & fzy

`fzf` & `fzy` allow for *interactive* and *fuzzy search* on the provided input. They are both new to me; waiting for your input!

Simply a better history?

```
alias H='history | fzf'
```

Or a helper to sort files?

```
function MV() {
    echo mv -v "$1" "$(find $HOME/Download/ -type d \
| fzf)"
}
# oder
MV() { echo mv -v "$1" "$(find $HOME/Download/ -type d | fzf)" ;}
```

6.12.7 ed & sed

`ed` and `sed` are line oriented editors, quite usefull in scripting. `ed` (*editor*) is specialised on inline editing files while `sed` (*stream editor*) is suited for stream editing. They share a similar, but not identical command set.

```
ed -s mein_datei <<< 'H\ng/muster/d\n,w'
sed 's/regexp/replacement/g' inputFileName > outputFileName
```

Others know a lot more on ed!

6.12.8 module

`module` provides for the dynamic modification of your environment:

```
module avail
module load gcc
module list
gcc -v
```

6.12.9 which

Find the real path to an programm you intend to run - in case of weird problems this lets you verify which programm is really run.

```
which gcc
gcc ~/TEST/Compiler/hello.c -o test
test
which test
./test
```

6.12.10 mktemp

Creates a temporary file or directory -

`mktemp /tmp/$USER_XXXXXXX`

6.12.11 install

`install` is usually called from within a shell script or a `Makefile` - more on `make` maybe in a later installment.

```
PREFIX=/temp/$USER
PROJECT=fritz

install --mode 755 --directory     $(PREFIX)/bin/
install --mode 755 --directory     $(PREFIX)/man/man1/
install --mode 755 $(PROJECT)      $(PREFIX)/bin/
install --mode 644 $(PROJECT).1.gz $(PREFIX)/man/man1/
```

Short digression to `mode` (see `chmod(1)` for more information):

755 means read and execute rights for everybody, write access for the owner of the file

644 means read rights for everybody, write access for the owner of the file, execute rights for nobody

6.13 further reading

- bash(1)
- Advanced Bash-Scripting Guide
- Bash Pitfalls
- The Bash Hackers Wiki
- Kenneth Ward Church: Unix for Poets

7 Build automation

7.1 Motivation

Automation of repeating processes is a central idea in computing. Build automation not only increases your productivity by eliminating redundant tasks but also increases confidence in results.

Early build automation used custom crafted shell scripts for compiling and installing software.

7.2 Make

Make is one of the oldest and still widely used programmes in build automation.

Besides building programs, make can be used to manage any project where files have to be updated automatically from others whenever they change. It can also work as replacement for (small) shell scripts.

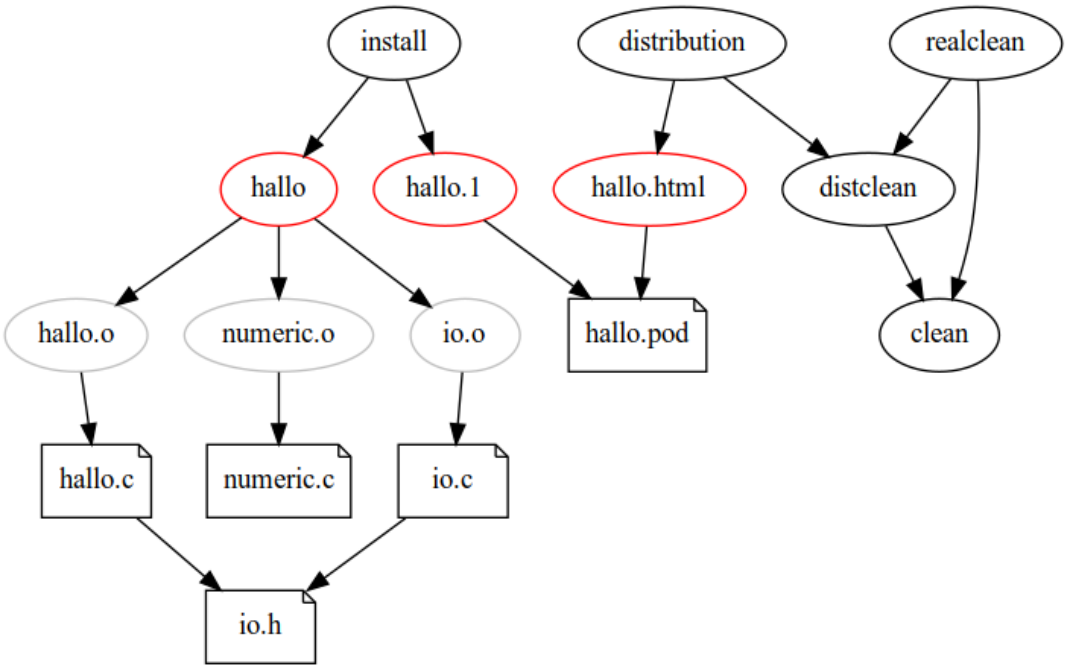
It is also the basis of most newer build systems, which create *Makefiles* as results.

Alternatives are usually confined to their specific ecosystem - eg. *ant* for Java programming, *SCons* is predominantly used by Python programmers, ...

7.2.1 What is make

Make is at its heart a solver of dependency graphs.

It reads a description of rules, targets and dependencies and tries to minimise the necessary steps to create requested targets.



7.2.2 Rules, Targets & Dependencies

In general, a *rule* looks like this:

```
targets : dependencies
        recipe
        ....
```

Targets are file names, separated by spaces - wildcards may be used. Usually there is only one target per rule.

A target is out of date if it does not exist or if it is older than any of the *dependencies* - also file names separated by spaces and optional wildcards.

The *recipe* describes the way to generate the *targets* from their *dependencies*. One major roadblock is indentation - a `Tab` character is necessary!

7.2.3 Phony Targets

If you write a rule whose recipe will not create the target file, the recipe will be executed every time the target comes up for remaking.

```
clean:
        rm *.o *~
```

If later on a file "clean" is created it would be considered up to date - no dependencies - therefore the cleaning action will never be performed. Make `clean` an dependency of the special target `.PHONY` to run this recipe unconditionally.

```
.PHONY: clean
clean:
        rm *.o *~
```

7.2.4 Variables

Variables can represent lists of file names, options to pass to compilers, programs to run, ...

To reference a variable use `$(variable)` or `${variable}`:

```
src = hallo.c io.c numeric.c
program : $(src)
       gcc program $(src)
```

7.2.5 Automatic variables

Some variables are set automatically inside rules:

`$@` The file name of the rules target of the rule.

`$<` The name of the first dependency.

`$^` The names of all the prerequisites.

`$?` The names of all the prerequisites newer than the target.

The last example could also be written as:

```
program : hallo.c io.c numeric.c
        gcc -o $@ $^
```

7.2.6 Implicit Rules

Some actions are performed quite regularly - like compiling C/C++ or FORTRAN code to object files and later linking them to executables.

For a lot of these recipes are already known to make; but you can create your own implicit rules:

```
%.pdf : %.tex
        rubber --pdf $<
```

7.2.7 Implicit Variables

The behaviour of builtin implicit rules is controlled by implicit variables; eg.

CXX compiler for C++

CXXFLAGS flags given to the C++ compiler

LDFLAGS flags given to the linker, such as -L.

LDLIBS libraries to link

7.2.8 Parameters & Options

`-n` Print the commands that would be executed, but do not execute them.

`-f <file>` Specify input should be read from `<file>`; if not present, the default of `./Makefile` is used.

`-C <directory>` Change to `<directory>` before doing anything else

7.3 Autoconf, Automake & Libtool

Autoconf, automake & libtool are a suite of GNU tools to help creating Makefiles working on a wide range of operating systems.

`autoconf` tests the computer system for selected features as installed system tools, library versions, widespread bugs ... A shell script `configure` with those tests is created.

`automake` converts a description of internal dependencies between components (`Makefile.am`) from a less verbose format into a stub `Makefile.in`.

`libtool` hides the complexity of generating special library types (such as shared libraries) behind a consistent interface.

The previous steps are usually executed by the provider of a software package. The customer later executes the `configure` script from autoconf with fitting parameters, a Makefile is created and after calling `make` and `make install` the installation should be ready to use.

```
tar zxvf hallo.tar.gz  &&  cd hallo
./configure --prefix=/tmp/test-hallo
make  &&  make install
```

Autoconf, automake & libtool used to be quite common; you might have seen `./configure` quite often. For newer projects it is not that common since alternatives are said to be easier to use.

7.4 CMake

7.4.1 Whats better with CMake

- Easier notation for dependencies with auto-detection
- makes out-of-tree builds easy
- easier cross compilation
- optional interactive User Interface `ccmake`
- very few external dependencies: C compiler and *native build tools*
  - supports **make**, nmake, ...
  - supports **Apple XCode**
  - supports **MS Visual Studio**
- module "CTest" provides a testing framework
- module "CPack" enables easy build of installation packages
  - for **Microsoft Windows**
  - for **Apple macOS**
  - and various UNIX/Linux package formats: Debian's **dpkg**, Redhat's **RPM**, ...

7.4.2 Example Configuration

For a minimal examples just create a file `CMakeLists.txt` next to your sources. Specify a project name, the executable and its sources:

```
project (Hallo)
add_executable (hallo hallo.c io.c numeric.c)
```

7.4.3 Find a library

```
project (Hallo)
add_executable (hallo hallo.c io.c numeric.c)

find_package (BLAS)
if (BLAS_FOUND)
  target_link_libraries (hallo ${BLAS_LIBRARIES})
endif (BLAS_FOUND)
```

See `/usr/share/cmake-X.XX/Modules/` for libraries known to CMake.

7.5 Further reading

- `https://en.wikipedia.org/wiki/Make_(software)`
- `man make` and more elaborated `info make`
- Gary V. Vaughan & Ben Elliston & Tom Tromey & Ian Lance Taylor . GNU Autoconf, Automake, and Libtool . New Riders publishing . 2006 – available online
- Ken Martin & Bill Hoffman . Mastering CMake - A Cross-Platform Build System . Kitware Inc. 2010 . ISBN 978-1-930934-22-1
- `https://gitlab.kitware.com/cmake/community/wikis/home`
- `https://gitlab.kitware.com/cmake/community/wikis/doc/tutorials/How-To-Find-Libra`