YOU, LOGIN, AND LINUX

Nikhef Computing Course, Tuesday 2022-11-22

Dennis van Dok

Nikhef

LEARNING GOALS

- Know how to login; ssh keys, protecting ssh keys, setup and safely use (proxy) tunnels
- Basic passwords security
- Manage your personal home page
- Understand and apply general unix principles
- Basics of scripting and programming
- Know the most common command line tools
- Understand file permissions and how to change them

UNIX FOR PHYSICISTS

The purpose of this talk is to give a few helpful pointers to aspects of the general Linux computing environment that may be beneficial to know for a student of physics, i.e. not someone who purposely explores the realm of computing but rather sees this as (at best) a useful tool or (at worst) a necessary hurdle to overcome.

THE PHILOSOPHY OF UNIX

What? Unix has a philosophy?

YES! AND CULTURE, TOO

The Unix environment has never been known to be particularly user friendly and it can be daunting to get started.

The general philosophy seems to be that the user is supposed to be able to figure out how to solve even the most complicated tasks using a combination of very basic tools.

DO ONE THING...

There is a minimalist approach to Unix that may take a little while to get used to.

- The basic tools are made to work together
- Everything deals with text streams
- So tools can be strung together and one tool's output is input to another
- The user interface is secondary

FINDING SUPPORT

Although the learning curve can be steep, there is actually plenty of help available.

- Nowadays the web has plenty of answers
- Your peers may be of tremendous support
- With enough basic skills, finding help on your own becomes very feasible

DEVELOPING SKILLS

Nikhef

Nikhef

SHOULD YOU LEARN A NEW SKILL?

T	time normally spent on related tasks
I	investment
R	rate of productivity increase

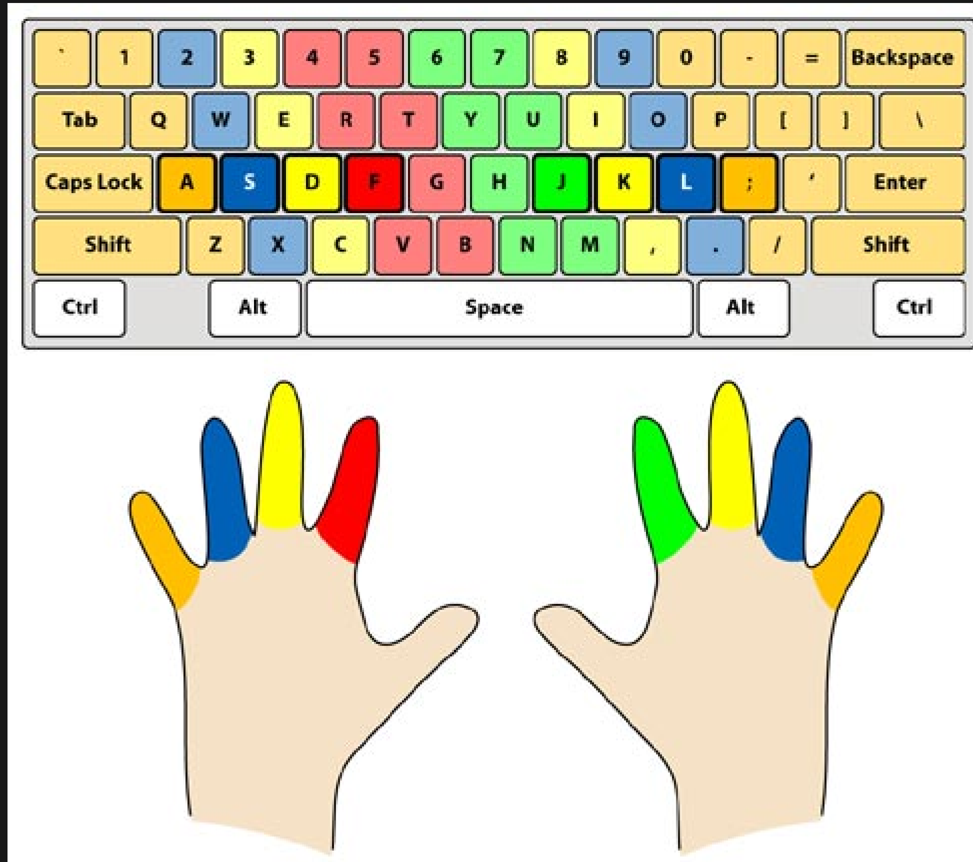
SHOULD YOU LEARN A NEW SKILL?

T	time normally spent on related tasks
I	investment
R	rate of productivity increase

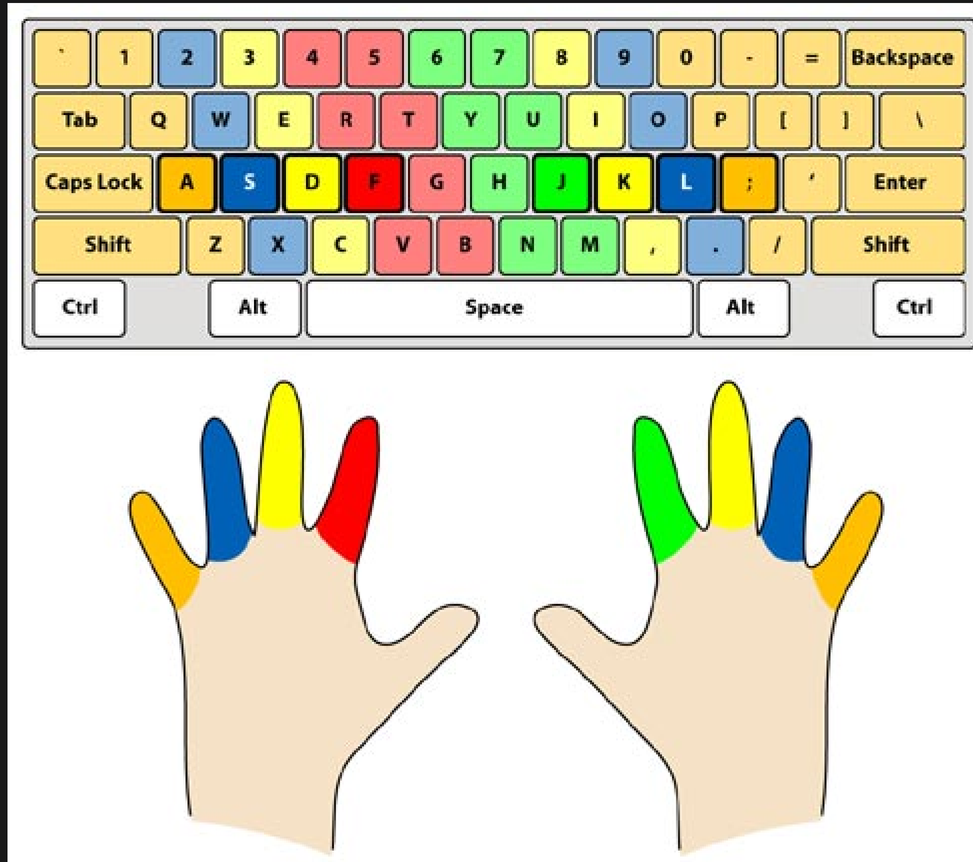
Learning a skill is worthwhile if

$$T \geq I + \frac{T}{R}$$

SHOULD YOU LEARN TOUCH TYPING?

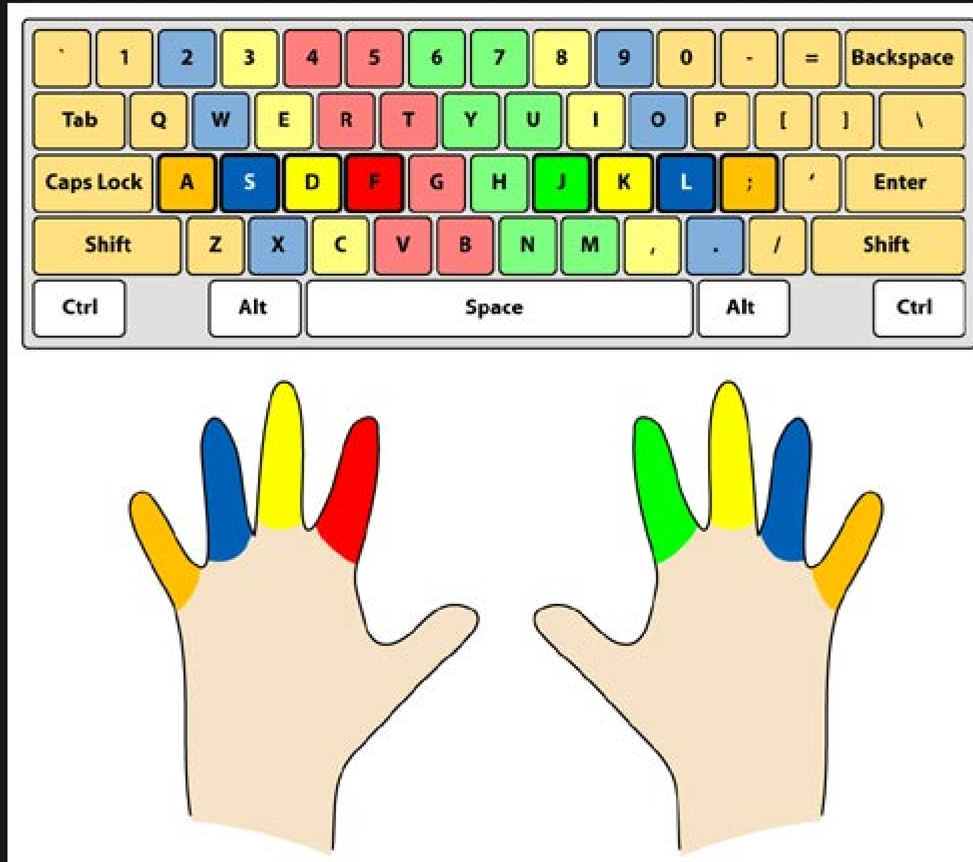


SHOULD YOU LEARN TOUCH TYPING?



- $T \approx 1000\text{h}$
- $I \approx 10\text{h}$
- $R \approx 2$

SHOULD YOU LEARN TOUCH TYPING?



- $T \approx 1000\mathrm{h}$
- $I \approx 10\mathrm{h}$
- $R \approx 2$

(Yes, absolutely)

SO HOW WILL I KNOW WHAT TO LEARN?

T, I, and R can only be learned from experience.

UNIX PRIMER

The operating system that you will encounter for most of your work is called Linux. it is an open source implementation of the UNIX operating system (originally) for the x86 architecture.

Starting in 1991, it rapidly gained traction due to the ubiquity of cheap PC hardware. It outpaced commercial UNIXen in many fields.

It may by now be considered the de facto standard for web servers, cloud computing and server and batch computing.

LINUX AT NIKHEF

The Linux kernel is core of the computing environment, but that made complete by a plethora of tools and services to integrate with it. This integration is done by companies and organisations that bundle everything up and make it available for installation and use. These are called distributions.

RED HAT ENTERPRISE LINUX

The distribution of choice in our field has been based on the Red Hat Enterprise Linux suite. Because Linux and all of the software that goes with it is open source, anybody can take the sources and rebuild what Red Hat has done.

This has happened several times. The CentOS distribution used to be a faithful rebuild of RHEL, but it has been bought by Red Hat and effectively dismantled; future rebuilds will be sourced from teams like Alma Linux or Rocky Linux.

THE BASICS

Linux is a multi-user, multi-tasking operating system. The kernel will schedule processes to use the available cores in a time-sharing fashion. The processes may interact with the system through system calls. File system interactions, network call-outs and virtual memory requests all go through the kernel.

Linux complies with the POSIX specification.

INPUT AND OUTPUT

In the most basic form, each process has three input and output streams.

`stdin`	standard input
`stdout`	standard output
`stderr`	standard error

These may be connected to a (pseudo) terminal for interactive user shells.

PROCESSES

Every process in the system is listed in the process tree and has a unique ID. The process with PID 1 is the startup process that is responsible for getting the entire system started up. Every other process is a child or descendent of this process. New processes are created by forking the parent process; the child inherits all of its parent's properties, like memory and open files.

MEMORY MANAGEMENT

Linux employs a virtual memory mapping system that gives each process it's own private space. The program code is mapped into the code segment; the data segment is allowed to grow as more memory is demanded. There is no guarantee that requested virtual memory is actually available as physical RAM. Linux employs an oversubscribing model that allows processes much more virtual memory than is technically available, because in practice many programs do not actually use all of the requested memory.

SWAP

If the system runs out of RAM it will start to map pages out, preferring pages that have been least recently used. If that is not enough, some of the memory may be committed to swap space. This usually slows down the system considerably.

THE FILE SYSTEM

All files in the system are organised in a single file tree. The root of the tree is usually on the local hard drive, but various parts of the file system may be mounted from the network. There are also some pseudo file system entries that give a file-like view of parts of the system, such as `/proc` for the process table, and `/dev` for connected devices such as peripherals.

PERMISSIONS

Each entry in the file tree has a set of permission bits:

```
  ,- user (read, write, execute)
  |  ,- group (read, write, execute)
  |  |  ,- other (read, write, execute)
  |  |  |
-rwxrwxrwx
```

This can be symbolically represented as above or as an octal number:

```
-rwxr-xr-x  755
-rw-r-----  640
```

The execute permission is used for both traversing down a directory and for actual execution of programs.

Nikhef

LINKS AND INODES

Each entry in the file system has an inode that is *linked* to a directory. Directories link to themselves (seen as the '.' entry) and to their parent directory ('..'). Opening a file in a process increases the link count, removing it from a directory decreases the link count. When the link count hits 0, the file is considered deleted and its storage space may be recycled.

Nikhef

HARD AND SYMBOLIC LINKS

A file may be linked multiple times, in different directories and under different names. Directories may not be linked in this way, as that would lead to chaos.

This type of link is called a *hard link*. It's counterpart the *symbolic link* is simply a string of text that points to another location.

```
lrwxrwxrwx.   1 root root     7 26 mrt  2020 bin -> usr/bin/
```

File permissions don't apply to symbolic links. The system will treat the use of symbolic links as if the target file was meant.

USERS, GROUPS AND PRIVILEGE

The process with PID 1 is run under user ID 0, or `root`. This is the system account that has full privileges over the system. Only system administrators can use this account.

There are many system user accounts for running system services that don't require full access.

Each process runs under a certain user and group ID. This ID determines what parts of the file system the process has access to. The Nikhef systems share the user and group identity through the LDAP directory.

NETWORK SOCKETS

Processes may open sockets on the network. An internet socket has an address and a port number (0–65535). Outbound connections can be made to send and receive data.

Listening on local ports allow other processes to connect inbound.

A connection has two ports: a source and destination. Port numbers below 1024 are considered privileged.

Firewall rules restrict what kind of traffic is allowed, either inbound or outbound.

GETTING UNIX

Getting Linux on your laptop:

- http://get.debian.net/
- https://www.ubuntu.com/download
- https://getfedora.org/
- https://software.opensuse.org/
- https://rockylinux.org/download/
- https://mirrors.almalinux.org/isos.html

APPLE HARDWARE

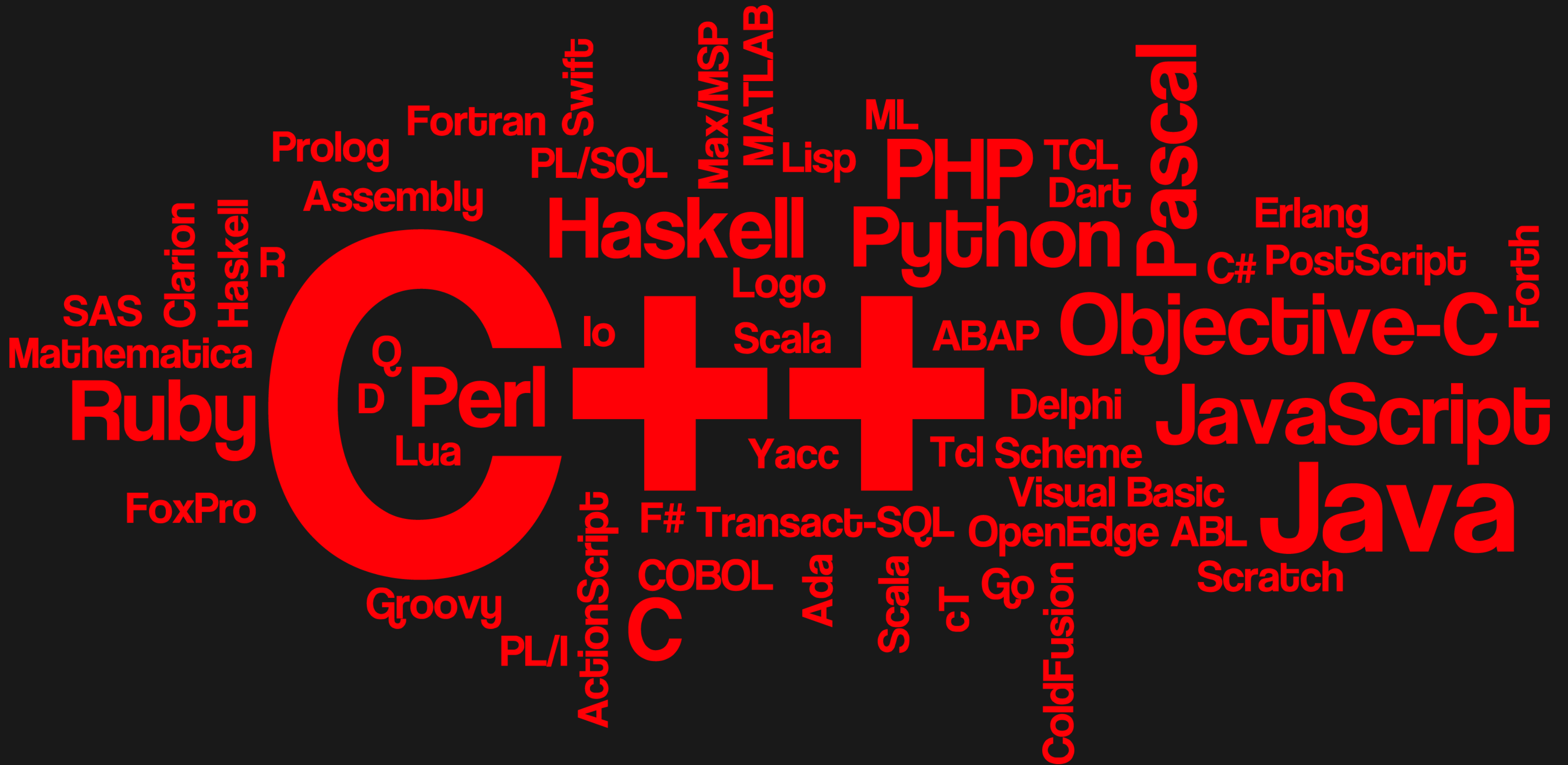
- OS X = Unix
- VirtualBox/VMWare
- hard-core install Linux anyway

MICROSOFT WINDOWS

Often the best choice when there is but one choice. In that case:

- Dual Boot
- VMWare/VirtualBox
- CygWin
- WSL 2

PROGRAMMING LANGUAGES



SCRIPTING LANGUAGES

- No compilation required
- Easy prototyping
- Can be used interactively
- Ideal to build workflows

Examples:

- Bash (both interactive and for programming)
- Python
- Perl

COMPILED LANGUAGES

- Translate down to the CPU instruction level
- High performance
- Various degrees of abstraction away from the underlying architecture

Examples:

- C/C++
- Fortran
- Go
- Rust

MOST LIKELY COMBO

Python/C++

(Special recommendation: Jupyterlab)

A NOTE ON C++

There is a decade of architectural development between current CPUs (AMD EPYC 7H12) and what we still had a few year ago (Intel Xeon E5-2650). The clock speed, however, is still in the same ballpark.

Principally, your C++ program will compile to both. Technically, to make use of all the advancements in processor design it takes a lot of insider knowledge of both the CPU and compiler optimisation.

LOGIN AND STOOMBOOT

You will encounter Linux at Nikhef on the login server and on the stoomboot batch system. Both run a Red Hat Enterprise Linux compatible OS.

These systems are accessible via **ssh**.

THE LOGIN SERVER

The login server is the only system available to you if you want to connect to the Nikhef infrastructure if you are not on the Nikhef network.

(Using a VPN like eduVPN counts as being 'on' the Nikhef network.)

This system should not be used for anything other than the most basic tasks. It has **no** computational power, so don't even think about compiling your software here.

YOUR PERSONAL HOMEPAGE

The `public_html` in your home directory is automatically turned into a web page. Write a basic HTML file called `index.html` there and presto! You now have web home.

https://www.nikhef.nl/~dennisvd/

A FEW NOTES ON YOUR PUBLIC PRESENCE

You get to decide how much you want to share about yourself. But be careful with publishing anything with personal data of others!

- Automation is great, but even a listing of a project directory or the stoomboot queues may expose user identities.
- PHP scripts can be very powerful, but they may contain security holes that invite abuse.

SSH

- secure remote shell
- Passwordless
- versatile

HOW IT WORKS

A securely encrypted connection is made between the client and the server.

The client authenticates itself by some means (e.g. password, public key). If the user is authorised, the server process forks a new shell on behalf of the user and attaches is to a pseudo terminal device.

Nikhef

SETTINGS

.ssh/config

```
Host *.nikhef.nl
    ControlMaster auto
    ControlPath /tmp/%h-%p-%r.shared
Host *
    ForwardAgent yes
    User yournamehere
    HashKnownHosts yes
```

SSH PUBLIC/PRIVATE KEY

```
ssh-keygen
cat ${HOME}/.ssh/id_rsa.pub > authorized_keys
scp authorized_keys login:.ssh/authorized_keys
```

Permissions:

```
drwxr-xr-x .ssh/
-rw-r--r-- .ssh/authorized_keys
-r--r--r-- .ssh/id_rsa.pub
-r-------- .ssh/id_rsa
```

PUBLIC KEY AUTHENTICATION

The client authenticates by presenting a public key that is authorised by the target user. A cryptographic challenges is presented that the client can answer only because it holds the private key.

The private key is *really* private. Don't share it or copy it to other systems!

The public part goes to all machines that you want to log on to.

MULTIPLE PRIVATE KEYS

If you have more than one device as your starting point (e.g. a laptop but also a desktop computer at work) both systems get their own private key. Just add both public key to the `authorized_keys` file.

AGENT FORWARDING

```
ssh-add -l        # list keys in the agent
ssh -A login      # login with agent forwarding
```

SSH AGENT

Logging in through a chain of servers is easier with an ssh agent. Normally an agent is already started for you.

The forwarding means that the agent can be reached through a backchannel.

This saves so much typing of passwords that this should almost be considered mandatory.

Nikhef

PROXY FROM OUTSIDE NIKHEF

```
Host stbci2.proxy
    Hostname stbc-i2.nikhef.nl
    user yournamehere
    CheckHostIP no
    ProxyCommand ssh -q -A login.nikhef.nl /usr/bin/nc %h %p 2>/dev/null
```

PROXY JUMPING

This little trick so useful that recent implementations of ssh have now incorporated this functionality so you could try the `ProxyJump` option instead. See the man page for `ssh_config`.

In combination with Agent forwarding this means you get to log on to Stoomboot from anywhere in the world without typing your password once.

SSHFS

Fuse mount your remote home directory locally:

```
sshfs login.nikhef.nl: /tmp/login
ll /tmp/login/
fusermount -u /tmp/login
```

COMMAND LINE SHELL

- tell the computer what to do, one line at a time
- most powerful way of direct interaction
- also used for scripting and fast prototyping
- ideal for taking notes as you go

WHICH SHELL DO I NEED?

select your default shell at https://sso.nikhef.nl/chsh.

`/bin/bash`	**YES**
`/bin/zsh`	**YES**
`/bin/csh`	**NO!**

Nikhef

TUNING

- everything can be tuned
- but you must resist
- use only the common enhancement

STARTUP FILES

login shell	`.bash_profile`
non-login shell	`.bashrc`

This distinction is outmoded.

.bash_profile

```
if [ -f "$HOME/.bashrc" ]; then
    . "$HOME/.bashrc"
fi
```

PATH

`.bashrc`

```
if [ -d "$HOME/bin" ] ; then
   PATH="$HOME/bin:$PATH"
fi
```

Do not put `.` in your `PATH` and certainly not at the beginning! This poses a security risk because you will not be sure that you are not running a program from a local directory that you did not intend to run. It is better to adopt the notation of `./program` for local programs.

COMPLETIONS

- pressing `TAB` will auto-complete your command line
- works better with the package `bash-completions` installed

Nikhef

HISTORY

.bashrc

```
# don't keep more than one copy of a repeated command
HISTCONTROL=ignoredups
# append to the history file, don't overwrite it
shopt -s histappend
# keep plenty of history
HISTSIZE=65000
# useful on systems with shared home directories
HISTFILE=${HOME}/.bash_history-$(hostname)
# keep track of time
HISTTIMEFORMAT='%F %T %Z # '
```

HISTORY RECALL

- Arrow up/down cycles through previous commands.
- `Ctrl-R` reverse search in history
- type `Ctrl-R` again to cycle back through matches
- or type more characters to refine the search term
- press enter to rerun the found command
- or press arrow keys to edit the command line

RECALL THE LAST ARGUMENT

Seeing is believing.

```
stat /some/path/to/file
# now I want to run cat on the same file
cat <ESC><.>
cat /some/path/to/file
```

PROMPT

.bashrc

```
PS1='\u@\h:\w \A $(__git_ps1 " (%s)")\$ '
```

This shows:

```
a07@lena:/project/newton 11:24 (master)$
```

Shows host name, working directory and current git branch.

Nikhef

ALIASES

```
alias ls='ls --color=tty'
alias ll='ls -lhF'
alias rm='rm -i'
alias mv='mv -i'
```

The interactive flag on dangerous commands are your training wheels.

KEEPING NOTES

- use `script` to capture an entire session
- run a jupyter notebook with a bash kernel
- emacs org-mode babel extension

Nikhef

SCRIPTING

Write `myscript.sh`:

```
# my first script
echo "This is my first shellscript"
```

And then run it like

```
bash ./myscript.sh
```

Turn it into an executable like so:

```
#!/bin/bash
# my first script
echo "This is my first shellscript"
```

followed by

```
chmod +x myscript.sh
./myscript.sh
```

ESCAPING

Make a habit out of always quoting variables like so:

```
"${var}"
```

and you will never go wrong.

EVAL IS EVIL

Do **not** use `eval` ever.

By the time you think you need eval, you need to switch to a real programming language.

Nikhef

PARSING COMMAND-LINE OPTIONS

```
#!/bin/sh
proxyhost=login.nikhef.nl
proxyport=8888
while getopts :h:p: OPT; do
    case $OPT in
    h|+h) proxyhost="$OPTARG" ;;
    p|+p) proxyport="$OPTARG" ;;
    *) echo "usage: `basename $0`"\
           "[+-h proxyhost] [+-p proxyport} [--] ARGS..."
       exit 2 ;;
    esac
done
shift `expr $OPTIND - 1`
OPTIND=1
ssh -n -N -f -D "$proxyport" "$proxyhost" "$@"
```

DANGERS OF QUOTES

Jeff thoroughly tested the following code. Then he changed one line. What went wrong?

```
#!/bin/bash
# clean up leftover files
# echo 'running in test mode'
echo 'now it's running in production'
path=var/batch/jobs
# it's ok to drop old file
retention="30"
find /$path -type f -mtime +$retention -exec rm {} +
```

```
#!/bin/bash
# clean up leftover files
# echo 'running in test mode'
echo 'now it's running in production'
path=var/batch/jobs
# it's ok to drop old file
retention="30"
find /$path -type f -mtime +$retention -exec rm {} +
```


```
#!/bin/bash
# clean up leftover files
# echo 'running in test mode'
echo 'now it's running in production'
path=var/batch/jobs
# it's ok to drop old file
retention="30"
find /$path -type f -mtime +$retention -exec rm {} +
```

DEBUGGING SHELL SCRIPTS

You will find yourself at times pondering why your shell script went south. Here is what you do next.

DON'T IGNORE ERRORS

```
echo $?
```

FAIL EARLY AND GRACEFULLY

```
set -e
trap 'fail $LINENO' ERR
fail() {
    echo "error on line $1" >&2
}
```

Nikhef

INPUT, OUTPUT, ERRORS?

input	stdin	0
output	stdout	1
output	stderr	2

REDIRECTIONS

Redirect both output streams to separate files.

```
run=`date -u +%FT%T`
./analysis.sh > "output.$run" 2> "err.$run"
```

DEBUGGING STATEMENTS

```
echo "now starting the frobnicator" >&2
```

Nikhef

TRACES

```
set -x
foo=somevalue
echo $foo
set +x
echo done
```

Renders:

```
+ foo=somevalue
+ echo somevalue
somevalue
+ set +x
done
```

DEBUGGING—CHECK THE ENVIRONMENT

Dump the environment and check carefully:

- `PATH`
- `LD_LIBRARY_PATH`
- `LD_RUNPATH`
- `PYTHONPATH`
- `LANG`, `LC_*`

KEEPING IT IN ONE FILE

For completeness sake, here we compound stdout and stderr onto a single file.

```
./whatever.sh > all_the_output 2>&1
```

Mind the ordering. First you need to send stdout to a file, then you want to send stderr to the same stream.

COMMON UNIX TOOLS

"do one thing and do it well."

- --help
- man/info/tldr
- Google

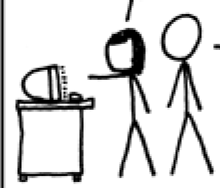
REGULAR EXPRESSIONS

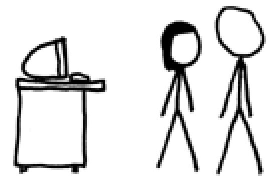
Find e-mail addresses:

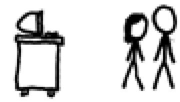
```
grep -E -o "\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,6}\b"
```

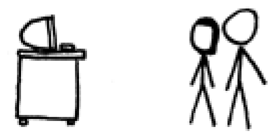
WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!

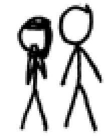

BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!


EVERYBODY STAND BACK.


I KNOW REGULAR EXPRESSIONS.



PERL!
TAP TAP



SOME OF THE MORE COMMON TOOLS

Some classic tools have been upstaged by modern alternatives, which bring more colour, git awareness and general usefulness at the cost of a basic pureness. They may not be generally available on all systems, and when writing scripts it's perhaps best to avoid them.

Where such a fancier alternative exists, it's listed in column 2.

Nikhef

TEXT MANIPULATION

`cat`	`bat`	just listed here for the most useless use of cat award
`sed`		streamline editor with regular expression powers
`awk`		the duct tape of Unix tools
`grep`	`ack`	find strings in files
`sort`		order lines
`jq`		`sed` and `grep` for JSON files

Nikhef

cut		select fields from each line
diff	meld	show differences between files
head / tail		tail -f is actually useful
tar		roll directories into tarballs
gzip		compress files or data streams

Nikhef

FILE SYSTEM

`ls`	`exa`	swiss army knife of file listings
`find`	`fd`	most of the time you want to use locate instead
`touch`		create files out of nowhere, update timestamps
`cp`		copy
`mv`		move or rename
`ln`		link
`tee`		copy stdin to stdout **and** a file

Nikhef

`rm`		really remove
`rsync`		copy on steroids
`which`		where is my executable?
`stat`		what can we tell about a file
`du`	`ncdu`	disk usage
`df`	`duf`	file system free space

SYSTEM PROCESSES

`ps`	`procs`	list processes, like `ps aux` or `ps -ef`
`top`	`htop`	who is eating my cpu and memory?
`kill`		sending signals
`bg/fg`		background/foreground programs
`lsof`		find open files
`vmstat`		memory, buffers and io
`free`		overview of memory

NETWORK

`ip`	swiss army knife of network tools
`ip addr`	show network addresses on this system
`ip route`	show the routing table
`ping`	see if we can reach a machine
`dig`	query DNS
`traceroute`	see which path takes us to a machine
`ssh`	secure shell
`nc`	netcat, less useless than cat
`curl`	swiss army knife of the web

PACKAGE MANAGEMENT

`apt/dpkg`	Debian's package manager
`yum/rpm`	Red Hat's package manager
`zypper`	OpenSUSE
`pip`	Python package tool
`conda`	More general packaging
`dnf`	Fedora packaging

PIPELINES

Traditional Unix tools are designed to work with stream processing in mind. With ‘pipes’, the tools can be linked together like perls on a string.

Below are a few examples.

JOB MANIPULATION ON STOOMBOOT BATCH SYSTEM

Find running jobs owned by user id and delete them (you can only delete your own jobs, of course).

```
qdel `qselect -u dennisvd -s "R" `
```

Nikhef

FIND AND GREP

This traverses a directory and finds all files of a certain name and then tries to grep for a certain pattern in these files.

```
find . -type d \( -path \*/.svn \
    -o -path \*/.git \) -prune -o \
    -type f \( -name \*.txt \) \
    -exec grep --color -i -nH -e searchterm {} +
```

MANIPULATE A SET OF PREDICTABLY NUMBERED FILES

```
for i in `seq -f file-%03g.txt 1 100` ; do
    sort -t, -n -k2 $i | cut -d, -f2,4-8 | \
        tail -n 1 > ${i%.*}.ord
done
```

A set of 100 comma-separated data files is numerically sorted on the second field, cut to only output fields 2, 4, 5, 6, 7, and 8, and then the last lines are saved to an output file.

DISK USAGE REPORT

```
du -s * | sort -n
```

Show which file/directory uses the most disk space.

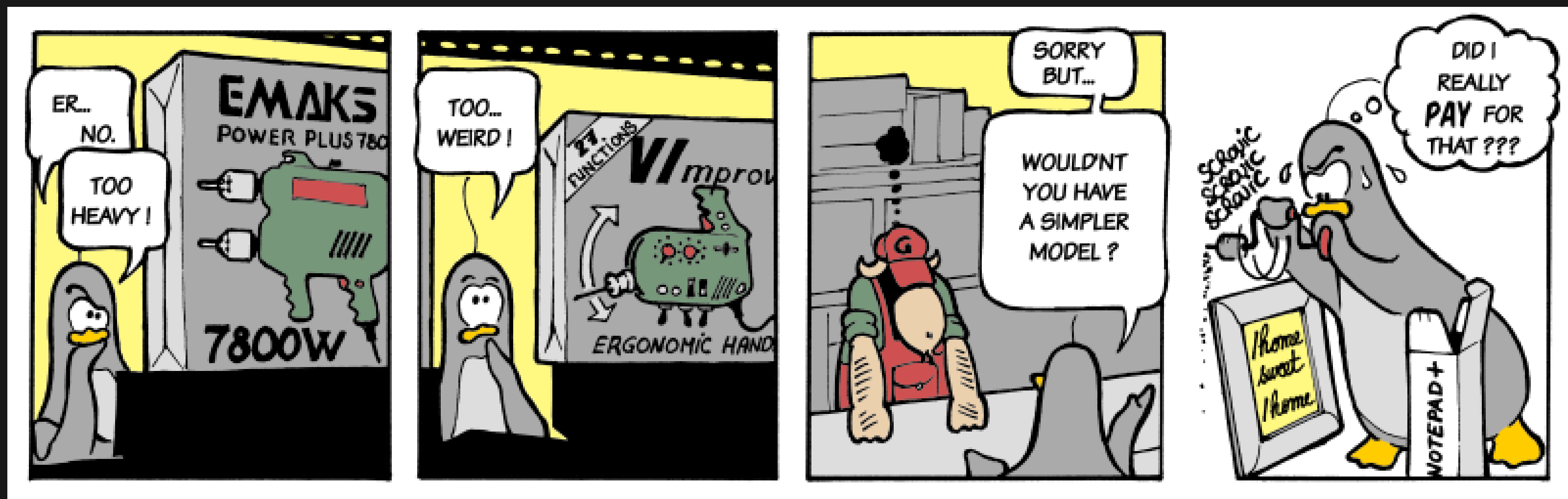
MOST RECENTLY CHANGED FILES

```
ls -lrt    # sort by timestamp
find . -mmin -10 -ls    # find files changed in the last 10 minutes
```

EDITING FILES

At some point you will need to edit files: source code, LaTeX files, shell scripts, configuration files…

Modern Linux systems have plenty of editors to choose from.



EMACS

- The thermonuclear word processor
- Everything and the kitchen sink
- Now with org-mode

EMACS

- The thermonuclear word processor
- Everything and the kitchen sink
- Now with org-mode

- $T \approx 1000$
- $I \approx \infty$
- $R \approx 100$

Emacs has a reputation for being slow and bloated, as well as overly complex. In truth, this editor has stood the test of time. There is active development and a ton of packages for every type of file and every type of workflow.

cons	pros
not generally installed everywhere	can edit files remotely
steep learning curve	built-in documentation
encourages heavy customisation	superbly extensible

VIM

Originally `vi`, its pedigree going back to the original editor called `ed`.

VIM

Originally `vi`, its pedigree going back to the original editor called `ed`.

- $T \approx 1000$
- $I \approx 10$
- $R \approx 3$

The original text editor of Unix. Nowadays it is actually "VI Improved" or VIM, which is much more powerful. The graphical version is called gvim. It can be personalised and extended.

cons	pros
editing modes require practice	powerful editing with very few keystrokes
limited extensibility	installed on nearly every system
strictly just an editor	Remote editing at lightning speed

SCREEN/TMUX

Sometimes you remote session should last longer than your workday. Or your laptop's battery.

The `screen` utility allocates a pseudo terminal attached to a background process independent of your session. You can run multiple shells in a screen and manoeuvre around with the Ctrl-A prefix. Type `Ctrl-A ?` for a help screen.

Nikhef

The tmux utility is a remake of screen, with modernised session handling, scripting, split screen, and ease of use. It is still less ubiquitous than screen so you may not have the option to run it unless you bring your own.

GIT

Version control of all your work, notes, programming, etc.

Nikhef has a public gitlab.

- $T \approx 100$
- $I \approx 10$
- $R \approx 2$

WORKFLOWS

- gitflow
- OneFlow

(This may not be your choice to make.)

SECURITY

Security considerations are usually not at the top of everyone's priority list. The adage: "Convenience, Speed, Security: pick two" might as well be

> *Convenience, speed, security: we know you will pick convenience and speed.*

RULE 1

Talk to the experts. At least once.

WHAT DO THE EXPERTS SAY?

As an aside, the experts are extremely pessimistic about our ability to keep the bad guys at bay forever. We read about data breaches at large companies, hospitals, and government organisations on a daily basis.

The best we can hope to do is be prepared and have adequate damage control in place.

RULE 2—PASSWORDS

Treat passwords with extreme care.

Passwords are considered ‘something only you know’, but as soon as you write them down somewhere, on a piece of paper or in a file, you could inadvertently share this with others.

Never put passwords in a script. There is always a better way. Be aware that passwords typed on the command line will appear in your history file.

RULE 3—DATA

Where does this data go? Who has access to it? The GDPR is very strict on how to handle personal information.

For Nikhef, personal data includes **user identities**.

This means that publishing the output of `qstat` on a personal web page is already a violation!

Nikhef

RULES 4 THROUGH ∞

- protect your security tokens (ssh private key)
- strong passwords
- different passwords everywhere
- do not log in from a public computer
- encrypt your phone
- encrypt your laptop
- encrypt your grandmother
- program with a deep mistrust of human beings

TEMPORARY FILES AND DIRECTORIES

Established practice for safely creating temporary files is by using `mktemp`.

```
tmpfile=`mktemp`
tmpdir=`mktemp -d`
```

This takes care of creating a new file with a randomised name that is guaranteed to be owned by the user.

USING PASSWORDS IN SCRIPTS

Sometimes scripts need to use a password to authenticate or unlock. The script can read the password from `stdin` and keep it in a local variable for the time that it is needed.

```
stty -echo
echo "enter password:"
read passwd
stty echo
mkproxy --passin - <<<$passwd
unset passwd
```

Be aware that putting passwords on the command-line means that it will show up in the process list.

FINALLY

Learn just enough Linux to get things done

http://alexpetralia.com/posts/2017/6/26/learning-linux-bash-to-get-things-done

Learning git branching

https://learngitbranching.js.org/

Advanced Bash-Scripting Guide

http://tldp.org/LDP/abs/html/

5 modern alternatives to essential Linux command-line tools

https://opensource.com/article/20/6/modern-linux-command-line-tools

Focus Hard. In Reasonable Bursts. One Day at a Time.

- https://www.calnewport.com/blog/2009/08/20/focus-hard-in-reasonable-bursts-one-day-at-a-time/

#Linux on Freenode.net IRC

- https://freenode.linux.community/how-to-connect/

Gitlab server at Nikhef

- https://gitlab.nikhef.nl/

- Let me Google that for you
  - http://bfy.tw/FDe5
- Emacs Org mode (it made this!)
  - http://orgmode.org/
- Reveal.js (it also made this!)
  - https://revealjs.com/