

R5CYBgos - Introduction to computer systems - 3

Samuel Péliissier (samuel.pelissier@centralesupelec.fr)

2025



Interacting with the system

Historically, terminals \simeq physical keyboard / monitor,

- Connected to the rest of the computing unit
- Write instructions, display results



The VT100 video terminal (contrary to terminals printing output on paper)

Virtual terminals / terminal emulators

Nowadays, terminals:

- Are software, with the same role as before: accept input, display output
- Can be provided by the kernel (CTRL+ALT+F1, F2...)
- Or higher level, with *graphical* terminal emulators
 - 🐧 Gnome Terminal, KDE Konsole, Alacritty...
 - 🪟 Windows Console
- Now include additional features: tabs, multiplexing (tmux), GPU acceleration...
- "tty" \simeq terminal

Why can't we just click on things?

- Most modern distributions distribute GUIs (Graphical User Interfaces) to interact with the system
- Some OSes and most users only ever interact using GUIs, e.g., Windows


Why should we use a terminal?

- Typing commands is (once mastered) faster and more reproducible
- Commands can be chained and scripted to automate complex tasks using multiple programs
- Not *everything* is available through GUIs
- Some setups do not have a screen (e.g., servers)
- Optional: you're not *most users*, and commands allow a better understanding of the system

Wait, what about the shell?

Shell:

- Something to start other programs
- Generally speaking, used as a command line interface (CLI) / interactive prompt
- Interpret commands, pass them to the kernel
- Can be used to write scripts
- Command history and command line editing
- Manage foreground and background processes (e.g., &, fg, bg)
- Controls where the output goes, does not display it

 bash, zsh, fish

 cmd.exe, PowerShell

Wait, what about the shell?

Each shell has its specificities and built-in commands

```
~ > time echo "a"  
a
```

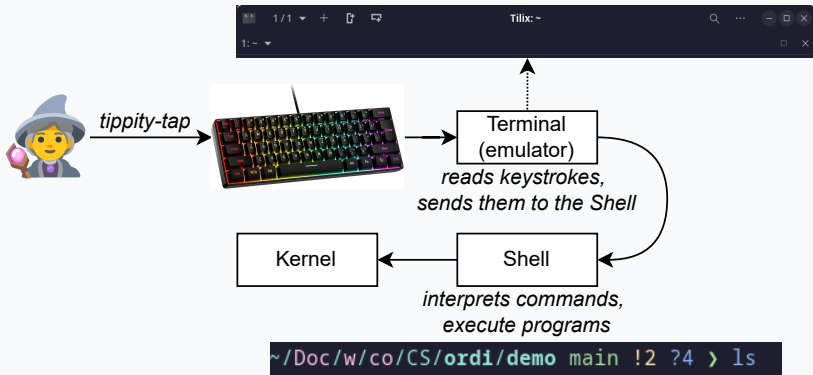
Executed in	6.00 micros	fish	external
usr time	6.00 micros	6.00 micros	0.00 micros
sys time	1.00 micros	1.00 micros	0.00 micros

```
~ > bash  
[s@computer ~]$ time echo "a"  
a
```

```
real    0m0,000s  
user    0m0,000s  
sys     0m0,000s
```

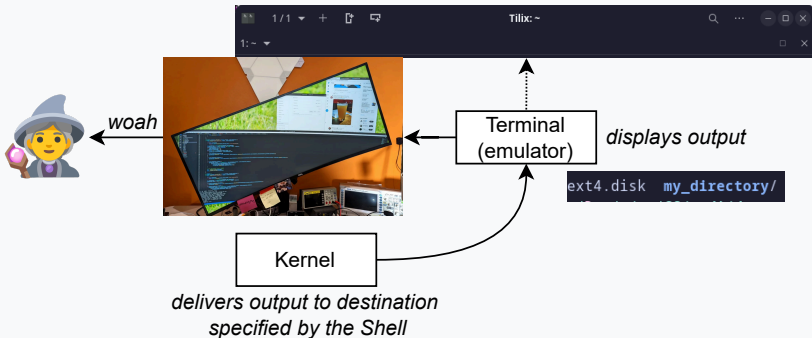
A more visual representation

A **terminal** runs a **shell**,
which communicates with the rest of the system



A more visual representation

A **terminal** runs a **shell**,
which communicates with the rest of the system



Input and output

As said in slide 6, the input/output are *standardized*:

- `stdin`: input
- `stdout`: output
- `stderr`: error

(Bonus: every shell command returns a code, e.g., error (-1), success (0))

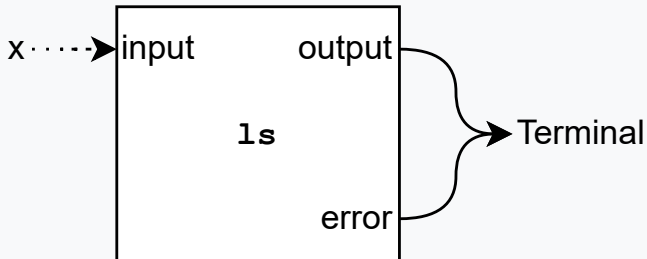
Bash allows redirecting I/O *somewhere*:

- > redirects `stdout` to a file (overwrites!)
- >> appends `stdout` to a file
- 2> redirects `stderr`
- < redirects `stdin` from a file

Input and output: redirections

Input can correspond to the keyboard or a file

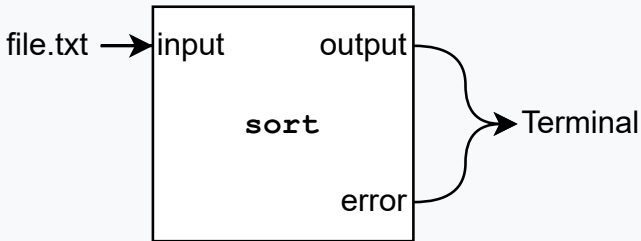
`ls`



Input and output: redirections

Input can correspond to the keyboard or a file

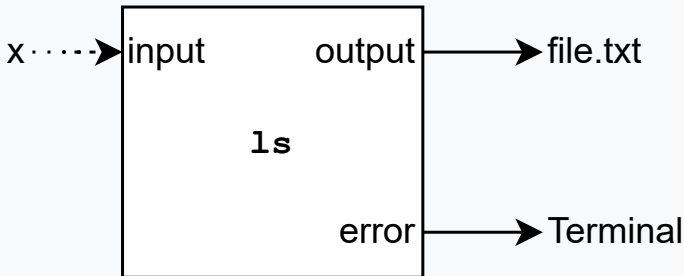
```
sort < file.txt
```



Input and output: redirections

Outputs can be the screen (i.e., the terminal) or a file

```
# Redirect standard output  
ls > file.txt
```

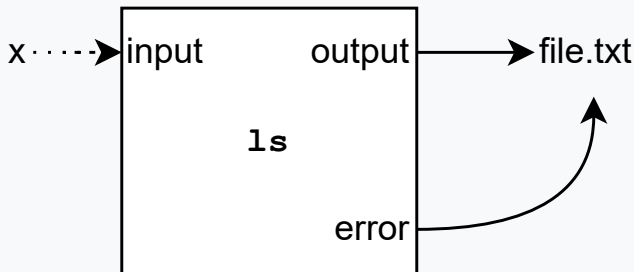


Input and output: redirections

Ouputs can be the screen (i.e., the terminal) or a file

```
# Redirect standard error  
ls /nonexistent 2> errors.txt
```

```
# Redirect both stdout and stderr  
ls /etc /nonexistent > file.txt 2>&1
```

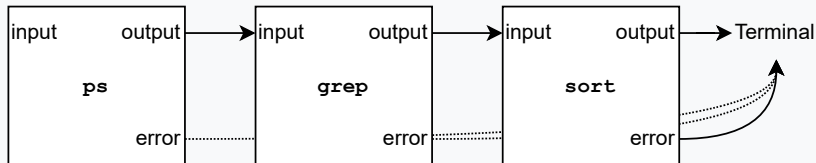


Input and output: pipes

Redirections send i/o to or from files, but sometimes we want to *connect commands directly*.

A **pipe** (|) connects the stdout of one command to the stdin of the next:

```
ps aux | grep python | sort -r
```



Input and output 🎁 final boss

We can not create I/O loops but... Everything is a file

- We can create *named pipes* (FIFO files)
- Other processes can read these streams as input
- (No data is written on disk; kernel buffers only)

```
mkfifo /tmp/fifo
```

```
nc 127.0.0.1 80 < /tmp/fifo | script.sh > /tmp/fifo
```

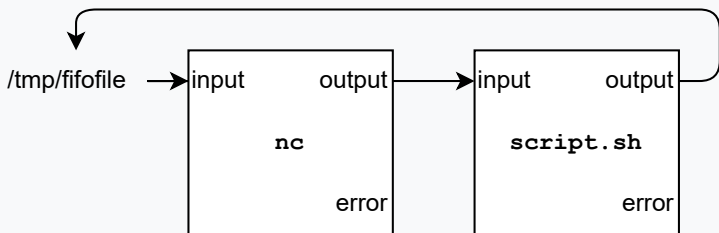
Input and output 🎁 final boss

We can not create I/O loops but... Everything is a file

- We can create *named pipes* (FIFO files)
- Other processes can read these streams as input
- (No data is written on disk; kernel buffers only)

```
mkfifo /tmp/fifo
```

```
nc 127.0.0.1 80 < /tmp/fifo | script.sh > /tmp/fifo
```



Multiple commands

Sometimes we want to run multiple commands...

```
command1 ; command2
```

```
command1 && command2
```

What's the difference? What happens if the first command fails?

Modifying behavior

How can we modify the behavior of a program?

Modifying behavior

How can we modify the behavior of a program?

Parameters and options (`ps aux`, `sort -r`)

What if multiple programs/processes need the same configurable information? What if I am lazy and don't want to use parameters for each command line?

Modifying behavior

How can we modify the behavior of a program?

Parameters and options (`ps aux, sort -r`)

What if multiple programs/processes need the same configurable information? What if I am lazy and don't want to use parameters for each command line?

env variables!

A process starts in an *environment*, i.e. a specific context, including variables set by the OS and/or the user.

Environment variables

Variables are scoped: either local to the Shell process, or available system-wide

```
# Display all environment variables
printenv
# Set an env var, local for the shell process
MYVAR="hello"
# Display a single variable
echo $MYVAR
# Nothing shows up
printenv | grep hello
# If we want to make it available
# for all children processes
export MYVAR="hello"
```

Environment variables

What if I want to always set a given variable?

Environment variables

What if I want to always set a given variable?

Put its declaration in `~/ .bashrc`

Following the PATH

Common executable are organized in different directories (cf. previous class)

```
which ls  
# /usr/bin/ls
```

To know where to look for them, the OS keeps a list in the PATH environment variable

```
echo $PATH  
# /usr/local/bin /usr/bin /bin [...]
```

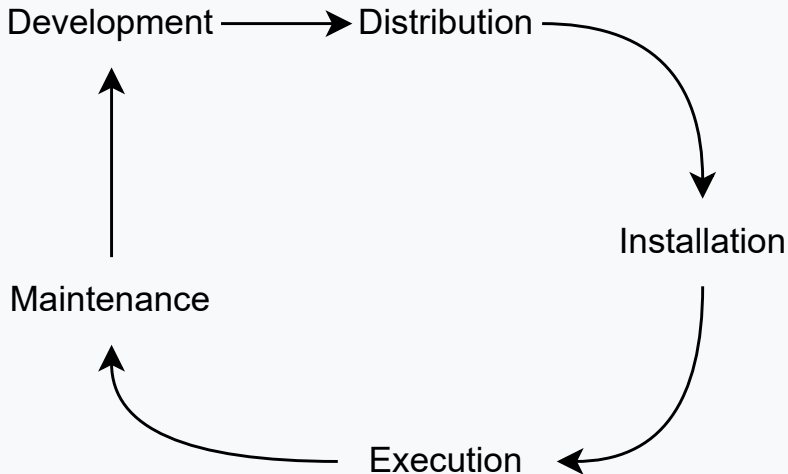
System interactions TL;DR

- We interact with the system through a terminal running a shell
- Programs and commands usually have an input / output
- We can plug these I/O into other commands
- To change the behavior of a command, we can use parameters, options, and environment variables

What is a Software?

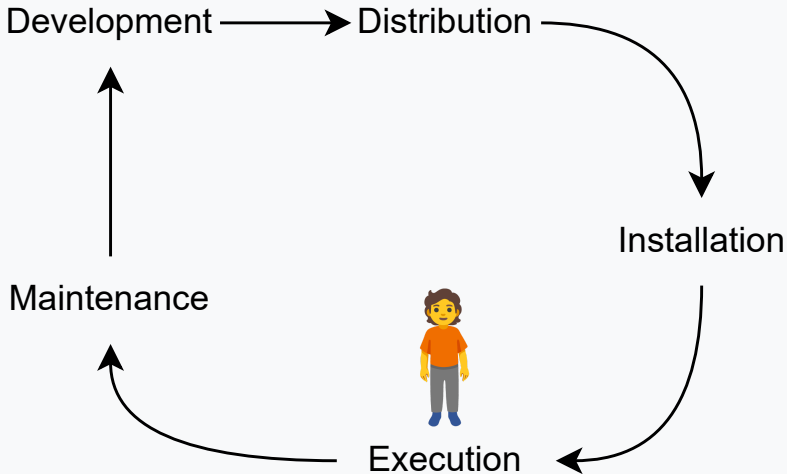
- A set of instructions that tell a computer how to perform specific tasks
- Multiples types of it
 - *System software*: Operating systems, drivers, utilities
 - *Application software*: Programs that help users perform tasks (e.g., browsers, text editors)
 - *Development software*: softwares building softwares: compilers, IDEs, debuggers

Software lifecycle



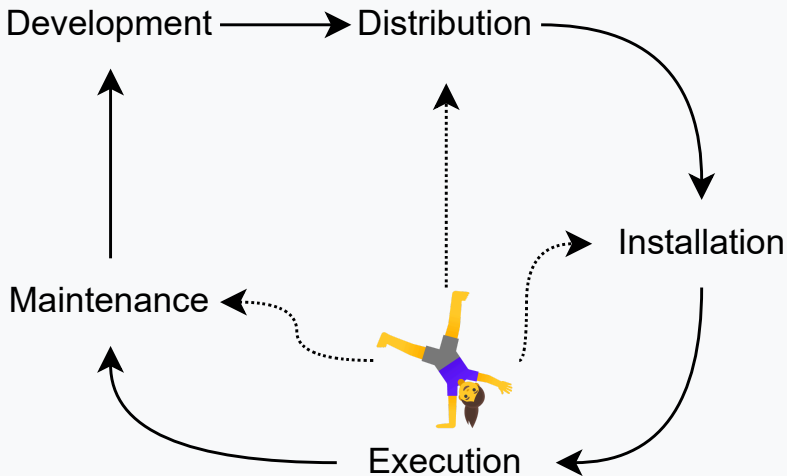
Software lifecycle

Users are not only interested in the execution of the program...



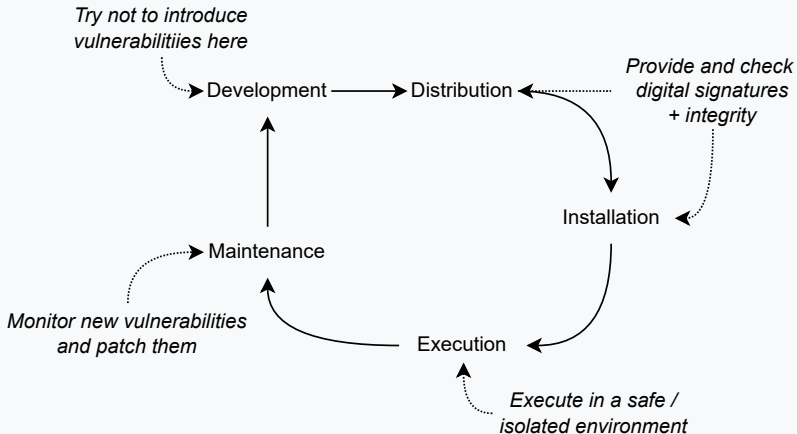
Software lifecycle

... but with other aspects of the lifecycle



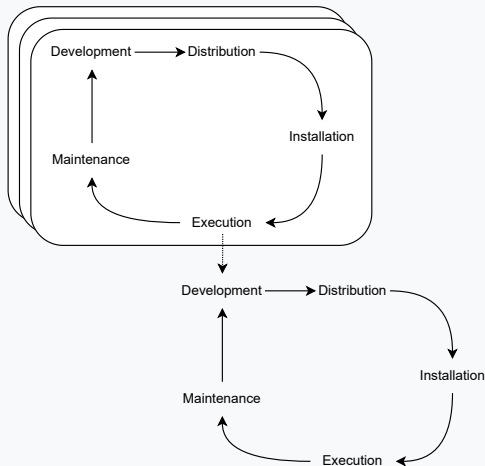
Software lifecycle

Security is relevant for each part of the lifecycle



Software lifecycle

Development is done using other software
(e.g., text editors, compilers, etc.)



Package Managers

For users:

- Installing software manually is *time-consuming* and requires *specific knowledge*

For developers:

- Numerous steps to distribute and install correctly
- Supporting every single flavor of user-equipment is hard
- Debugging users is time-consuming

Package Managers

Hence, **package managers**:

- Tools that automate the process of installing, updating, configuring, and removing *software packages*
- Simplify software management and ensure consistency across systems

Helps with questions like:

- What if a new version of a package is available?
- How should dependencies be installed and shared between packages?
- How can we verify the authenticity and integrity of packages?

Package Managers

The packages are organized in **repositories**:

- Centralized servers hosting verified software packages and metadata (e.g., version, authors, publication date)
- Repositories can be official (maintained by the OS distribution community) or custom/private

Package Managers

Some package managers:

- apt: Debian/Ubuntu-based distributions
- pacman: Arch Linux
- dnf: Fedora & Red Hat-based distributions
- WinGet: Windows package manager by Microsoft
- brew: MacOS

Package Managers

Some package managers:

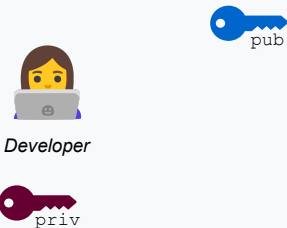
- apt: Debian/Ubuntu-based distributions
- pacman: Arch Linux
- dnf: Fedora & Red Hat-based distributions
- WinGet: Windows package manager by Microsoft
- brew: MacOS

The same concepts apply for programming languages:

- pip: Python
- cargo: Rust
- npm: JavaScript
- ...

Package Managers security

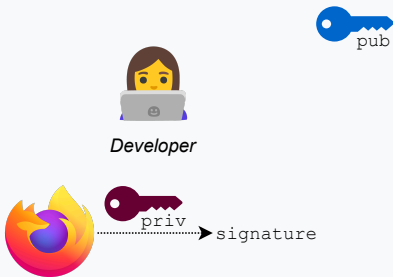
How can I be sure that the software I download and install is authentic and unmodified?



The developer has a pair of cryptographic keys:

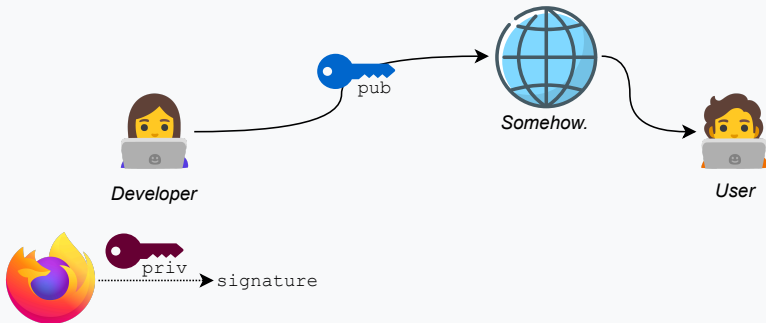
- *Private key*: kept secret
- *Public key*: distributed to everyone

Package Managers security



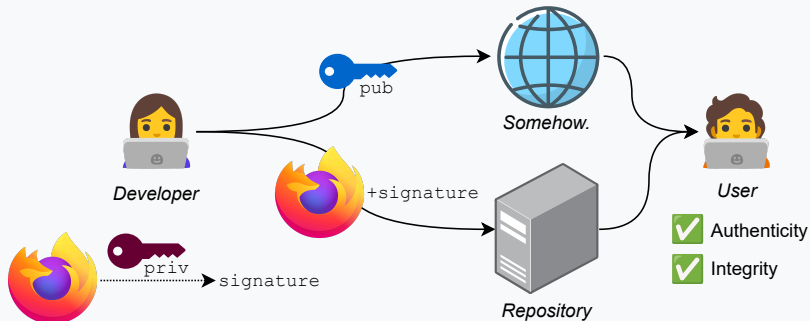
The developer uses the **private key** to create a **digital signature** for the software package

Package Managers security



And the developer share the public key with the world

Package Managers security



When downloading the package, the user also downloads the **signature**. The user verifies:

- **Integrity:** The package has not been altered or corrupted since it was signed

Debugging

Using a system \Rightarrow debugging a system

Debugging



1

Allez sur wooclap.com

2

Entrez le code d'événement
dans le bandeau supérieur

Code d'événement
MOLIJD

<https://app.wooclap.com/MOLIJD>

A side note on LLMs

Using a system \Rightarrow debugging a system

Debugging a system \Rightarrow understanding a system

A side note on LLMs

As of November 2025, LLMs (e.g., ChatGPT, Claude) *are able to help you debug programs...*

...until they can't (e.g., when a new version was not used for training, the setup/problem is completely unknown, the configuration is too complex, or due to hallucination).

In that case, **you *must* be able to debug alone.**

Learning how to debug “manually” is a mandatory skill that *cannot* be replaced by blindly copying error messages into an LLM prompt.

(Same issue with StackOverflow: a quick copy/paste without understanding may lead to catastrophic failures)

A side note on LLMs

As of November 2025, LLMs (e.g., ChatGPT, Claude) *are able to help you debug programs...*

...until they can't (e.g., when a new version was not used for training, the setup/problem is completely unknown, the configuration is too complex, or due to hallucination).

In that case, **you *must* be able to debug alone.**

Learning how to debug “manually” is a mandatory skill that *cannot* be replaced by blindly copying error messages into an LLM prompt.

*Or at least *understand* when the LLM's suggestions are nonsense.

Debugging 101

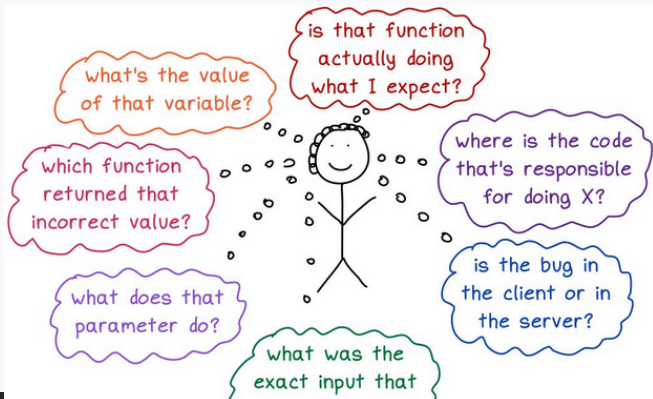
It's never magic

1. *Read the error message*
2. Understand the error message
3. Read the error message again because you actually didn't read it the first time
4. If not directly understood, look for more information online/in the documentation
5. *Change something*
6. Try again

Debugging 101

A good method from Julia Evans:

1. Come up with a question about the bug
2. Figure out how to get the answer to the question
3. Repeat until I understand the bug



Example: python syntax

```
~/Doc/work/cours/CS/ordi/demo/debug main !2 ?3 > python3 syntax.py 1
Traceback (most recent call last):
  File "/home/s/Documents/work/cours/CS/ordi/demo/debug/syntax.py", line 6, in <module>
    print_something_cool()
    ~~~~~~^
TypeError: print_something_cool() missing 1 required positional argument: 'n'
```

Example: python

- Where is the issue?
- What is it about?

```
Traceback (most recent call last):  
  File "/home/s/Documents/work/cours/CS/ordi/demo/debug/syntax.py", line 6, in <module>  
    print_something_cool()  
    ~~~~~  
TypeError: print_something_cool() missing 1 required positional argument: 'n'
```

Most programming languages error traces include:

- The line number
- The last function name
- The error type

Example: grub

```
sudo update-grub
[sudo] password for me:
Sourcing file `/etc/default/grub'
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-4.15.0-72-generic
Found initrd image: /boot/initrd.img-4.15.0-72-generic
Found linux image: /boot/vmlinuz-4.15.0-60-generic
Found initrd image: /boot/initrd.img-4.15.0-60-generic
Found linux image: /boot/vmlinuz-4.15.0-20-generic
Found initrd image: /boot/initrd.img-4.15.0-20-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
error: syntax error.
error: Incorrect command.
error: syntax error.
Syntax error at line 145
Syntax errors are detected in generated GRUB config file.
Ensure that there are no errors in /etc/default/grub
and /etc/grub.d/* files or please file a bug report with
/boot/grub/grub.cfg.new file attached.
```

Example: grub

```
GNU GRUB  version 2.12-9
```

```
Minimal BASH-like line editing is supported. For the first word,  
TAB lists possible command completions. Anywhere else TAB lists  
possible device or file completions. To enable less(1)-like paging,  
"set pager=1".
```

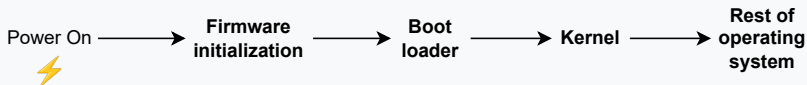
```
grub> _
```

Example: grub

```
GNU GRUB version 2.12-9

Minimal BASH-like line editing is supported. For the first word,
TAB lists possible command completions. Anywhere else TAB lists
possible device or file completions. To enable less(1)-like paging,
"set pager=1".

grub> _
```



How would you deal with such a bug? What if the disk is encrypted?

Example: installing a new program on Linux

Sometimes, programs are not available through the package manager

```
user@DebianNetInstall:~$ node
bash: node: command not found
```

```
user@DebianNetInstall:~$ mv ~/Downloads/node-v22.21.0-linux-x64/bin/node /opt/
mv: cannot move '/home/user/Downloads/node-v22.21.0-linux-x64/bin/node' to '/opt/node': Permission denied
```

```
user@DebianNetInstall:~$ sudo mv ~/Downloads/node-v22.21.0-linux-x64/bin/node /opt/
[sudo] password for user:
user is not in the sudoers file.
```

Logs

Sometimes:

- The error message is incomplete / there is no error message
- Events already happened

logs to the rescue: what actually happened on the system

Logs: Linux

Common places:

- **journalctl**: view systemd journal logs
 - Logs for a specific service: `journalctl -u apache2`
 - Recent errors and warnings: `journalctl -xe`
- **/var/log/**: traditional log directory
 - Still used by some programs to store their own logs (e.g., `/var/log/apache2/` for access/error logs)
- **dmesg**: kernel ring buffer messages
 - Useful for hardware / driver-related issues

Logs: Windows

Common places:

- **Event Viewer** (`eventvwr.msc`)
 - GUI, central place, neatly organized
 - Both Windows/System logs and applications/services logs
 - Filter or search by Event ID, Source, or Level (Error, Warning, Info)
- **PowerShell**: `Get-EventLog / Get-WinEvent`
 - `Get-EventLog -LogName System -n 20`
 - `Get-WinEvent -LogName Application | where LevelDisplayName -eq "Error"`
- **Log files** in `C:\Windows\System32\winevt\Logs`

Getting help

For existing programs:

```
man command # manual
```

```
command --help # or -help
```

```
# Increase the logs
```





```
command --verbose
```

```
command -v
```

📖 <https://explainshell.com/> (uses manual entries to explain commands & parameters)

For your programs: use the right tools, and abuse prints

Resources

-  The Pocket Guide to Debugging, Julia Evans <https://jvns.ca/blog/2022/12/21/new-zine--the-pocket-guide-to-debugging/>
-  What is the exact difference between a 'terminal', a 'shell', a 'tty' and a 'console'? <https://unix.stackexchange.com/questions/4126/what-is-the-exact-difference-between-a-terminal-a-shell-a-tty-and-a-console>
-  Is cmd.exe a shell, a terminal emulator or both?
<https://stackoverflow.com/a/31089434>
-  Windows Event Viewer demo
<https://learn.microsoft.com/en-us/shows/inside/event-viewer>