# Introduction to Operating Systems

## 4. Command Line Interface

2026-11-03

USO Team

# Bibliography


1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Sections 2.1 - 2.2 - *The Bourne Shell*, *Using the Shell*
     - Section 2.4.2 - *Shell Globbing ("Wildcards")*
     - Sections 2.8 - 2.10 - *Environment and Shell Variables*, *The Command Path*, *Special Characters*
     - Section 2.16 - *Listing and Manipulating Processes*
   - Chapter 3 - *Devices*
     - Section 3.4 - *Device Name Summary* (terminals)
   - Chapter 11 - *Introduction to Shell Scripts*
     - Section 11.2 - *Quoting and Literals*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 4 - *Procese*
     - Section 4.3.3 - *Foreground și background*
     - Sections 4.5.2, 4.6.1
   - Chapter 7 - *Interfața în linia de comandă*
     - Sections 7.1, 7.2.3 - 7.2.5
   - Chapter 9 - *Pornirea sistemului*
     - Section 9.6.1 - *Pornirea terminalelor de login*


 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

*It was a mistake to think that GUIs ever would, could, or even should, eliminate CLIs.*

Jeffrey Snover (architect of Windows PowerShell)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Command Line Interface

- Terminal & Shell
- Foreground & Background
- Sessions and Terminals
- Signals
- Environment Variables
- The Command Line
- Quoting, Escaping and Globbing
- Running a Command

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Terminal & Shell

*How we talk to the operating system*

# Bibliography

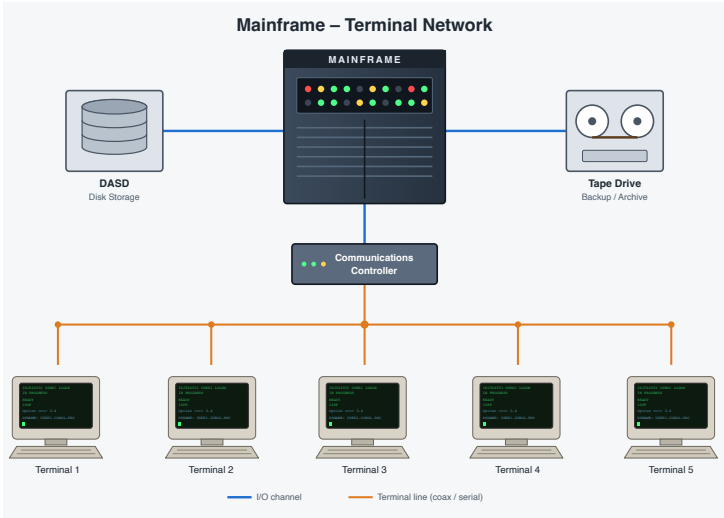
For this section


1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Sections 2.1 - 2.2 - *The Bourne Shell*, *Using the Shell*
   - Chapter 3 - *Devices*
     - Section 3.4 - *Device Name Summary* (terminals)
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 4 - *Procese*
     - Section 4.6.1 - *Terminale*
   - Chapter 7 - *Interfața în linia de comandă*
     - Section 7.1 - *Shellul. Funcționarea shellului*
   - Chapter 9 - *Pornirea sistemului*
     - Section 9.6.1 - *Pornirea terminalelor de login*


 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Classic* Terminal

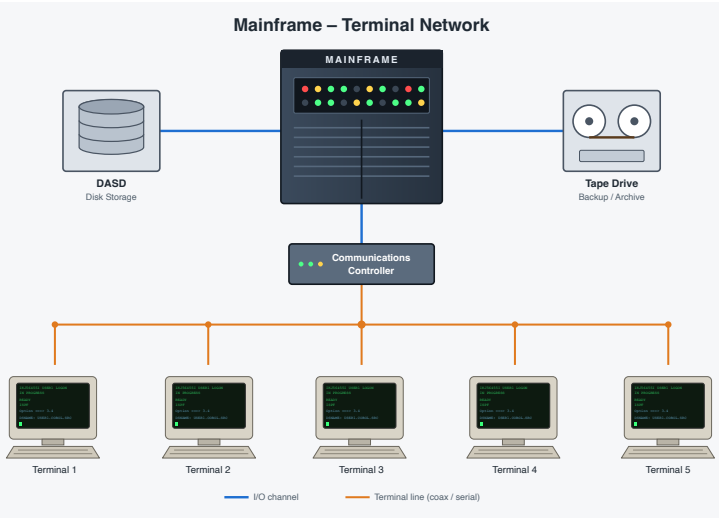
Many users, one computer




 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Classic* Terminal

Many users, one computer

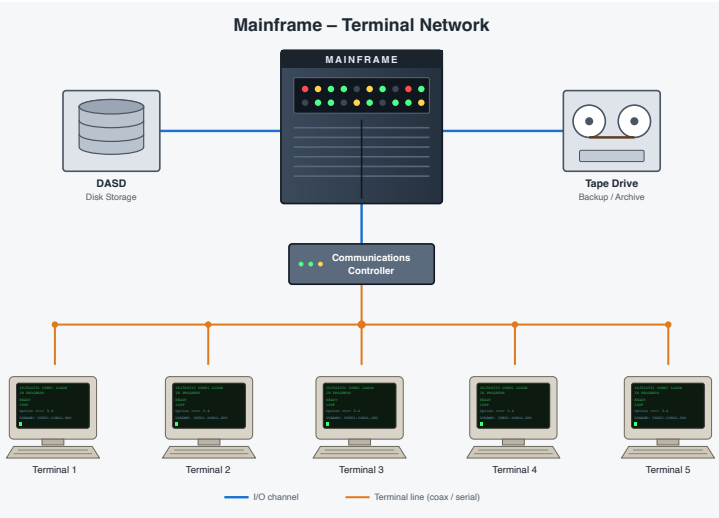



- one large computer (the *mainframe*), many users

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Classic* Terminal

Many users, one computer

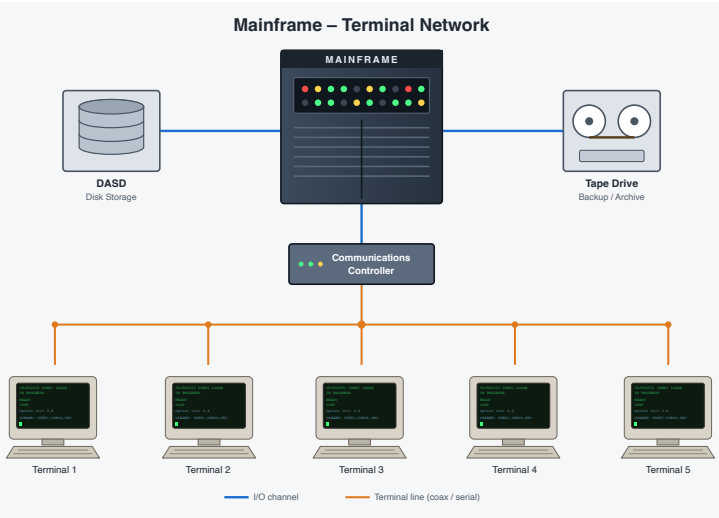



- one large computer (the *mainframe*), many users
- every user has a **terminal**: a keyboard and a printer or a screen, **no processor**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Classic* Terminal

Many users, one computer

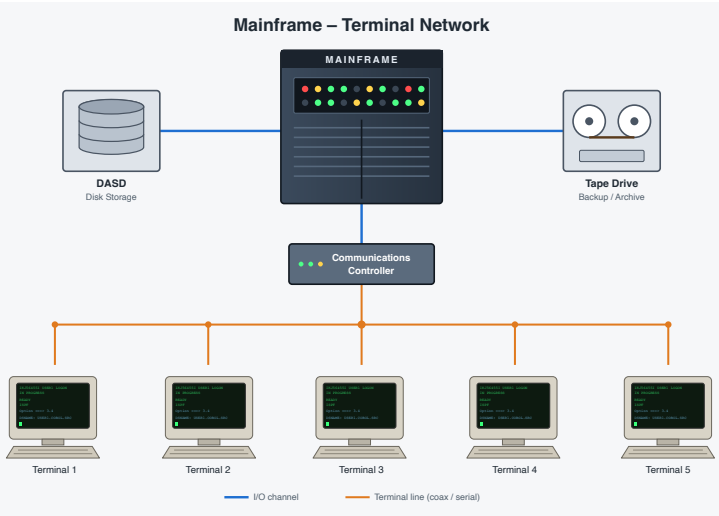



- one large computer (the *mainframe*), many users
- every user has a **terminal**: a keyboard and a printer or a screen, **no processor**
- the terminals were *TeleTypes* (**TTY**): they send the keys to the computer and print what it sends back

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Classic* Terminal

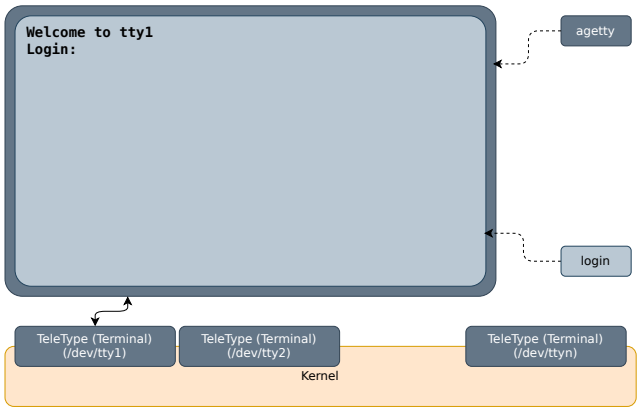
Many users, one computer




- one large computer (the *mainframe*), many users
- every user has a **terminal**: a keyboard and a printer or a screen, **no processor**
- the terminals were *TeleTypes* (**TTY**): they send the keys to the computer and print what it sends back
- 💡 Linux still calls terminals `tty`: `/dev/tty1`, `/dev/tty2`, ...

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Real* Terminal

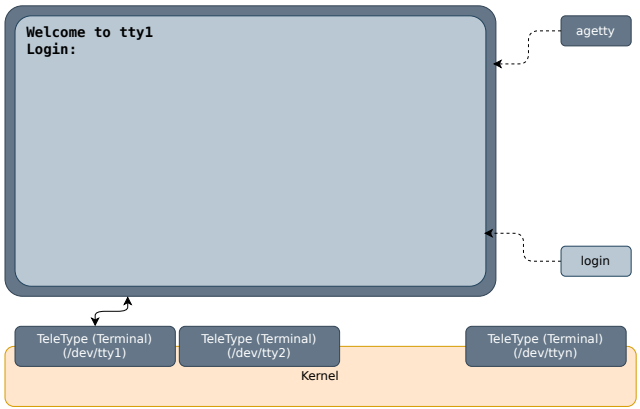



```
[PID 1]  systemd (init)
     # The first user-space process
     # started by the kernel
```

- the kernel provides several *virtual terminals*: `/dev/tty1` ... `/dev/tty6`
- they share the same screen and keyboard

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Real* Terminal

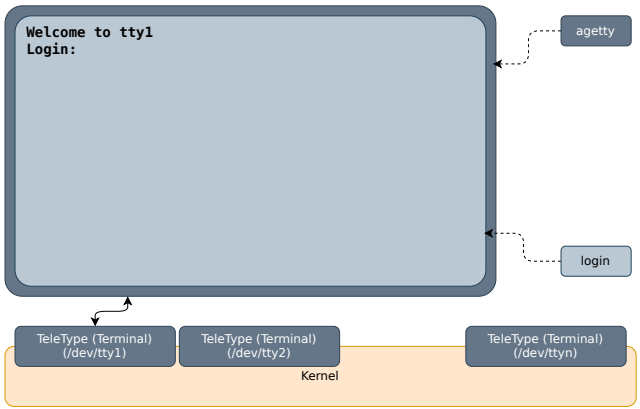



```
[PID 1]  systemd (init)
    # The first user-space process
    # started by the kernel
└── [PID 20+]  agetty tty1
        # Waits for a login on /dev/tty1
```

- the kernel provides several *virtual terminals*: `/dev/tty1` ... `/dev/tty6`
- they share the same screen and keyboard

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Real* Terminal

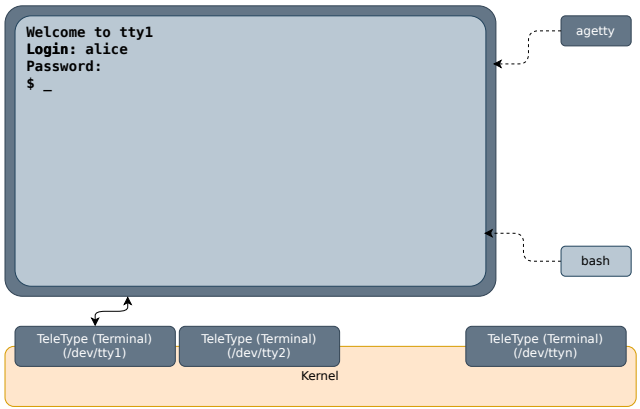



```
[PID 1]  systemd (init)
     # The first user-space process
     # started by the kernel
├── [PID 20+]  agetty tty1
├── [PID 21+]  agetty tty2
├── [PID ...]  ...
└── [PID 25+]  agetty tty6
       # One for every text terminal
```

- the kernel provides several *virtual terminals*: `/dev/tty1` ... `/dev/tty6`
- they share the same screen and keyboard
- `Alt` + `F1` ... `F6` changes the terminal

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Real* Terminal



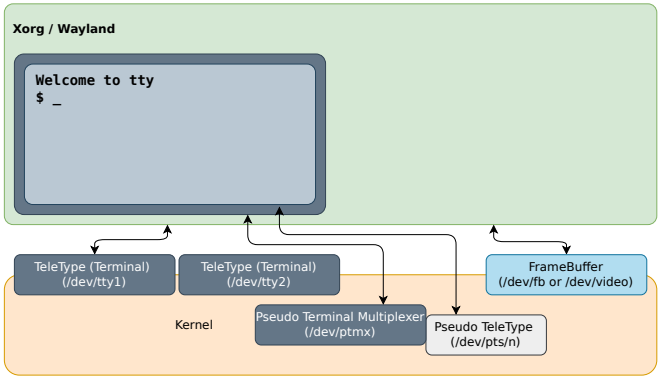

```
[PID 1]  systemd (init)
     # The first user-space process
     # started by the kernel
├── [PID 20+]  login alice
│      # agetty became login (exec)
│      └── [PID 30+]  bash
│             # alice's shell, on /dev/tty1
├── [PID 21+]  agetty tty2
├── [PID ...]  ...
└── [PID 25+]  agetty tty6
```

User `alice` logs in using `/dev/tty1`

- the kernel provides several *virtual terminals*: `/dev/tty1` ... `/dev/tty6`
- they share the same screen and keyboard
- `Alt` + `F1` ... `F6` changes the terminal

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Real* Terminal




- the kernel provides several *virtual terminals*: `/dev/tty1` ... `/dev/tty6`
- they share the same screen and keyboard
- `Alt` + `F1` ... `F6` changes the terminal

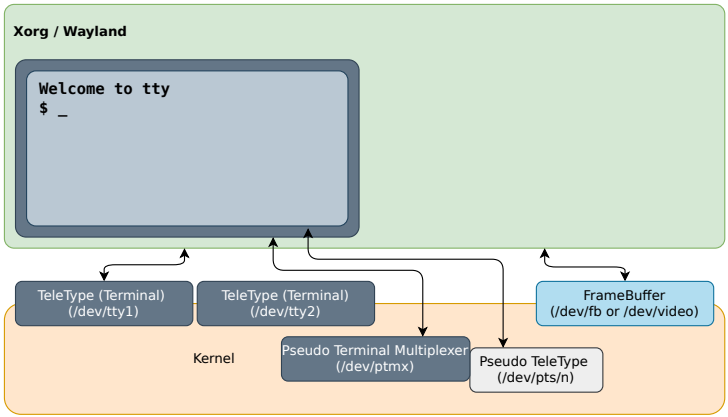
```
[PID 1]  systemd (init)
     # The first user-space process
     # started by the kernel
├── [PID 20+]  login alice
│     └── [PID 30+]  bash
├── [PID 21+]  agetty tty2
├── [PID ...]  ...
├── [PID 25+]  agetty tty6
└── [PID 8+]  gdm / sddm / lightdm
       # Graphical login
```

The graphical UI is another application, using one of the terminals

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Virtual* Terminal

Graphical Mode Linux




## Switch Modes

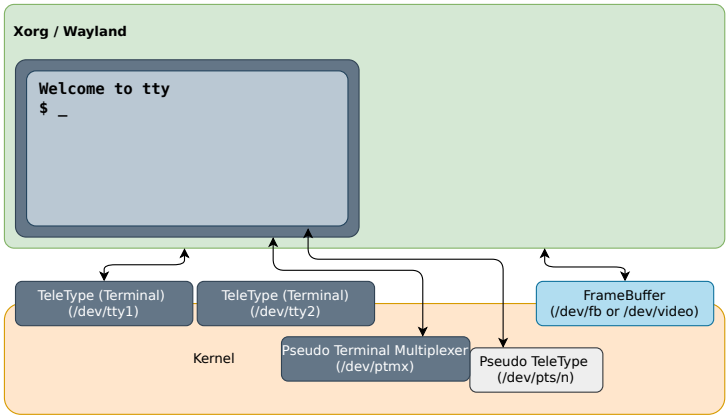
- Ctrl + Alt + F3 ... F6 → text mode
- Ctrl + Alt + F1 or F2 → back to the graphical mode (F7 on older systems)

*Xorg* or *Wayland* (display servers) take over a terminal (`/dev/tty1` or `/dev/tty2`) and draw on the screen through the framebuffer (`/dev/fb0`)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Virtual* Terminal

Graphical Mode Linux




*Xorg* or *Wayland* (display servers) take over a terminal (`/dev/tty1` or `/dev/tty2`) and draw on the screen through the framebuffer (`/dev/fb0`)

## Switch Modes

- `Ctrl` + `Alt` + `F3` ... `F6` → text mode
- `Ctrl` + `Alt` + `F1` or `F2` → back to the graphical mode (`F7` on older systems)

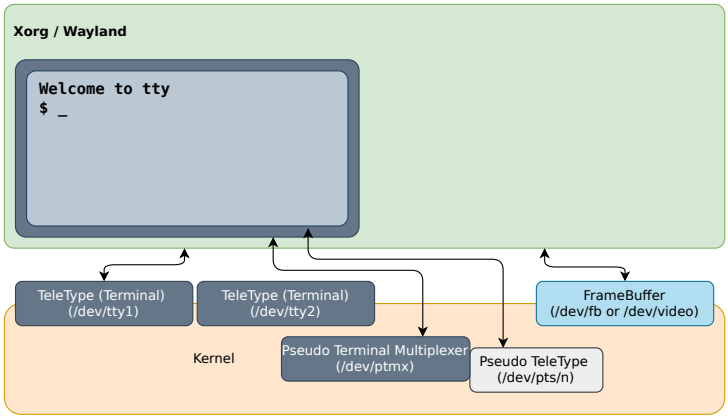
## Console Applications

- run in a *terminal emulator* (`kitty`, `ptyxis`, `alacritty`, ...)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Virtual* Terminal

Graphical Mode Linux




*Xorg* or *Wayland* (display servers) take over a terminal (`/dev/tty1` or `/dev/tty2`) and draw on the screen through the framebuffer (`/dev/fb0`)

## Switch Modes

- Ctrl + Alt + F3 ... F6 → text mode
- Ctrl + Alt + F1 or F2 → back to the graphical mode (F7 on older systems)

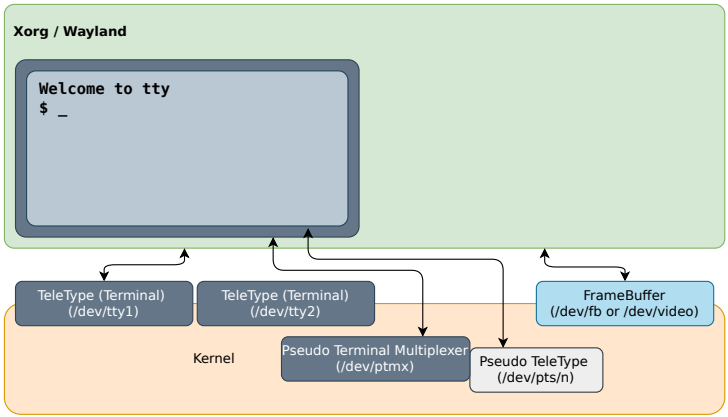
## Console Applications

- run in a *terminal emulator* (`kitty`, `ptyxis`, `alacritty`, ...)
- the terminal emulator is a **graphical** application

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The *Virtual* Terminal

Graphical Mode Linux




*Xorg* or *Wayland* (display servers) take over a terminal (`/dev/tty1` or `/dev/tty2`) and draw on the screen through the framebuffer (`/dev/fb0`)

## Switch Modes

- Ctrl + Alt + F3 ... F6 → text mode
- Ctrl + Alt + F1 or F2 → back to the graphical mode (F7 on older systems)

## Console Applications

- run in a *terminal emulator* (`kitty`, `ptyxis`, `alacritty`, ...)
- the terminal emulator is a **graphical** application
- it provides a *pseudo terminal*, `/dev/pts/n`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Shell

Operating System Interaction



*A process whose main purpose is to allow the interaction with the Operating System*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Shell

Operating System Interaction



*A process whose main purpose is to allow the interaction with the Operating System*

## ⌨ Command Line

- `bash`, `sh`, `zsh`, `fish`
- `cmd`
- `powershell`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Shell

Operating System Interaction



*A process whose main purpose is to allow the interaction with the Operating System*

## ⌨️ Command Line

- `bash`, `sh`, `zsh`, `fish`
- `cmd`
- `powershell`

## 🔤 Text User Interface

- `mc`
- `nnn`
- `yazi`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Shell

Operating System Interaction



*A process whose main purpose is to allow the interaction with the Operating System*

## ⌨️ Command Line

- `bash`, `sh`, `zsh`, `fish`
- `cmd`
- `powershell`

## 🔤 Text User Interface

- `mc`
- `nnn`
- `yazi`

## 🖱️ Graphical

- `explorer.exe`, Finder, `nautilus`
- Total Commander
- Control Panel, Settings

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Terminal & Shell

Two different programs



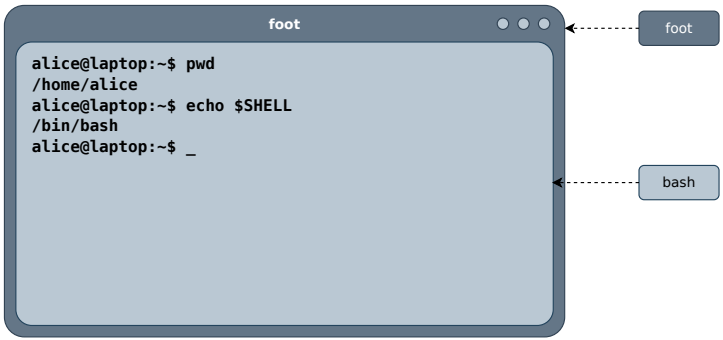

## Terminal (`foot`)

- the window, the font, the colors
- reads the keyboard and draws the text

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Terminal & Shell

Two different programs




## Terminal (`foot`)

- the window, the font, the colors
- reads the keyboard and draws the text

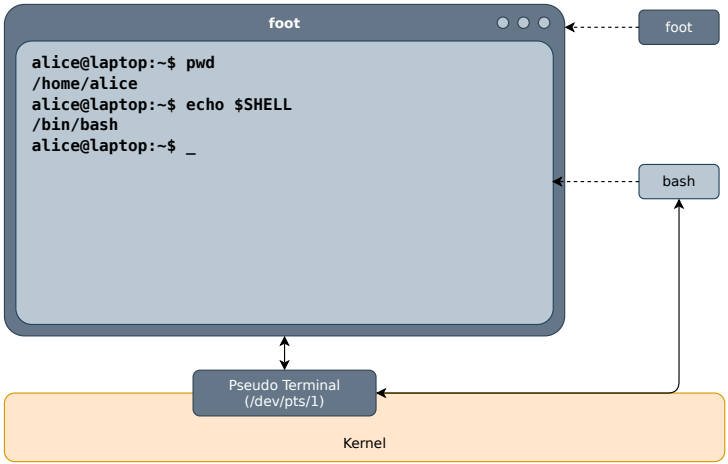
## Shell (`bash`)

- the text inside the window: the prompt and the output
- runs the commands

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Terminal & Shell

Two different programs




## Terminal (`foot`)

- the window, the font, the colors
- reads the keyboard and draws the text

## Shell (`bash`)

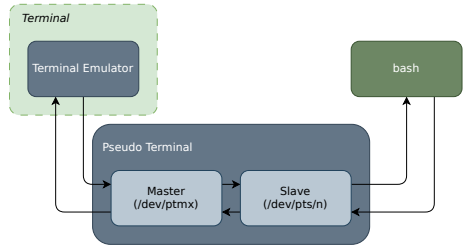
- the text inside the window: the prompt and the output
- runs the commands
- they talk through the *pseudo terminal* `/dev/pts/1`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Terminal Emulator

Provides a pseudo terminal

`kitty`, `ptyxis`, `alacritty`, ...




`bash`, `sh`, `zsh`, ...

- only see a terminal: `/dev/pts/n`

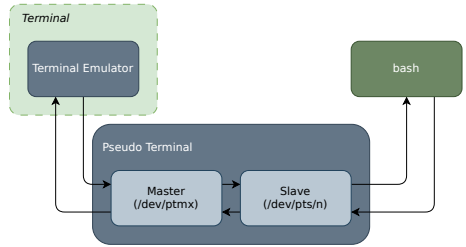
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Terminal Emulator

Provides a pseudo terminal

`kitty`, `ptyxis`, `alacritty`, ...

1. opens `/dev/ptmx`, the kernel creates a *pseudo terminal*: a **master** (kept by the emulator) and a **slave** (`/dev/pts/n`)




`bash`, `sh`, `zsh`, ...

- only see a terminal: `/dev/pts/n`

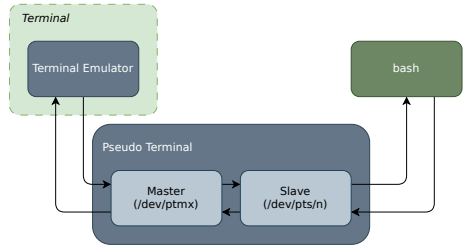
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Terminal Emulator

Provides a pseudo terminal

`kitty`, `ptyxis`, `alacritty`, ...

1. opens `/dev/ptmx`, the kernel creates a *pseudo terminal*: a **master** (kept by the emulator) and a **slave** (`/dev/pts/n`)
2. starts the shell (`fork` + `exec`) with `0`, `1` and `2` connected to `/dev/pts/n`




`bash`, `sh`, `zsh`, ...

- only see a terminal: `/dev/pts/n`

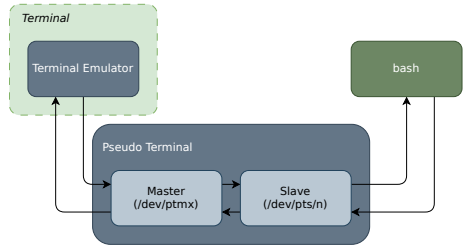
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Terminal Emulator

Provides a pseudo terminal

`kitty`, `ptyxis`, `alacritty`, ...

1. opens `/dev/ptmx`, the kernel creates a *pseudo terminal*: a **master** (kept by the emulator) and a **slave** (`/dev/pts/n`)
2. starts the shell (`fork` + `exec`) with `0`, `1` and `2` connected to `/dev/pts/n`
3. the keys you press are written into the **master**, the shell reads them from `stdin`




`bash`, `sh`, `zsh`, ...

- only see a terminal: `/dev/pts/n`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Terminal Emulator

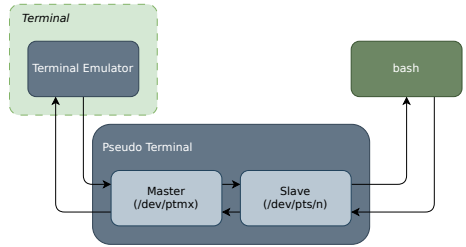
Provides a pseudo terminal

`kitty`, `ptyxis`, `alacritty`, ...

1. opens `/dev/ptmx`, the kernel creates a *pseudo terminal*: a **master** (kept by the emulator) and a **slave** (`/dev/pts/n`)
2. starts the shell (`fork` + `exec`) with `0`, `1` and `2` connected to `/dev/pts/n`
3. the keys you press are written into the **master**, the shell reads them from `stdin`
4. the shell writes into `stdout`, the emulator reads it from the **master** and draws the text in its window

`bash`, `sh`, `zsh`, ...

- only see a terminal: `/dev/pts/n`




 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Terminal Emulator

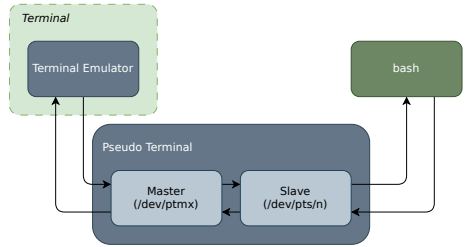
Provides a pseudo terminal

`kitty`, `ptyxis`, `alacritty`, ...

1. opens `/dev/ptmx`, the kernel creates a *pseudo terminal*: a **master** (kept by the emulator) and a **slave** (`/dev/pts/n`)
2. starts the shell (`fork` + `exec`) with `0`, `1` and `2` connected to `/dev/pts/n`
3. the keys you press are written into the **master**, the shell reads them from `stdin`
4. the shell writes into `stdout`, the emulator reads it from the **master** and draws the text in its window

`bash`, `sh`, `zsh`, ...

- only see a terminal: `/dev/pts/n`




```
$ tty                    # the terminal of the shell
/dev/pts/1
$ ps                     # the processes of this
terminal
    PID TTY          TIME CMD
   4011 pts/1    00:00:00 bash
   4242 pts/1    00:00:00 ps
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Signals

*notifications for the processes*

# Bibliography

For this section


1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Section 2.16 - *Listing and Manipulating Processes*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 4 - *Procese*
     - Section 4.5.2 - *Semnale*


 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Signals



✉ the OS sends a message to a process

- a message sent from the OS to a process
- a number
- may have an *extra payload number*

**List of signals**

```
$ kill -l # sends a signal, does not kill a process
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Signals



✉ the OS sends a message to a process

- a message sent from the OS to a process
- a number
- may have an *extra payload number*

**List of signals**

```
$ kill -l # sends a signal, does not kill a process
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Signals



✉ the OS sends a message to a process

- a message sent from the OS to a process
- a number
- may have an *extra payload number*

**List of signals**

```
$ kill -l # sends a signal, does not kill a process
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
...             64) SIGRTMAX
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Handling Signals

override the default signal handler

When a ✉️ signal arrives, the process will

```
#!/bin/bash

# the signal handler
function no_interruptions_please () {
    echo "No interruptions please"
}

# register the signal handler
trap no_interruptions_please SIGINT

while true; do
  echo "working"
  sleep 1
done

# exit (default action for SIGINT)
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Handling Signals

override the default signal handler

When a ✉️ signal arrives, the process will

1. 🛑 stop whatever it is doing
2. 🚥 execute
   - the **default action** (*if no signal handler is registered*)

```
#!/bin/bash

# the signal handler
function no_interruptions_please () {
    echo "No interruptions please"
}

# register the signal handler
trap no_interruptions_please SIGINT

while true; do
  echo "working"
  sleep 1
done

# exit (default action for SIGINT)
```

💡 `Ctrl` + `C` sends `SIGINT`, the default action stops the script

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Handling Signals

override the default signal handler

When a ✉️ signal arrives, the process will

1. 🛑 stop whatever it is doing
2. 🚥 execute
   - the **default action** (*if no signal handler is registered*)
   - OR the **signal handler** and continue its task

```bash
#!/bin/bash

# the signal handler
function no_interruptions_please () {
    echo "No interruptions please"
}

# register the signal handler
trap no_interruptions_please SIGINT

while true; do
  echo "working"
  sleep 1
done

# exit (default action for SIGINT)
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Handling Signals

override the default signal handler

When a ✉️ signal arrives, the process will

1. 🛑 stop whatever it is doing
2. 📼 execute
   - the **default action** (*if no signal handler is registered*)
   - OR the **signal handler** and continue its task

```
trap function_name SIGNAL
```

```
#include <signal.h>
void handler(int signum) { /* ... */ }
signal(SIGINT, handler);
```

```
#!/bin/bash

# the signal handler
function no_interruptions_please () {
    echo "No interruptions please"
}

# register the signal handler
trap no_interruptions_please SIGINT

while true; do
  echo "working"
  sleep 1
done

# exit (default action for SIGINT)
```

💡 Ctrl + C now prints the message, the script keeps working

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Default Signal Actions

| Default Action | Description |
|---|---|
| **Term** | Immediately ends the process. Can usually be caught or ignored. |
| **Term (Unblockable)** | Ends the process and **cannot** be caught, blocked, or ignored (`SIGKILL`). |
| **Core** | Terminates the process **and creates a core dump** for debugging. |
| **Stop** | Suspends the process until it receives `SIGCONT`. |
| **Stop (Unblockable)** | Suspends the process, and **cannot** be caught or ignored (`SIGSTOP`). |
| **Cont** | Resumes a process that was previously stopped. |
| **Ign** | Signal is discarded; no effect on the process. |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Linux Signals: Default Actions (1/2)

| # | Signal | Default Action | Description |
|---|---|---|---|
| 1 | `SIGHUP` | Term | Hangup detected on controlling terminal |
| 2 | `SIGINT` | Term | Interrupt from keyboard (`Ctrl` + `C`) |
| 3 | `SIGQUIT` | Core | Quit from keyboard (`Ctrl` + `\`) |
| 4 | `SIGILL` | Core | Illegal instruction |
| 5 | `SIGTRAP` | Core | Trace/breakpoint trap |
| 6 | `SIGABRT` | Core | Abort signal from `abort()` |
| 7 | `SIGBUS` | Core | Bus error (bad memory access) |
| 8 | `SIGFPE` | Core | Floating-point exception |
| 9 | `SIGKILL` | Term (Unblockable) | Kill signal |
| 10 | `SIGUSR1` | Term | User-defined signal 1 |
| 11 | `SIGSEGV` | Core | Invalid memory reference |
| 12 | `SIGUSR2` | Term | User-defined signal 2 |
| 13 | `SIGPIPE` | Term | Broken pipe |
| 14 | `SIGALRM` | Term | Timer signal from `alarm()` |
| 15 | `SIGTERM` | Term | Termination signal |
| 16 | `SIGSTKFLT` | Term | Stack fault on coprocessor (unused) |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Linux Signals: Default Actions (2/2)

| # | Signal | Default Action | Description |
|---|---|---|---|
| 17 | `SIGCHLD` | Ign | Child stopped or terminated |
| 18 | `SIGCONT` | Cont | Continue if stopped |
| 19 | `SIGSTOP` | Stop (Unblockable) | Stop process |
| 20 | `SIGTSTP` | Stop | Stop from terminal (`Ctrl` + `Z`) |
| 21 | `SIGTTIN` | Stop | Background read from tty |
| 22 | `SIGTTOU` | Stop | Background write to tty |
| 23 | `SIGURG` | Ign | Urgent condition on socket |
| 24 | `SIGXCPU` | Core | CPU time limit exceeded |
| 25 | `SIGXFSZ` | Core | File size limit exceeded |
| 26 | `SIGVTALRM` | Term | Virtual timer expired |
| 27 | `SIGPROF` | Term | Profiling timer expired |
| 28 | `SIGWINCH` | Ign | Window resize signal |
| 29 | `SIGIO` | Term | I/O now possible |
| 30 | `SIGPWR` | Term | Power failure |
| 31 | `SIGSYS` | Core | Bad system call |
| 34–64 | `SIGRTMIN` – `SIGRTMAX` | Term | Real-time signals (implementation-defined) |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Foreground & Background

*Run more than one process from the same shell*

# Bibliography

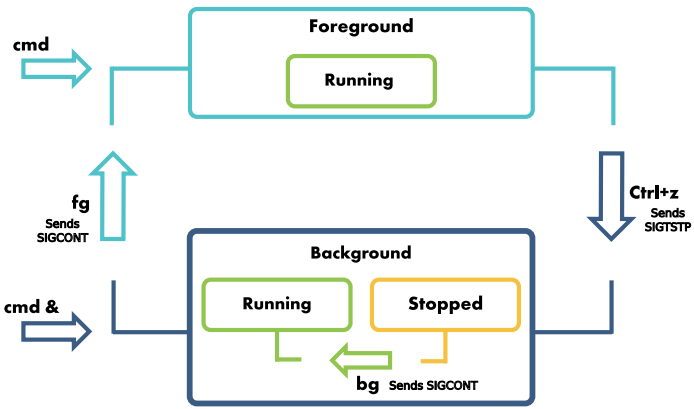
For this section


1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Section 2.16 - *Listing and Manipulating Processes*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 4 - *Procese*
     - Section 4.3.3 - *Foreground și background*
     - Section 4.5.2 - *Semnale*


 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Foreground and Background Process

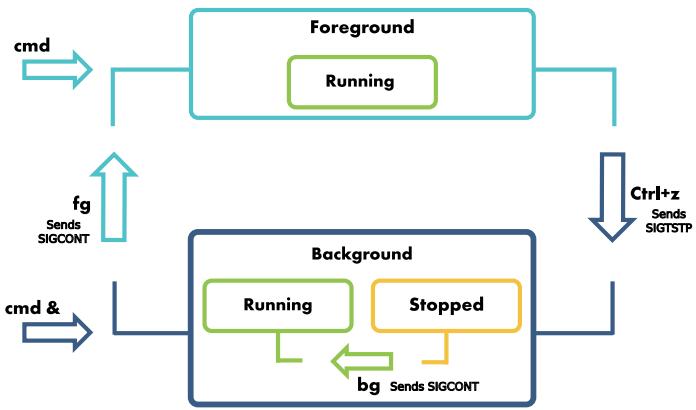
The terminal and the shell define this




 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Foreground and Background Process

The terminal and the shell define this



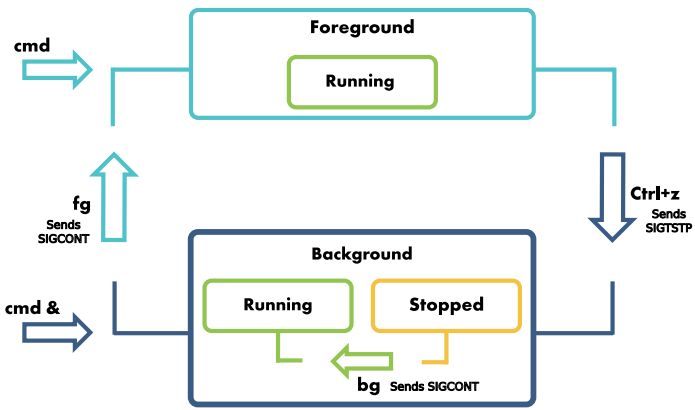

## Foreground

- **only one** process
- reads the keyboard (`stdin`)
- the shell *waits* for it

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Foreground and Background Process

The terminal and the shell define this




## Foreground

- **only one** process
- reads the keyboard (`stdin`)
- the shell *waits* for it

## Background

- **any number** of processes, called *jobs*
- *running* or *stopped*
- the shell does not wait, it shows the prompt

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Start a Job

Add `&` at the end: the shell starts the command and shows the prompt **right away**

```
$ sleep 100 &
[1] 4242
```



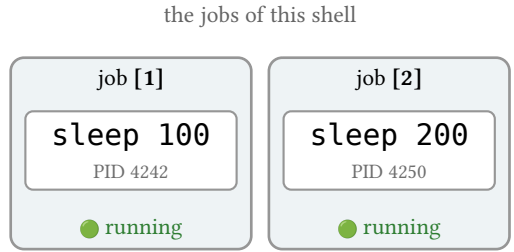

💡 `[1]` is the **job ID**, it exists only in this shell, `4242` is the **PID**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Start a Job

Add `&` at the end: the shell starts the command and shows the prompt **right away**

```
$ sleep 100 &
[1] 4242
$ sleep 200 &
[2] 4250
```



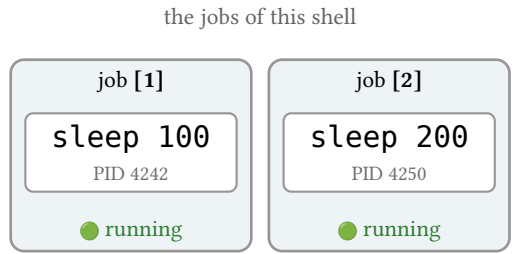

💡 a second job, `[2]`: the shell does not wait, both run at the same time

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Start a Job

Add `&` at the end: the shell starts the command and shows the prompt **right away**

```
$ sleep 100 &
[1] 4242
$ sleep 200 &
[2] 4250
$ jobs
[1]-  Running     sleep 100 &
[2]+  Running     sleep 200 &
```



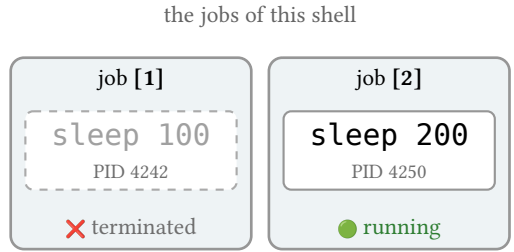

💡 `jobs` lists the jobs of **this** shell: `+` the current job (the last one), `-` the previous one

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Start a Job

Add `&` at the end: the shell starts the command and shows the prompt **right away**

```
$ sleep 100 &
[1] 4242
$ sleep 200 &
[2] 4250
$ jobs
[1]-  Running     sleep 100 &
[2]+  Running     sleep 200 &
$ kill %1
[1]-  Terminated  sleep 100
```




💡 `%1` means *job 1*: it works with `kill`, `fg` and `bg`, job `[2]` keeps running

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Make a Job

Move a foreground process to the background

- run the command
  - ⚠️ redirect its output, a job still writes to the same terminal

```
$ sleep 100 > /dev/null
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Make a Job

Move a foreground process to the background

- run the command
  - ⚠️ redirect its output, a job still writes to the same terminal
- press `Ctrl` + `Z` (sends `SIGTSTP`)
  - 🙄 the process might ignore it

```
$ sleep 100 > /dev/null
^Z
[1]+  Stopped     sleep 100 > /dev/null
```




 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Make a Job

Move a foreground process to the background

- run the command
  - ⚠️ redirect its output, a job still writes to the same terminal
- press `Ctrl` + `Z` (sends `SIGTSTP`)
  - 🙄 the process might ignore it
- find the job `ID` (`jobs`)

```
$ sleep 100 > /dev/null
^Z
[1]+  Stopped     sleep 100 > /dev/null
$ jobs
[1]+  Stopped     sleep 100 > /dev/null
```




 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Make a Job

Move a foreground process to the background

- run the command
  - ⚠️ redirect its output, a job still writes to the same terminal
- press `Ctrl` + `Z` (sends `SIGTSTP`)
  - 🙄 the process might ignore it
- find the job `ID` (`jobs`)
- run `bg %ID` (sends `SIGCONT`)

```
$ sleep 100 > /dev/null
^Z
[1]+  Stopped     sleep 100 > /dev/null
$ jobs
[1]+  Stopped     sleep 100 > /dev/null
$ bg %1
[1]+ sleep 100 > /dev/null &
```




 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Make a Job

Move a foreground process to the background

- run the command
  - ⚠️ redirect its output, a job still writes to the same terminal
- press `Ctrl` + `Z` (sends `SIGTSTP`)
  - 🙄 the process might ignore it
- find the job `ID` (`jobs`)
- run `bg %ID` (sends `SIGCONT`)
- run `jobs` to verify if it still runs

```
$ sleep 100 > /dev/null
^Z
[1]+  Stopped     sleep 100 > /dev/null
$ jobs
[1]+  Stopped     sleep 100 > /dev/null
$ bg %1
[1]+ sleep 100 > /dev/null &
$ jobs
[1]+  Running     sleep 100 > /dev/null &
```




 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Make a Job

Move a foreground process to the background

- run the command
  - ⚠️ redirect its output, a job still writes to the same terminal
- press `Ctrl` + `Z` (sends `SIGTSTP`)
  - 🙄 the process might ignore it
- find the job `ID` (`jobs`)
- run `bg %ID` (sends `SIGCONT`)
- run `jobs` to verify if it still runs

```
$ sleep 100 > /dev/null
^Z
[1]+  Stopped     sleep 100 > /dev/null
$ jobs
[1]+  Stopped     sleep 100 > /dev/null
$ bg %1
[1]+ sleep 100 > /dev/null &
$ jobs
[1]+  Running     sleep 100 > /dev/null &
$ fg %1
sleep 100 > /dev/null
```

## Bring it back in the foreground

- run `fg %ID`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Sessions and Terminals

*How the processes of a terminal are grouped*

# Bibliography

For this section

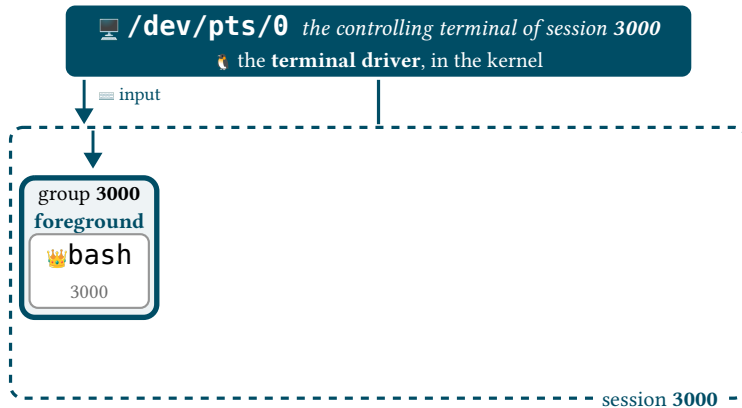

1. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 4 - *Procese*
     - Section 4.3.3 - *Foreground și background*
     - Section 4.5.2 - *Semnale*


 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Sessions and the Terminal

How the kernel groups the processes of a terminal

```
$ echo $$
3000
```




👑 the shell starts a new **session**: a session has **one** controlling terminal, `/dev/pts/0`

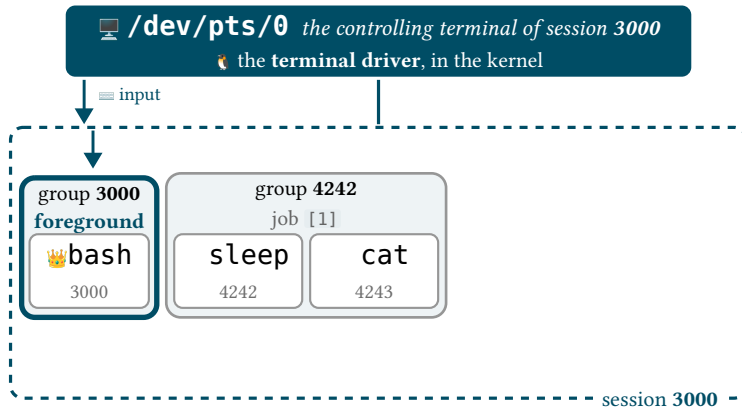
⌨️ the input of the terminal goes to the **foreground** group, now `bash` (`$$` is its PID)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Sessions and the Terminal

How the kernel groups the processes of a terminal

```
$ echo $$
3000
$ sleep 100 | cat &
[1] 4243
```



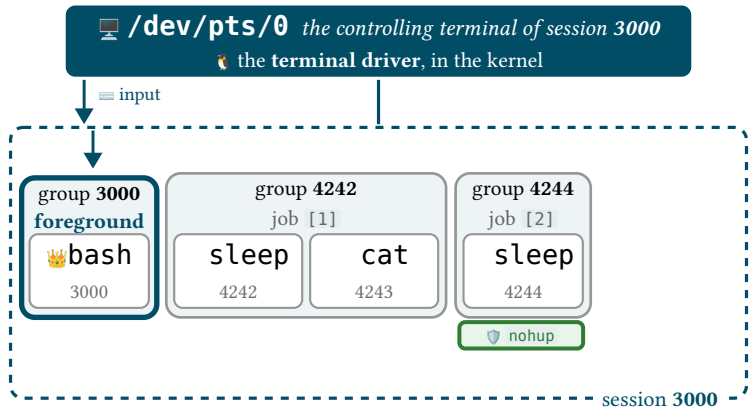

- a **process group**: one job, `sleep 100 | cat` is **one** group of two processes
- a signal (Ctrl + C) reaches the whole group

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Sessions and the Terminal

How the kernel groups the processes of a terminal

```
$ echo $$
3000
$ sleep 100 | cat &
[1] 4243
$ nohup sleep 200 > /dev/null &
[2] 4244
nohup: ignoring input
```



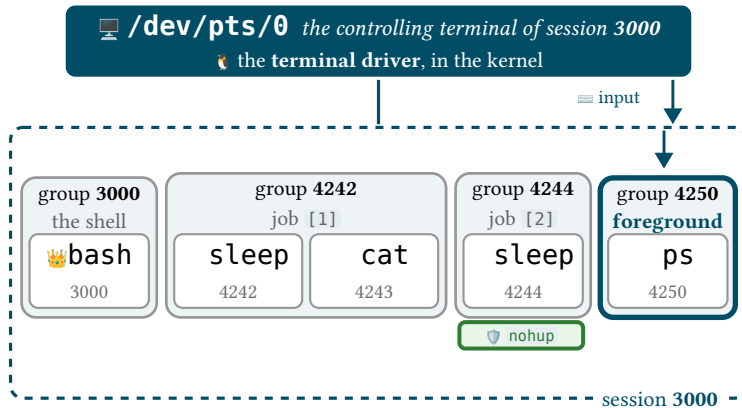

`nohup` makes the program **ignore** `SIGHUP`: a shield against it

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Sessions and the Terminal

How the kernel groups the processes of a terminal

```
$ echo $$
3000
$ sleep 100 | cat &
[1] 4243
$ nohup sleep 200 > /dev/null &
[2] 4244
nohup: ignoring input
$ ps -o pid,pgid,sid,stat,comm
    PID    PGID     SID STAT COMMAND
   3000    3000    3000 Ss   bash
   4242    4242    3000 S    sleep
   4243    4242    3000 S    cat
   4244    4244    3000 S    sleep
   4250    4250    3000 R+   ps
```




💡 a **session**: all the groups of one terminal, the same `SID`

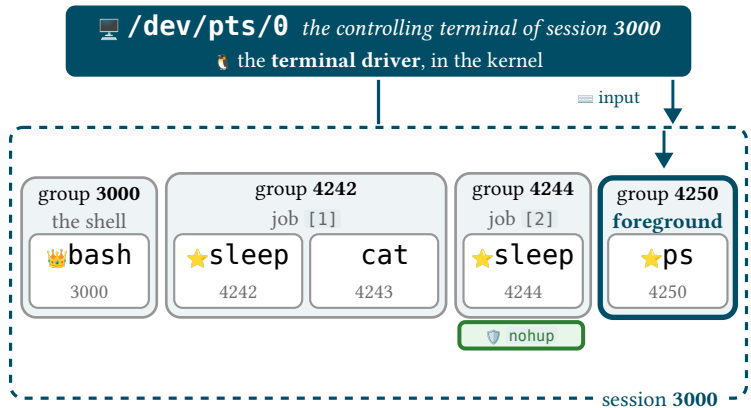
⌨️ only the **foreground** reads the input, `+` in `STAT`; a background job that reads gets `SIGTTIN` (stopped)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Sessions and the Terminal

How the kernel groups the processes of a terminal

```
$ echo $$
3000
$ sleep 100 | cat &
[1] 4243
$ nohup sleep 200 > /dev/null &
[2] 4244
nohup: ignoring input
$ ps -o pid,pgid,sid,stat,comm
    PID    PGID     SID STAT COMMAND
   3000    3000    3000 Ss   bash
   4242    4242    3000 S    sleep
   4243    4242    3000 S    cat
   4244    4244    3000 S    sleep
   4250    4250    3000 R+   ps
```




**session leader**: its PID is the `SID` **group leader**: its PID is the `PGID`

the **group leader**: the first process of the group, its PID is the `PGID`

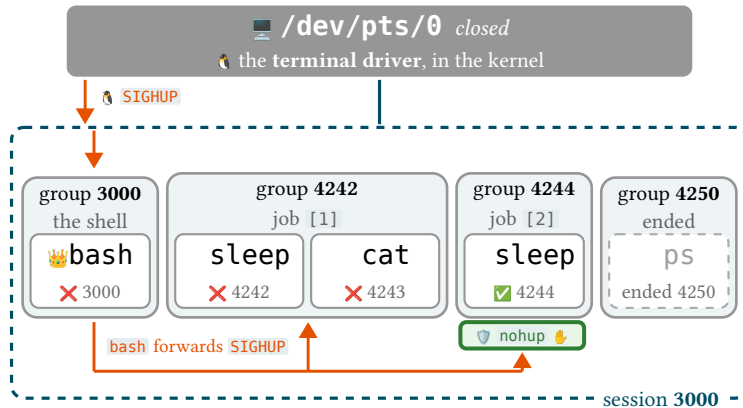
the **session leader**: the process that started the session, its PID is the `SID`, `s` in `STAT`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Sessions and the Terminal

How the kernel groups the processes of a terminal

```
$ echo $$
3000
$ sleep 100 | cat &
[1] 4243
$ nohup sleep 200 > /dev/null &
[2] 4244
nohup: ignoring input
$ ps -o pid,pgid,sid,stat,comm
    PID    PGID     SID STAT COMMAND
   3000    3000    3000 Ss   bash
   4242    4242    3000 S    sleep
   4243    4242    3000 S    cat
   4244    4244    3000 S    sleep
   4250    4250    3000 R+   ps
```




when the terminal closes, the **terminal driver** sends `SIGHUP` to the session leader, `bash`

`bash` sends it to its jobs: job `[1]` ends, job `[2]` ignores it and keeps running

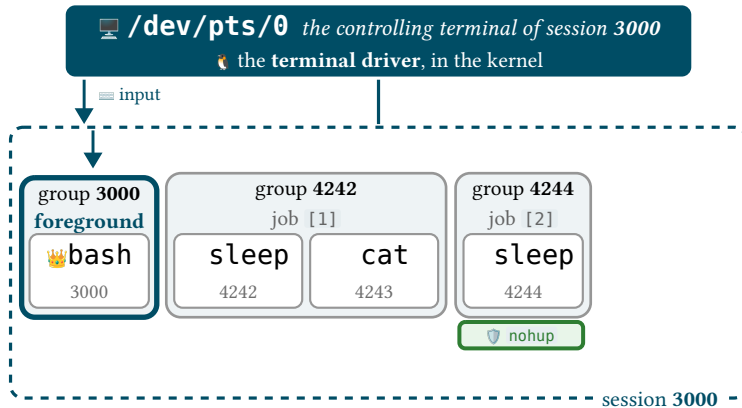
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# When `bash` Is Killed

`SIGKILL` cannot be caught: the shell cannot forward `SIGHUP` any more

🖥️ *terminal `/dev/pts/0` (`bash`, PID 3000)*

```
$ sleep 100 | cat &
[1] 4243
$ nohup sleep 200 > /dev/null &
[2] 4244
nohup: ignoring input
```




💡 two jobs in the session of `bash`, as before

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# When `bash` Is Killed

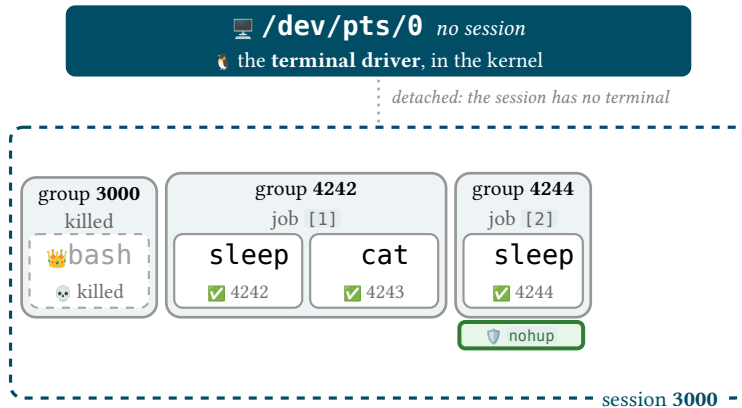
`SIGKILL` cannot be caught: the shell cannot forward `SIGHUP` any more

🖥️ *terminal `/dev/pts/0` (`bash`, PID 3000)*

```
$ sleep 100 | cat &
[1] 4243
$ nohup sleep 200 > /dev/null &
[2] 4244
nohup: ignoring input
```

🖥️ *another terminal, `/dev/pts/1`*

```
$ kill -KILL 3000
```




💀 `SIGKILL` cannot be caught: `bash` ends at once, it cannot forward anything

🐧 the session leader ended: the kernel detaches the terminal from the session

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# When `bash` Is Killed

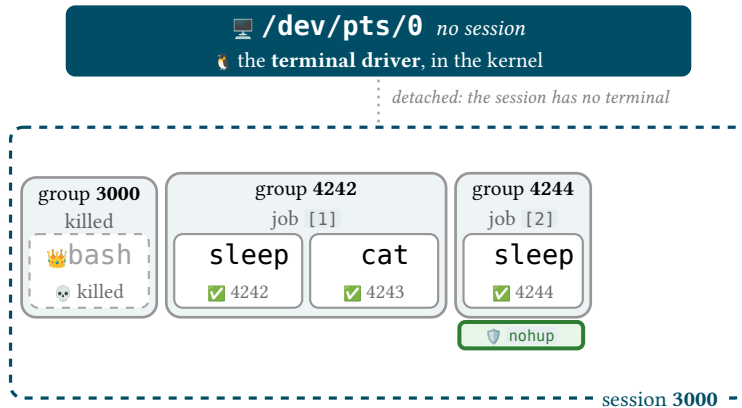
`SIGKILL` cannot be caught: the shell cannot forward `SIGHUP` any more

🖥️ *terminal `/dev/pts/0` (`bash`, PID 3000)*

```
$ sleep 100 | cat &
[1] 4243
$ nohup sleep 200 > /dev/null &
[2] 4244
nohup: ignoring input
```

🖥️ *another terminal, `/dev/pts/1`*

```
$ kill -KILL 3000
$ ps -o pid,ppid,sid,tty,comm -s 3000
    PID    PPID     SID TT       COMMAND
   4242       1    3000 ?        sleep
   4243       1    3000 ?        cat
   4244       1    3000 ?        sleep
```




💡 the jobs keep running: no controlling terminal (`TT` is `?`), their new parent is PID 1

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# When `bash` Is Killed

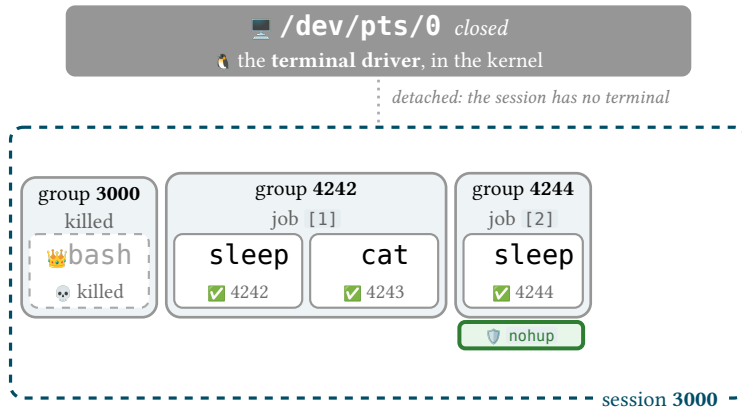
`SIGKILL` cannot be caught: the shell cannot forward `SIGHUP` any more

🖥️ *terminal `/dev/pts/0` (`bash`, PID 3000), closed*

```
$ sleep 100 | cat &
[1] 4243
$ nohup sleep 200 > /dev/null &
[2] 4244
nohup: ignoring input
```

🖥️ *another terminal, `/dev/pts/1`*

```
$ kill -KILL 3000
$ ps -o pid,ppid,sid,tty,comm -s 3000
    PID    PPID     SID TT       COMMAND
   4242       1    3000 ?        sleep
   4243       1    3000 ?        cat
   4244       1    3000 ?        sleep
```




🔌 the terminal closes: no `SIGHUP` reaches the jobs, they keep running in the background

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Keys and Signals

The terminal turns some keys into signals for the **foreground** process

| Keys | Sends | Default action |
|---|---|---|
| `Ctrl` + `C` | `SIGINT` | *terminate* (interrupt) the process |
| `Ctrl` + `Z` | `SIGTSTP` | *stop* the process, it becomes a stopped job |
| `Ctrl` + `\` | `SIGQUIT` | *terminate* the process and write a core dump |
| `Ctrl` + `D` | no signal | *end of file*: `stdin` has nothing more to read |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Keys and Signals

The terminal turns some keys into signals for the **foreground** process

| Keys | Sends | Default action |
|---|---|---|
| `Ctrl` + `C` | `SIGINT` | *terminate* (interrupt) the process |
| `Ctrl` + `Z` | `SIGTSTP` | *stop* the process, it becomes a stopped job |
| `Ctrl` + `\` | `SIGQUIT` | *terminate* the process and write a core dump |
| `Ctrl` + `D` | no signal | *end of file*: `stdin` has nothing more to read |

💡 a **job** that tries to read the keyboard receives `SIGTTIN` and is *stopped*, only the foreground process can read it

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Keys and Signals

The terminal turns some keys into signals for the **foreground** process

| Keys | Sends | Default action |
|---|---|---|
| `Ctrl` + `C` | `SIGINT` | *terminate* (interrupt) the process |
| `Ctrl` + `Z` | `SIGTSTP` | *stop* the process, it becomes a stopped job |
| `Ctrl` + `\` | `SIGQUIT` | *terminate* the process and write a core dump |
| `Ctrl` + `D` | no signal | *end of file*: `stdin` has nothing more to read |

💡 a **job** that tries to read the keyboard receives `SIGTTIN` and is *stopped*, only the foreground process can read it

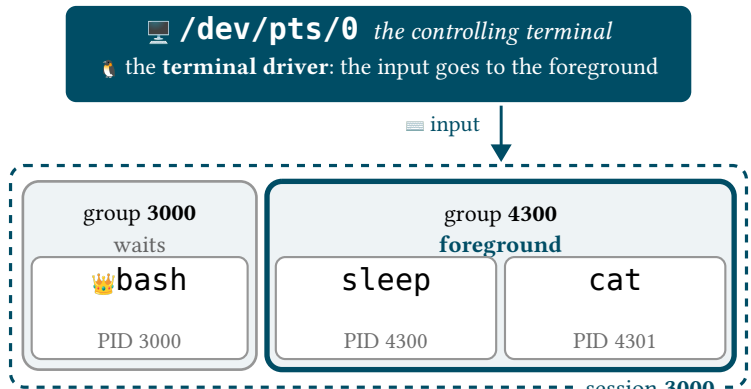
💡 `bg` and `fg` send `SIGCONT` to continue a stopped job

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Ctrl+C, Ctrl+\ and Ctrl+Z

The terminal driver turns three keys into signals for the **foreground** group

```
$ sleep 100 | cat
```



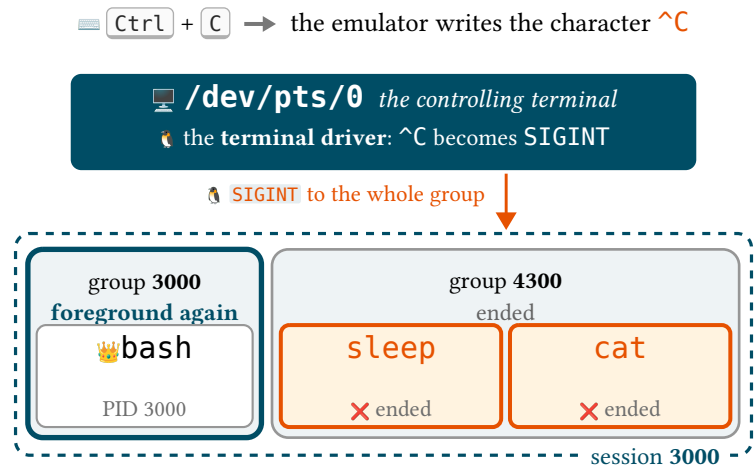

⌨️ `sleep 100 | cat` is the **foreground** group: it gets the input, `bash` waits for it

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Ctrl+C, Ctrl+\ and Ctrl+Z

The terminal driver turns three keys into signals for the **foreground** group

```
$ sleep 100 | cat
^C
```




the terminal driver sends `SIGINT` to **every** process of the foreground group

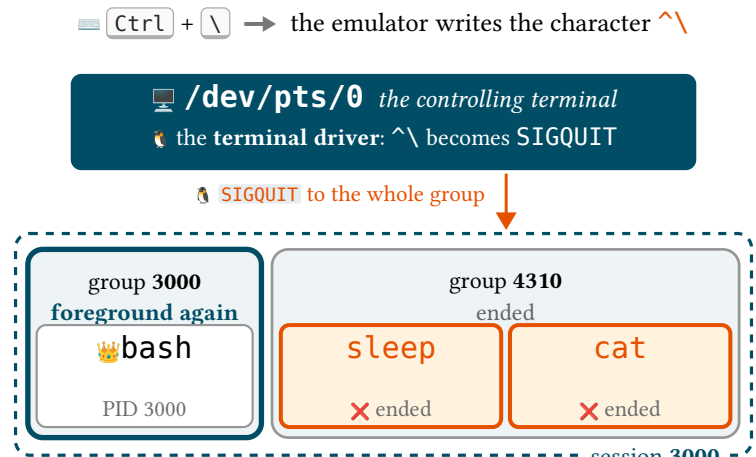
`sleep` and `cat` end, `bash` gets the terminal back and shows the prompt

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Ctrl+C, Ctrl+\ and Ctrl+Z

The terminal driver turns three keys into signals for the **foreground** group

```
$ sleep 100 | cat
^C
$ sleep 200 | cat
^\Quit
```




🐧 a new job, `Ctrl` + `\` becomes `SIGQUIT`: the whole group ends, like `Ctrl` + `C`

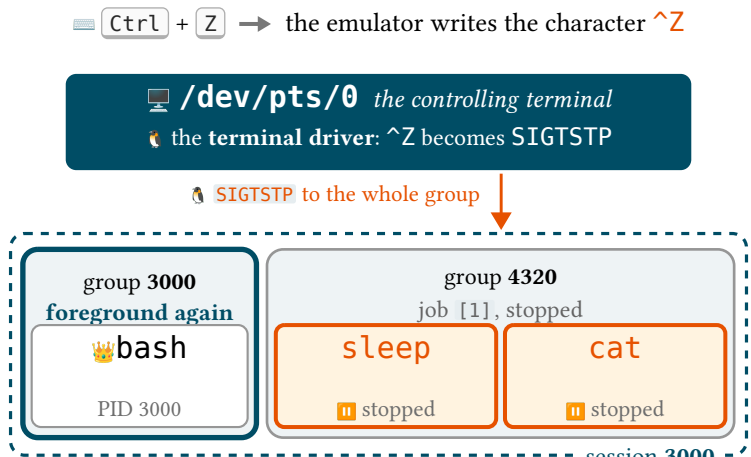
💡 it also writes a **core dump** (a copy of the memory, for debugging), if `ulimit -c` allows it

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Ctrl+C, Ctrl+\ and Ctrl+Z

The terminal driver turns three keys into signals for the **foreground** group

```
$ sleep 100 | cat
^C
$ sleep 200 | cat
^\Quit
$ sleep 300 | cat
^Z
[1]+  Stopped                 sleep 300 | cat
```




a new job, Ctrl + Z becomes `SIGTSTP`: the whole group is **stopped**, not ended

`bash` gets the terminal back, the group becomes job `[1]`, `fg` continues it

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Environment Variables

*Settings that every process inherits*

# Bibliography

For this section


1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Section 2.8 - *Environment and Shell Variables*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 7 - *Interfața în linia de comandă*
     - Section 7.2.3 - *Variabile shell*


 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Environment Variables

`NAME=value` texts that are part of every process

- every process has a list of `NAME=value` texts, its *environment*

| variables of `bash` | |
|---|---|
| HOME | /home/alice |
| USER | alice |
| SHELL | /bin/bash |
| PATH | /usr/local/bin:/usr/bin:/bin |
| LANG | en_US.UTF-8 |
| PWD | /home/alice/Movies |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Environment Variables

`NAME=value` texts that are part of every process

- every process has a list of `NAME=value` texts, its *environment*
- a process **receives** it from its parent: `fork` copies it, `exec` keeps it

| variables of `bash` | |
|---|---|
| HOME | /home/alice |
| USER | alice |
| SHELL | /bin/bash |
| PATH | /usr/local/bin:/usr/bin:/bin |
| LANG | en_US.UTF-8 |
| PWD | /home/alice/Movies |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Environment Variables

`NAME=value` texts that are part of every process

- every process has a list of `NAME=value` texts, its *environment*
- a process **receives** it from its parent: `fork` copies it, `exec` keeps it
- programs read it to configure themselves: `getenv("HOME")` in C, `$HOME` in `bash`

| variables of `bash` | |
|---|---|
| HOME | /home/alice |
| USER | alice |
| SHELL | /bin/bash |
| PATH | /usr/local/bin:/usr/bin:/bin |
| LANG | en_US.UTF-8 |
| PWD | /home/alice/Movies |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Environment Variables

`NAME=value` texts that are part of every process

- every process has a list of `NAME=value` texts, its *environment*
- a process **receives** it from its parent: `fork` copies it, `exec` keeps it
- programs read it to configure themselves: `getenv("HOME")` in C, `$HOME` in `bash`
- the names are written in **UPPER CASE**, by convention

| variables of `bash` | |
|---|---|
| HOME | /home/alice |
| USER | alice |
| SHELL | /bin/bash |
| PATH | /usr/local/bin:/usr/bin:/bin |
| LANG | en_US.UTF-8 |
| PWD | /home/alice/Movies |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Environment Variables

`NAME=value` texts that are part of every process

- every process has a list of `NAME=value` texts, its *environment*
- a process **receives** it from its parent: `fork` copies it, `exec` keeps it
- programs read it to configure themselves: `getenv("HOME")` in C, `$HOME` in `bash`
- the names are written in **UPPER CASE**, by convention

| variables of `bash` | |
|---|---|
| HOME | /home/alice |
| USER | alice |
| SHELL | /bin/bash |
| PATH | /usr/local/bin:/usr/bin:/bin |
| LANG | en_US.UTF-8 |
| PWD | /home/alice/Movies |

```
$ printenv HOME    # one variable
/home/alice
$ echo "I am $USER" # the shell expands $USER
I am alice
$ env              # all the variables
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Inherited by the Children

Every process starts with a **copy** of the environment of its parent

| variables of `bash` | |
|---|---|
| HOME | /home/alice |
| USER | alice |
| SHELL | /bin/bash |
| PATH | /usr/local/bin:/usr/bin:/bin |

| variables of `sort` |
|---|
| |
| |
| |
| |

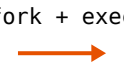
💡 `bash` has its own environment, like every process

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Inherited by the Children

Every process starts with a **copy** of the environment of its parent

| variables of `bash` | |
|---|---|
| HOME | /home/alice |
| USER | alice |
| SHELL | /bin/bash |
| PATH | /usr/local/bin:/usr/bin:/bin |




| variables of `sort` | |
|---|---|
| HOME | /home/alice |
| USER | alice |
| SHELL | /bin/bash |
| PATH | /usr/local/bin:/usr/bin:/bin |

💡 the shell starts `sort`: the child gets a **copy** of all the environment variables

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Inherited by the Children

Every process starts with a **copy** of the environment of its parent

| variables of `bash` | |
|---|---|
| HOME | /home/alice |
| USER | alice |
| SHELL | /bin/bash |
| PATH | /usr/local/bin:/usr/bin:/bin |

fork + exec →

| variables of `sort` | |
|---|---|
| HOME | /home/alice |
| USER | alice |
| SHELL | /bin/bash |
| **PATH** | **/tmp** |

The child process can change **its own copy** using `setenv("PATH", "/tmp")` or `PATH=/tmp`, the parent does not see the change

💡 a process can only pass variables **down** to its children, never **up** to its parent

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# export

Shell variables and environment variables: only the **exported** ones are passed to the children

```
$ GREETING=hello    # a shell variable
```




*your shell*

| variables of **bash** | | |
|---|---|---|
| HOME | /home/alice | |
| *GREETING* | *hello* | *(shell only)* |

💡 `NAME=value` makes a **shell variable**: it exists only inside this shell

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# export

Shell variables and environment variables: only the **exported** ones are passed to the children

```
$ GREETING=hello    # a shell variable
$ echo $GREETING
hello
```




*your shell*

| variables of **bash** | | |
|---|---|---|
| HOME | /home/alice | |
| *GREETING* | *hello* | *(shell only)* |

💡 `$NAME` is replaced by its value, by the shell, before the command starts

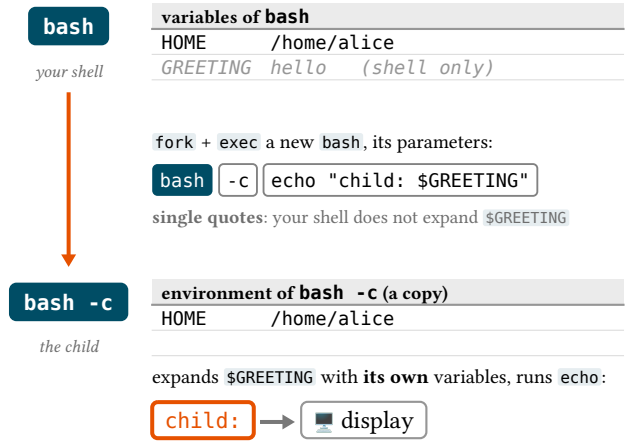
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# export

Shell variables and environment variables: only the **exported** ones are passed to the children

```
$ GREETING=hello    # a shell variable
$ echo $GREETING
hello

# run a new bash instance that runs
#  echo "child: $GREETING" - mind the '...'
$ bash -c 'echo "child: $GREETING"'
child:
```




💡 the child (`bash -c`) does **not** receive it, it is not part of the environment

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# `export`

Shell variables and environment variables: only the **exported** ones are passed to the children

```
$ GREETING=hello    # a shell variable
$ echo $GREETING
hello

# run a new bash instance that runs
#  echo "child: $GREETING" - mind the '...'
$ bash -c 'echo "child: $GREETING"'
child:
$ export GREETING   # an environment variable
```

**bash**

*your shell*

| variables of **bash** | |
|---|---|
| HOME | /home/alice |
| **GREETING** | **hello** |

💡 `export` marks it as an **environment variable**

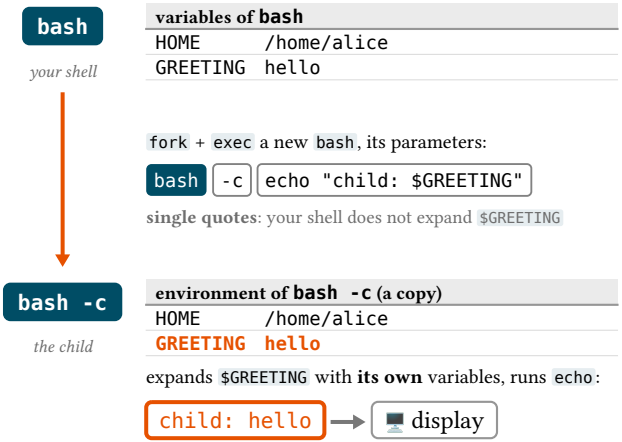
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# export

Shell variables and environment variables: only the **exported** ones are passed to the children

```
$ GREETING=hello    # a shell variable
$ echo $GREETING
hello

# run a new bash instance that runs
#  echo "child: $GREETING" - mind the '...'
$ bash -c 'echo "child: $GREETING"'
child:
$ export GREETING   # an environment variable
$ bash -c 'echo "child: $GREETING"'
child: hello
```




- every child started from now on receives a copy
- `export NAME=value` does both at once, `unset NAME` deletes it

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Defined by the Shell

Variables that are already there when the prompt shows up

| Variable | Value | Defined in (Fedora) |
|---|---|---|
| `HOME`, `USER`, `SHELL` | the home directory, the user, the shell | `/etc/passwd`, read by `login` when you log in |
| `PWD`, `OLDPWD` | the current and the previous directory | the shell itself, at every `cd` (used by `cd -`) |
| `PATH` | where commands are | `/etc/profile`, then `~/.bashrc` adds `~/.local/bin` and `~/bin` |
| `LANG` | the language | `/etc/locale.conf`, read by `/etc/profile.d/lang.sh` |
| `PS1` | what the prompt looks like, a **shell variable** | `/etc/bashrc`, read again by every interactive shell |

💡 the startup files are shell scripts: they run `export NAME=value` (or `NAME=value` for `PS1`)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Special Parameters

Set by the shell, they can only be read

- `$?` - the exit code of the last command (`0` means *success*)
- `$$` - the PID of the shell
- `$!` - the PID of the last job started with `&`

```
$ ls /nope
ls: cannot access '/nope': No such file or directory
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Special Parameters

Set by the shell, they can only be read

- `$?` - the exit code of the last command (`0` means *success*)
- `$$` - the PID of the shell
- `$!` - the PID of the last job started with `&`

```
$ ls /nope
ls: cannot access '/nope': No such file or directory
$ echo $?
2
```

💡 `ls` failed: the exit code is not `0`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Special Parameters

Set by the shell, they can only be read

- `$?` - the exit code of the last command (`0` means *success*)
- `$$` - the PID of the shell
- `$!` - the PID of the last job started with `&`

```
$ ls /nope
ls: cannot access '/nope': No such file or directory
$ echo $?
2
$ echo $?    # exit code of echo
0
```

💡 `$?` changes after **every** command

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Special Parameters

Set by the shell, they can only be read

- `$?` - the exit code of the last command (`0` means *success*)
- `$$` - the PID of the shell
- `$!` - the PID of the last job started with `&`

```
$ ls /nope
ls: cannot access '/nope': No such file or directory
$ echo $?
2
$ echo $?    # exit code of echo
0
$ echo $$
4011
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Defined in the Command Line

| Syntax | Defines |
|---|---|
| `NAME=value` | a **shell variable**, only for this shell |
| `export NAME=value` | an **environment variable**, for this shell and every command started after it |
| `NAME=value command` | an environment variable **only for** `command`, the shell does not keep it |
| `unset NAME` | deletes the variable |

```
$ date +%H:%M
11:47
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Defined in the Command Line

| Syntax | Defines |
|---|---|
| `NAME=value` | a **shell variable**, only for this shell |
| `export NAME=value` | an **environment variable**, for this shell and every command started after it |
| `NAME=value command` | an environment variable **only for** `command`, the shell does not keep it |
| `unset NAME` | deletes the variable |

```
$ date +%H:%M
11:47
$ TZ=Asia/Tokyo date +%H:%M   # only for this date
17:47
```

💡 `TZ` (time zone) is set only in the environment of this `date` process

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Defined in the Command Line

| Syntax | Defines |
|---|---|
| `NAME=value` | a **shell variable**, only for this shell |
| `export NAME=value` | an **environment variable**, for this shell and every command started after it |
| `NAME=value command` | an environment variable **only for** `command`, the shell does not keep it |
| `unset NAME` | deletes the variable |

```
$ date +%H:%M
11:47
$ TZ=Asia/Tokyo date +%H:%M   # only for this date
17:47
$ echo "[$TZ]"                # the shell does not have it
[]
```

💡 useful to change a setting just once, e.g.
`LANG=C sort names.txt`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Defined in the Command Line

| Syntax | Defines |
|---|---|
| `NAME=value` | a **shell variable**, only for this shell |
| `export NAME=value` | an **environment variable**, for this shell and every command started after it |
| `NAME=value command` | an environment variable **only for** `command`, the shell does not keep it |
| `unset NAME` | deletes the variable |

```
$ date +%H:%M
11:47
$ TZ=Asia/Tokyo date +%H:%M   # only for this date
17:47
$ echo "[$TZ]"                # the shell does not have it
[]
$ GREETING = hello
bash: GREETING: command not found
```

⚠️ **no spaces** around `=`: the shell thinks `GREETING` is a command

💡 use quotes for spaces in the value: `NAME="two words"`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The Command Line

*How the shell reads what you type*

# Bibliography

For this section


1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Section 2.10 - *Special Characters*
     - Section 2.14 - *Shell Input and Output*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 7 - *Interfața în linia de comandă*
     - Section 7.2.3 - *Variabile shell*
     - Section 7.2.4 - *Expandări*


 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# A Command Line

Assignments, the command, its parameters and the redirects

```
$ LANG=C sort -r "$HOME/my notes.txt" my\ list.txt *.csv > sorted.txt 2> errors.txt
```

`LANG=C`
assignments
(optional)

💡 `NAME=value` words **at the start**: variables only for this command

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# A Command Line

Assignments, the command, its parameters and the redirects

```
$ LANG=C sort -r "$HOME/my notes.txt" my\ list.txt *.csv > sorted.txt 2> errors.txt
```

`LANG=C` — assignments (optional)

`sort` — the command

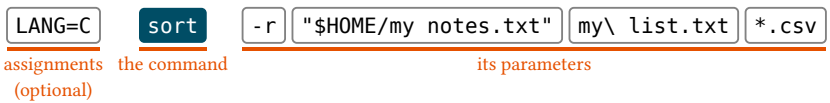
💡 the first word that is not an assignment is the **command**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# A Command Line

Assignments, the command, its parameters and the redirects

```
$ LANG=C sort -r "$HOME/my notes.txt" my\ list.txt *.csv > sorted.txt 2> errors.txt
```



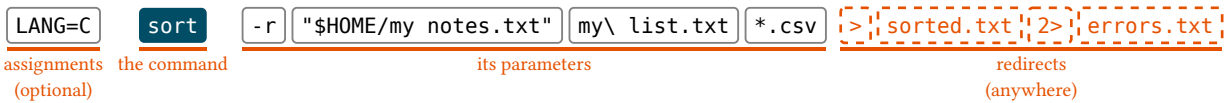

💡 everything else is a **parameter**: options, file names, texts

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# A Command Line

Assignments, the command, its parameters and the redirects

```
$ LANG=C sort -r "$HOME/my notes.txt" my\ list.txt *.csv > sorted.txt 2> errors.txt
```




💡 a redirect (`>`, `2>`, `<`, ...) and its file name are for the shell, not for `sort`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Splitting a Line

What the shell does, step by step, before the command starts

```
$ LANG=C sort -r "$HOME/my notes.txt" my\ list.txt *.csv > sorted.txt 2> errors.txt
```

*1. split into words* `LANG=C` `sort` `-r` `"$HOME/my notes.txt"` `my\ list.txt` `*.csv` `>` `sorted.txt` `2>` `errors.txt`

💡 split at the spaces, except inside quotes and after a `\`; `>` and `2>` are words of their own: 10 words

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Splitting a Line

What the shell does, step by step, before the command starts

```
$ LANG=C sort -r "$HOME/my notes.txt" my\ list.txt *.csv > sorted.txt 2> errors.txt
```

*1. split into words*

`LANG=C` | `sort` | `-r` | `"$HOME/my notes.txt"` | `my\ list.txt` | `*.csv` | `>` | `sorted.txt` | `2>` | `errors.txt`

*2. set apart*

`sort` | `-r` | `"$HOME/my notes.txt"` | `my\ list.txt` | `*.csv`

`LANG=C`

`> sorted.txt`

`2> errors.txt`

💡 the assignment (no spaces around `=`) and the redirects (`>` or `2>` + the next word) are set apart

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Splitting a Line

What the shell does, step by step, before the command starts

```
$ LANG=C sort -r "$HOME/my notes.txt" my\ list.txt *.csv > sorted.txt 2> errors.txt
```

*1. split into words*: `LANG=C` | `sort` | `-r` | `"$HOME/my notes.txt"` | `my\ list.txt` | `*.csv` | `>` | `sorted.txt` | `2>` | `errors.txt`

*2. set apart*: `sort` | `-r` | `"$HOME/my notes.txt"` | `my\ list.txt` | `*.csv`

`LANG=C`

`> sorted.txt`

`2> errors.txt`

*3. replace `$NAME`, `~`*: `sort` | `-r` | `"/home/alice/my notes.txt"` | `my\ list.txt` | `*.csv`

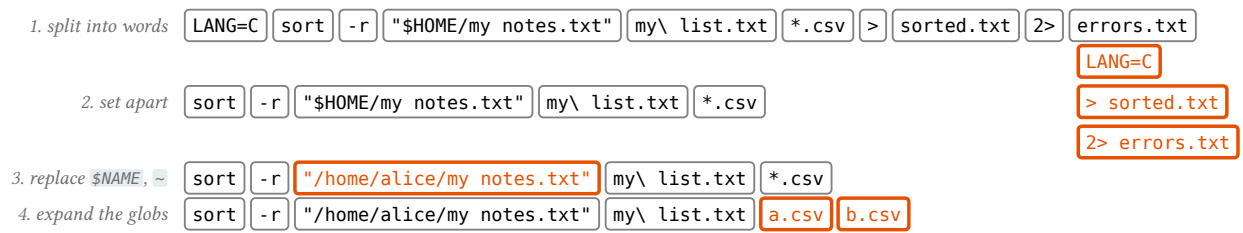
💡 `$HOME` is replaced inside `"..."` (not inside `'...'`), the result stays **one** word

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Splitting a Line

What the shell does, step by step, before the command starts

```
$ LANG=C sort -r "$HOME/my notes.txt" my\ list.txt *.csv > sorted.txt 2> errors.txt
```



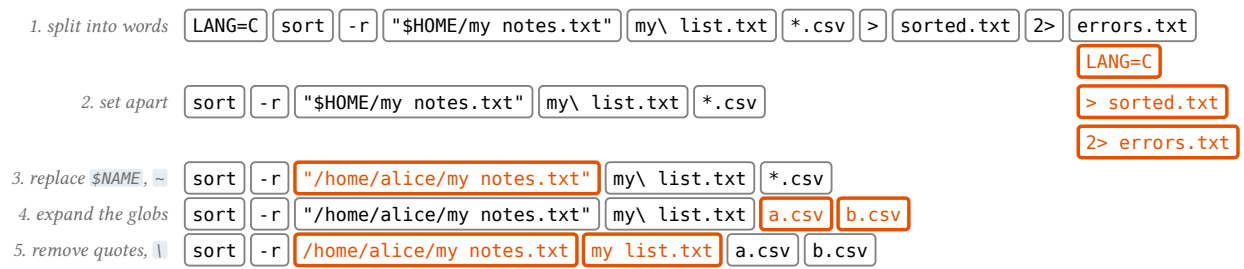

💡 `*.csv` is not quoted: the shell replaces it with the matching file names

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Splitting a Line

What the shell does, step by step, before the command starts

```
$ LANG=C sort -r "$HOME/my notes.txt" my\ list.txt *.csv > sorted.txt 2> errors.txt
```



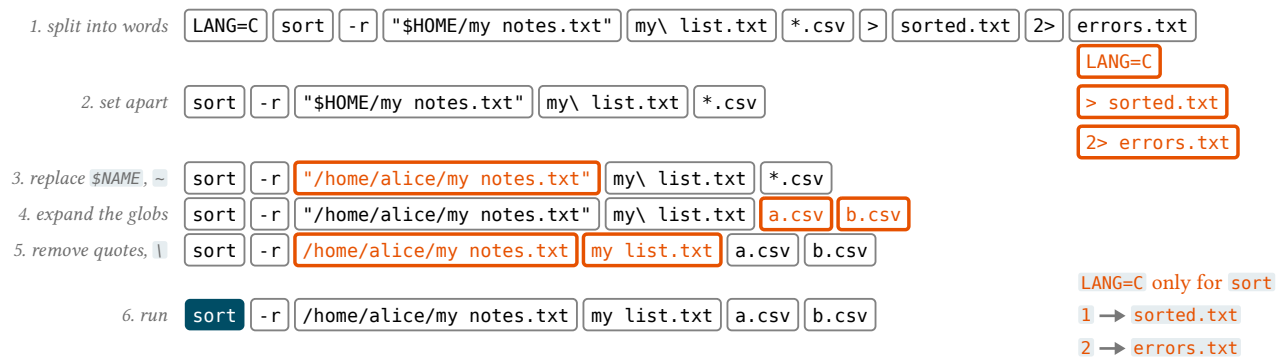

💡 the quotes and the `\` did their job: the shell removes them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Splitting a Line

What the shell does, step by step, before the command starts

```
$ LANG=C sort -r "$HOME/my notes.txt" my\ list.txt *.csv > sorted.txt 2> errors.txt
```




💡 the shell sets up the redirects (it creates `sorted.txt`), then starts `sort` with 5 parameters and `LANG=C`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Mixed Order

The parts can be mixed: only the assignments must be at the start

```
$ IN="my notes.txt" OUT=sorted.txt OPTS="-u -f"
$ LANG=C 2> errors.txt sort "$IN" '$IN' > "$OUT" -r $OPTS -t ',' *.csv
```




💡 `LANG=C` is at the start: an assignment, only for this command

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Mixed Order

The parts can be mixed: only the assignments must be at the start

```
$ IN="my notes.txt" OUT=sorted.txt OPTS="-u -f"
$ LANG=C 2> errors.txt sort "$IN" '$IN' > "$OUT" -r $OPTS -t ',' *.csv
```

`LANG=C` — assignment

`sort` — command

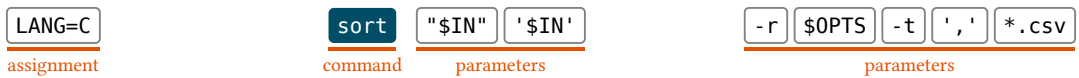
💡 the first word that is not an assignment or a redirect is the **command**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Mixed Order

The parts can be mixed: only the assignments must be at the start

```
$ IN="my notes.txt" OUT=sorted.txt OPTS="-u -f"
$ LANG=C 2> errors.txt sort "$IN" '$IN' > "$OUT" -r $OPTS -t ',' *.csv
```



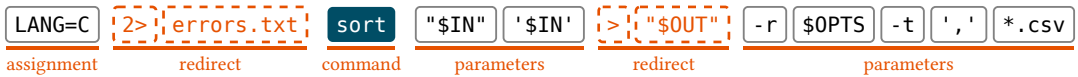

💡 the parameters do not have to be together, a redirect can be between them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Mixed Order

The parts can be mixed: only the assignments must be at the start

```
$ IN="my notes.txt" OUT=sorted.txt OPTS="-u -f"
$ LANG=C 2> errors.txt sort "$IN" '$IN' > "$OUT" -r $OPTS -t ',' *.csv
```




💡 the redirects can be anywhere: even before the command, or between the parameters

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Another Line

The assignment at the start, the redirects anywhere, double and single quotes, a glob

```
$ IN="my notes.txt" OUT=sorted.txt OPTS="-u -f"
$ LANG=C 2> errors.txt sort "$IN" '$IN' > "$OUT" -r $OPTS -t ',' *.csv
```

*1. split into words* `LANG=C` `2>` `errors.txt` `sort` `"$IN"` `'$IN'` `>` `"$OUT"` `-r` `$OPTS` `-t` `','` `*.csv`

💡 the first line sets `IN`, `OUT` and `OPTS`; the second one has 13 words, a redirect is even before the command

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Another Line

The assignment at the start, the redirects anywhere, double and single quotes, a glob

```
$ IN="my notes.txt" OUT=sorted.txt OPTS="-u -f"
$ LANG=C 2> errors.txt sort "$IN" '$IN' > "$OUT" -r $OPTS -t ',' *.csv
```

*1. split into words*

`LANG=C` `2>` `errors.txt` `sort` `"$IN"` `'$IN'` `>` `"$OUT"` `-r` `$OPTS` `-t` `','` `*.csv`

*2. set apart*

`sort` `"$IN"` `'$IN'` `-r` `$OPTS` `-t` `','` `*.csv`

`LANG=C`

`2> errors.txt`

`> "$OUT"`

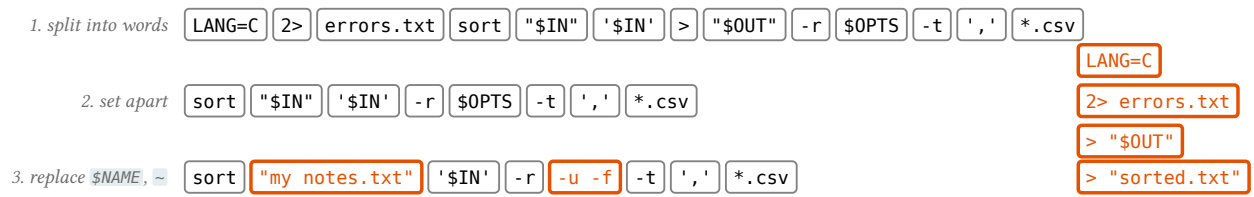
💡 the assignment (at the start) and the 2 redirects are set apart; `> "$OUT"` was in the **middle**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Another Line

The assignment at the start, the redirects anywhere, double and single quotes, a glob

```
$ IN="my notes.txt" OUT=sorted.txt OPTS="-u -f"
$ LANG=C 2> errors.txt sort "$IN" '$IN' > "$OUT" -r $OPTS -t ',' *.csv
```



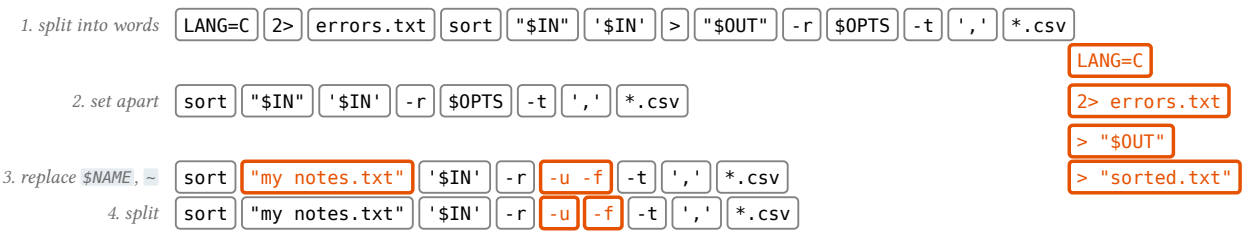

💡 `$IN` is replaced inside `"..."`, but **not** inside `'...'`; `$OPTS` and `$OUT` are replaced too

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Another Line

The assignment at the start, the redirects anywhere, double and single quotes, a glob

```
$ IN="my notes.txt" OUT=sorted.txt OPTS="-u -f"
$ LANG=C 2> errors.txt sort "$IN" '$IN' > "$OUT" -r $OPTS -t ',' *.csv
```



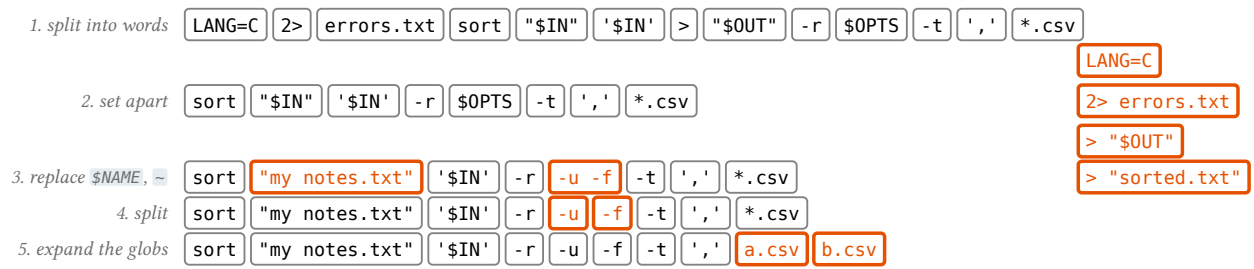

💡 `$OPTS` was not quoted: it becomes 2 words; `"$IN"` was quoted: `my notes.txt` stays one word

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Another Line

The assignment at the start, the redirects anywhere, double and single quotes, a glob

```
$ IN="my notes.txt" OUT=sorted.txt OPTS="-u -f"
$ LANG=C 2> errors.txt sort "$IN" '$IN' > "$OUT" -r $OPTS -t ',' *.csv
```



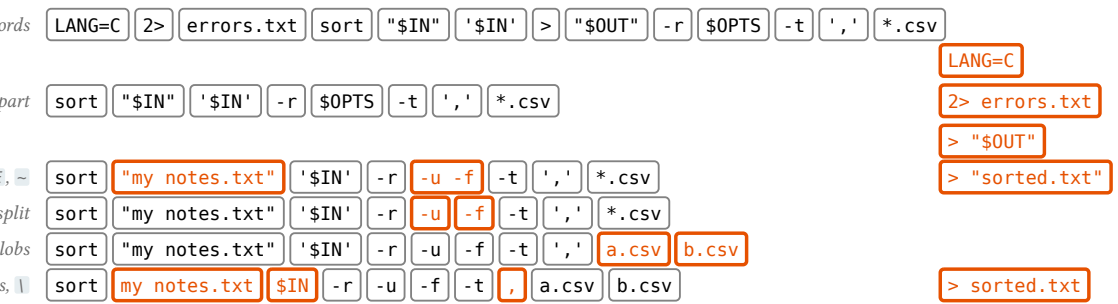

💡 `*.csv` is not quoted: the files of the current directory, `a.csv` and `b.csv`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Another Line

The assignment at the start, the redirects anywhere, double and single quotes, a glob

```
$ IN="my notes.txt" OUT=sorted.txt OPTS="-u -f"
$ LANG=C 2> errors.txt sort "$IN" '$IN' > "$OUT" -r $OPTS -t ',' *.csv
```



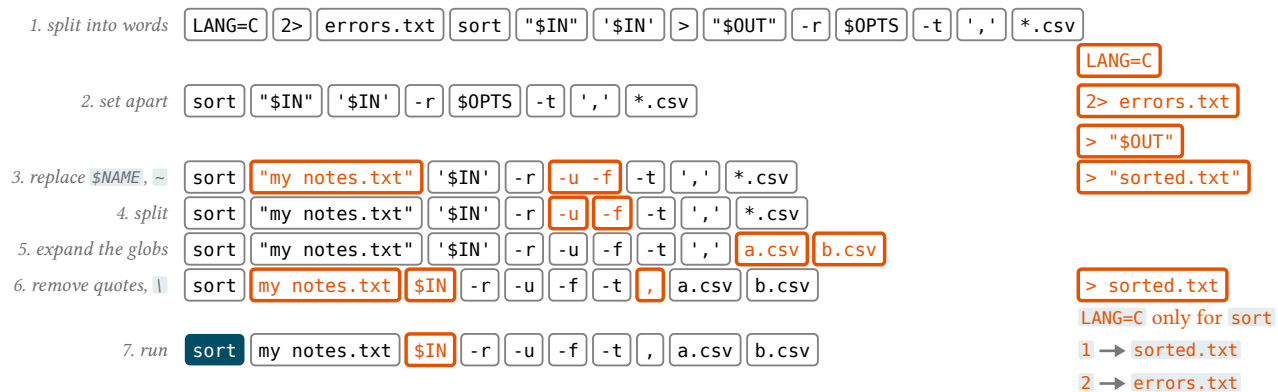

💡 the quotes did their job: the shell removes them, `'$IN'` becomes the text `$IN`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Another Line

The assignment at the start, the redirects anywhere, double and single quotes, a glob

```
$ IN="my notes.txt" OUT=sorted.txt OPTS="-u -f"
$ LANG=C 2> errors.txt sort "$IN" '$IN' > "$OUT" -r $OPTS -t ',' *.csv
```




💡 `sort` gets a file named `$IN`: it does not exist, the error goes to `errors.txt`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The Order of Redirects

The shell applies them **from left to right**

```
$ ls /nope > all.txt 2>&1
```

`1` → 🖥️ display
`2` → 🖥️ display

💡 `2>&1` means: `2` goes where `1` goes **now**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The Order of Redirects

The shell applies them **from left to right**

```
$ ls /nope > all.txt 2>&1
```




💡 `2>&1` means: `2` goes where `1` goes **now**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The Order of Redirects

The shell applies them **from left to right**

```
$ ls /nope > all.txt 2>&1
```




nothing on the display, the error is in `all.txt`

💡 `2>&1` means: `2` goes where `1` goes **now**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The Order of Redirects

The shell applies them **from left to right**

```
$ ls /nope > all.txt 2>&1
```




nothing on the display, the error is in `all.txt`

```
$ ls /nope 2>&1 > all.txt
```




💡 `2>&1` is a **copy**, not a link: changing `1` later does not change `2`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The Order of Redirects

The shell applies them **from left to right**

```
$ ls /nope > all.txt 2>&1
```




nothing on the display, the error is in `all.txt`

```
$ ls /nope 2>&1 > all.txt
```




💡 `2>&1` is a **copy**, not a link: changing `1` later does not change `2`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# The Order of Redirects

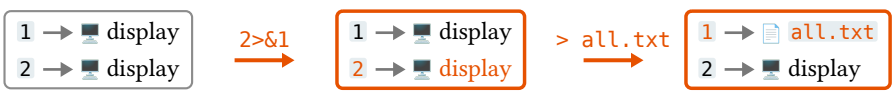
The shell applies them **from left to right**

```
$ ls /nope > all.txt 2>&1
```




nothing on the display, the error is in `all.txt`

```
$ ls /nope 2>&1 > all.txt
```




the error is on the **display**

💡 `2>&1` is a **copy**, not a link: changing `1` later does not change `2`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Quoting, Escaping and Globbing

*Special characters and file name patterns*

# Bibliography

For this section


1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Section 2.4.2 - *Shell Globbing ("Wildcards")*
     - Section 2.10 - *Special Characters*
   - Chapter 11 - *Introduction to Shell Scripts*
     - Section 11.2 - *Quoting and Literals*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 7 - *Interfața în linia de comandă*
     - Sections 7.2.4 - 7.2.5 - *Expandări, Escaping*


 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Quoting and Escaping

With `NAME="Alice   Smith"`: the quotes and the `\` are gone, only the words are left

*you type* → *the command and its parameters*

```
$ echo $NAME
```

→ `echo` `Alice` `Smith`

no quotes: replaced, then split at the spaces

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Quoting and Escaping

With `NAME="Alice   Smith"`: the quotes and the `\` are gone, only the words are left

| *you type* | | *the command and its parameters* | |
|---|---|---|---|
| `$ echo $NAME` | → | `echo` `Alice` `Smith` | no quotes: replaced, then split at the spaces |
| `$ echo "$NAME"` | → | `echo` `Alice   Smith` | double quotes: replaced, **one** word, spaces kept |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Quoting and Escaping

With `NAME="Alice   Smith"`: the quotes and the `\` are gone, only the words are left

| *you type* | | *the command and its parameters* | |
|---|---|---|---|
| `$ echo $NAME` | → | `echo` `Alice` `Smith` | no quotes: replaced, then split at the spaces |
| `$ echo "$NAME"` | → | `echo` `Alice   Smith` | double quotes: replaced, **one** word, spaces kept |
| `$ echo '$NAME'` | → | `echo` `$NAME` | single quotes: **nothing** is replaced |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Quoting and Escaping

With `NAME="Alice   Smith"`: the quotes and the `\` are gone, only the words are left

| *you type* | | *the command and its parameters* | |
|---|---|---|---|
| `$ echo $NAME` | → | `echo` `Alice` `Smith` | no quotes: replaced, then split at the spaces |
| `$ echo "$NAME"` | → | `echo` `Alice   Smith` | double quotes: replaced, **one** word, spaces kept |
| `$ echo '$NAME'` | → | `echo` `$NAME` | single quotes: **nothing** is replaced |
| `$ echo \$NAME` | → | `echo` `$NAME` | `\$`: the `$` is just a character |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Quoting and Escaping

With `NAME="Alice   Smith"`: the quotes and the `\` are gone, only the words are left

| *you type* | | *the command and its parameters* | | |
|---|---|---|---|---|
| `$ echo $NAME` | → | `echo` | `Alice` `Smith` | no quotes: replaced, then split at the spaces |
| `$ echo "$NAME"` | → | `echo` | `Alice   Smith` | double quotes: replaced, **one** word, spaces kept |
| `$ echo '$NAME'` | → | `echo` | `$NAME` | single quotes: **nothing** is replaced |
| `$ echo \$NAME` | → | `echo` | `$NAME` | `\$`: the `$` is just a character |
| `$ echo my\ file.txt` | → | `echo` | `my file.txt` | `\ `: the space does not split |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Quoting and Escaping

With `NAME="Alice   Smith"`: the quotes and the `\` are gone, only the words are left

| *you type* | | *the command and its parameters* | |
|---|---|---|---|
| `$ echo $NAME` | → | `echo` `Alice` `Smith` | no quotes: replaced, then split at the spaces |
| `$ echo "$NAME"` | → | `echo` `Alice   Smith` | double quotes: replaced, **one** word, spaces kept |
| `$ echo '$NAME'` | → | `echo` `$NAME` | single quotes: **nothing** is replaced |
| `$ echo \$NAME` | → | `echo` `$NAME` | `\$`: the `$` is just a character |
| `$ echo my\ file.txt` | → | `echo` `my file.txt` | `\ `: the space does not split |
| `$ echo "\$NAME is $NAME"` | → | `echo` `$NAME is Alice   Smith` | inside `"..."`, `\$` is a `$`, `$NAME` is replaced |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Quoting and Escaping

With `NAME="Alice   Smith"`: the quotes and the `\` are gone, only the words are left

| *you type* | | *the command and its parameters* | |
|---|---|---|---|
| `$ echo $NAME` | → | `echo` `Alice` `Smith` | no quotes: replaced, then split at the spaces |
| `$ echo "$NAME"` | → | `echo` `Alice   Smith` | double quotes: replaced, **one** word, spaces kept |
| `$ echo '$NAME'` | → | `echo` `$NAME` | single quotes: **nothing** is replaced |
| `$ echo \$NAME` | → | `echo` `$NAME` | `\$`: the `$` is just a character |
| `$ echo my\ file.txt` | → | `echo` `my file.txt` | `\ `: the space does not split |
| `$ echo "\$NAME is $NAME"` | → | `echo` `$NAME is Alice   Smith` | inside `"..."`, `\$` is a `$`, `$NAME` is replaced |
| `$ echo "it's" 'say "hi"'` | → | `echo` `it's` `say "hi"` | each kind of quotes protects the other one |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Quoting and Escaping

With `NAME="Alice   Smith"`: the quotes and the `\` are gone, only the words are left

| *you type* | | *the command and its parameters* | |
|---|---|---|---|
| `$ echo $NAME` | → | `echo` `Alice` `Smith` | no quotes: replaced, then split at the spaces |
| `$ echo "$NAME"` | → | `echo` `Alice   Smith` | double quotes: replaced, **one** word, spaces kept |
| `$ echo '$NAME'` | → | `echo` `$NAME` | single quotes: **nothing** is replaced |
| `$ echo \$NAME` | → | `echo` `$NAME` | `\$`: the `$` is just a character |
| `$ echo my\ file.txt` | → | `echo` `my file.txt` | `\ `: the space does not split |
| `$ echo "\$NAME is $NAME"` | → | `echo` `$NAME is Alice   Smith` | inside `"..."`, `\$` is a `$`, `$NAME` is replaced |
| `$ echo "it's" 'say "hi"'` | → | `echo` `it's` `say "hi"` | each kind of quotes protects the other one |
| `$ echo 'a\b'` | → | `echo` `a\b` | inside `'...'`, `\` is just a character |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Globing

Patterns for file names, replaced by the shell with the matching files

```
$ ls
docs  notes.txt  photo1.png  photo2.png  photo3.png  photoA.png  report.md  todo.txt
```

*you type*

```
$ echo *.txt
```

→

*the command and its parameters*

```
echo notes.txt todo.txt
```

`*`: any characters, or none

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Globing

Patterns for file names, replaced by the shell with the matching files

```
$ ls
docs  notes.txt  photo1.png  photo2.png  photo3.png  photoA.png  report.md  todo.txt
```

*you type* → *the command and its parameters*

`$ echo *.txt` → `echo` `notes.txt` `todo.txt`

`$ echo photo?.png` → `echo` `photo1.png` `photo2.png` `photo3.png` `photoA.png`

`*`: any characters, or none

`?`: exactly one character

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Globing

Patterns for file names, replaced by the shell with the matching files

```
$ ls
docs  notes.txt  photo1.png  photo2.png  photo3.png  photoA.png  report.md  todo.txt
```

| *you type* | | *the command and its parameters* | |
|---|---|---|---|
| `$ echo *.txt` | → | `echo` `notes.txt` `todo.txt` | `*`: any characters, or none |
| `$ echo photo?.png` | → | `echo` `photo1.png` `photo2.png` `photo3.png` `photoA.png` | `?`: exactly one character |
| `$ echo photo[12].png` | → | `echo` `photo1.png` `photo2.png` | `[12]`: one of these characters |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Globing

Patterns for file names, replaced by the shell with the matching files

```
$ ls
docs  notes.txt  photo1.png  photo2.png  photo3.png  photoA.png  report.md  todo.txt
```

| *you type* | | *the command and its parameters* | |
|---|---|---|---|
| `$ echo *.txt` | → | `echo` `notes.txt` `todo.txt` | `*`: any characters, or none |
| `$ echo photo?.png` | → | `echo` `photo1.png` `photo2.png` `photo3.png` `photoA.png` | `?`: exactly one character |
| `$ echo photo[12].png` | → | `echo` `photo1.png` `photo2.png` | `[12]`: one of these characters |
| `$ echo photo[0-9].png` | → | `echo` `photo1.png` `photo2.png` `photo3.png` | `[0-9]`: one in the range |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Globing

Patterns for file names, replaced by the shell with the matching files

```
$ ls
docs  notes.txt  photo1.png  photo2.png  photo3.png  photoA.png  report.md  todo.txt
```

| *you type* | | *the command and its parameters* | |
|---|---|---|---|
| `$ echo *.txt` | → | `echo` `notes.txt` `todo.txt` | `*`: any characters, or none |
| `$ echo photo?.png` | → | `echo` `photo1.png` `photo2.png` `photo3.png` `photoA.png` | `?`: exactly one character |
| `$ echo photo[12].png` | → | `echo` `photo1.png` `photo2.png` | `[12]`: one of these characters |
| `$ echo photo[0-9].png` | → | `echo` `photo1.png` `photo2.png` `photo3.png` | `[0-9]`: one in the range |
| `$ echo photo[!0-9].png` | → | `echo` `photoA.png` | `[!0-9]`: one **not** in it |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Globing

Patterns for file names, replaced by the shell with the matching files

```
$ ls
docs  notes.txt  photo1.png  photo2.png  photo3.png  photoA.png  report.md  todo.txt
```

| *you type* | *the command and its parameters* | |
|---|---|---|
| `$ echo *.txt` | `echo` `notes.txt` `todo.txt` | `*`: any characters, or none |
| `$ echo photo?.png` | `echo` `photo1.png` `photo2.png` `photo3.png` `photoA.png` | `?`: exactly one character |
| `$ echo photo[12].png` | `echo` `photo1.png` `photo2.png` | `[12]`: one of these characters |
| `$ echo photo[0-9].png` | `echo` `photo1.png` `photo2.png` `photo3.png` | `[0-9]`: one in the range |
| `$ echo photo[!0-9].png` | `echo` `photoA.png` | `[!0-9]`: one **not** in it |
| `$ echo docs/**/*.txt` | `echo` `docs/2026/exam.txt` `docs/plan.txt` | `**`: any directories (`globstar`) |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Globing

Patterns for file names, replaced by the shell with the matching files

```
$ ls
docs  notes.txt  photo1.png  photo2.png  photo3.png  photoA.png  report.md  todo.txt
```

| *you type* | *the command and its parameters* | |
|---|---|---|
| `$ echo *.txt` | `echo` `notes.txt` `todo.txt` | `*`: any characters, or none |
| `$ echo photo?.png` | `echo` `photo1.png` `photo2.png` `photo3.png` `photoA.png` | `?`: exactly one character |
| `$ echo photo[12].png` | `echo` `photo1.png` `photo2.png` | `[12]`: one of these characters |
| `$ echo photo[0-9].png` | `echo` `photo1.png` `photo2.png` `photo3.png` | `[0-9]`: one in the range |
| `$ echo photo[!0-9].png` | `echo` `photoA.png` | `[!0-9]`: one **not** in it |
| `$ echo docs/**/*.txt` | `echo` `docs/2026/exam.txt` `docs/plan.txt` | `**`: any directories (`globstar`) |
| `$ echo {a,b,c}.pdf` | `echo` `a.pdf` `b.pdf` `c.pdf` | `{ }`: a word for each, file or not |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Globing

Patterns for file names, replaced by the shell with the matching files

```
$ ls
docs  notes.txt  photo1.png  photo2.png  photo3.png  photoA.png  report.md  todo.txt
```

| *you type* | | *the command and its parameters* | |
|---|---|---|---|
| `$ echo *.txt` | → | `echo` `notes.txt` `todo.txt` | `*`: any characters, or none |
| `$ echo photo?.png` | → | `echo` `photo1.png` `photo2.png` `photo3.png` `photoA.png` | `?`: exactly one character |
| `$ echo photo[12].png` | → | `echo` `photo1.png` `photo2.png` | `[12]`: one of these characters |
| `$ echo photo[0-9].png` | → | `echo` `photo1.png` `photo2.png` `photo3.png` | `[0-9]`: one in the range |
| `$ echo photo[!0-9].png` | → | `echo` `photoA.png` | `[!0-9]`: one **not** in it |
| `$ echo docs/**/*.txt` | → | `echo` `docs/2026/exam.txt` `docs/plan.txt` | `**`: any directories (`globstar`) |
| `$ echo {a,b,c}.pdf` | → | `echo` `a.pdf` `b.pdf` `c.pdf` | `{ }`: a word for each, file or not |
| `$ echo *.pdf` | → | `echo` `*.pdf` | no match: kept (`bash`), error (`zsh`) |

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Running a Command

*From a line of text to a process*

# Bibliography

For this section


1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Section 2.9 - *The Command Path*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 7 - *Interfața în linia de comandă*
     - Section 7.1.3 - *Funcționarea shellului*
     - Section 7.1.3.1 - *Variabila de mediu PATH*
     - Section 7.1.4 - *Comenzi interne și comenzi externe*


 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Built-in and External Commands

Not every command is a program

## Built-in commands

- run **inside** the shell, no new process
- `cd`, `export`, `unset`, `jobs`, `fg`, `bg`, `type`, `exit`, ...

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Built-in and External Commands

Not every command is a program

## Built-in commands

- run **inside** the shell, no new process
- `cd`, `export`, `unset`, `jobs`, `fg`, `bg`, `type`, `exit`, ...

## External commands

- executable files, somewhere on the disk
- the shell needs their **path**
- `ls`, `sort`, `grep`, `bash`, ...

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Built-in and External Commands

Not every command is a program

## Built-in commands

- run **inside** the shell, no new process
- `cd`, `export`, `unset`, `jobs`, `fg`, `bg`, `type`, `exit`, ...

## External commands

- executable files, somewhere on the disk
- the shell needs their **path**
- `ls`, `sort`, `grep`, `bash`, ...

```
$ type cd
cd is a shell builtin
$ type sort
sort is /usr/bin/sort
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Built-in and External Commands

Not every command is a program

## Built-in commands

- run **inside** the shell, no new process
- `cd`, `export`, `unset`, `jobs`, `fg`, `bg`, `type`, `exit`, ...

## External commands

- executable files, somewhere on the disk
- the shell needs their **path**
- `ls`, `sort`, `grep`, `bash`, ...

```
$ type cd
cd is a shell builtin
$ type sort
sort is /usr/bin/sort
```

## Why must `cd`, `unset` and `exit` be built-in?

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Built-in and External Commands

Not every command is a program

## Built-in commands

- run **inside** the shell, no new process
- `cd`, `export`, `unset`, `jobs`, `fg`, `bg`, `type`, `exit`, ...

## External commands

- executable files, somewhere on the disk
- the shell needs their **path**
- `ls`, `sort`, `grep`, `bash`, ...

```
$ type cd
cd is a shell builtin
$ type sort
sort is /usr/bin/sort
```

## Why must `cd`, `unset` and `exit` be built-in?

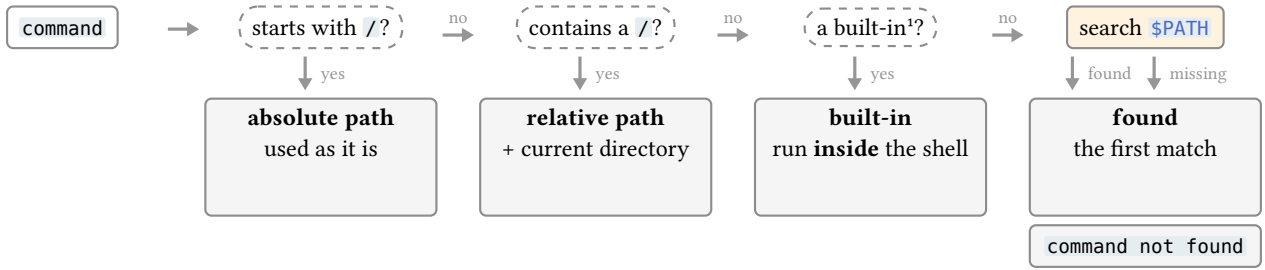
💡 they change **the shell itself**: `cd` its current directory, `unset` its variables, `exit` stops it

💡 a child process can only change **itself**, never its parent

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Which File?

External commands need a **path**: the shell turns the first word into a file to start



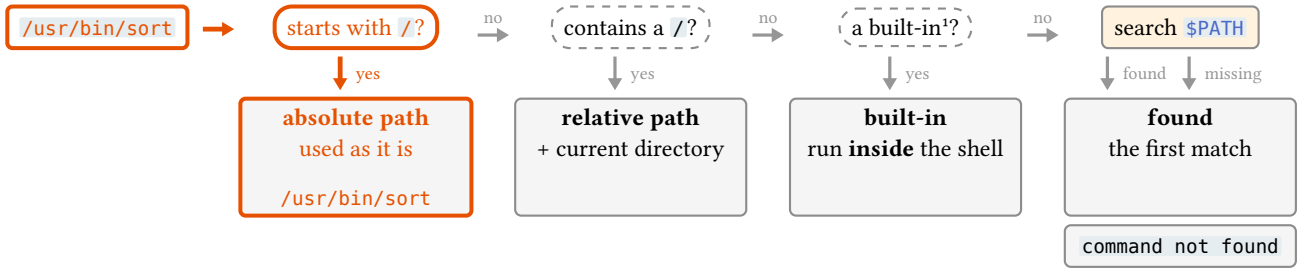

💡 the current directory is `/home/alice`

[1] it is actually a *function*, *alias* or *builtin*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Which File?

External commands need a **path**: the shell turns the first word into a file to start




*you type*

```
$ /usr/bin/sort fruits.txt
```

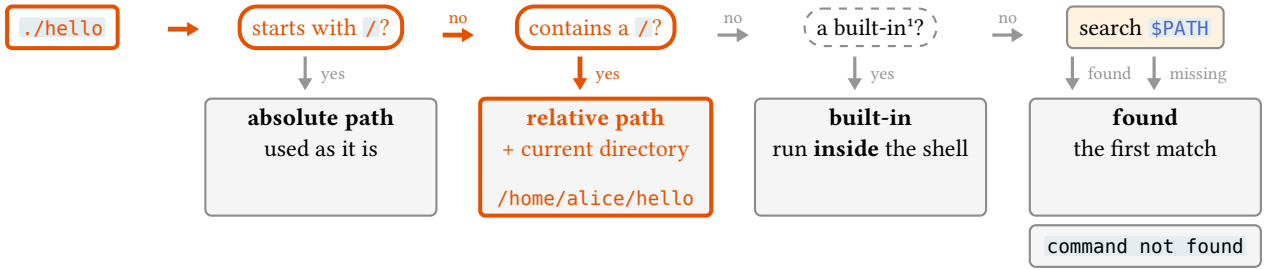
✅ an absolute path is used as it is, no search

[1] it is actually a *function*, *alias* or *builtin*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Which File?

External commands need a **path**: the shell turns the first word into a file to start




*you type*

```
$ ./hello
Hello!
```

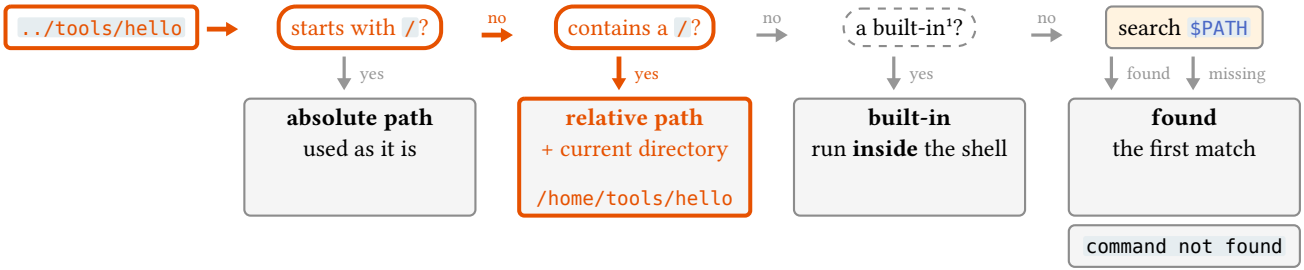
✅ `/home/alice` + `./hello` → `/home/alice/hello`

[1] it is actually a *function*, *alias* or *builtin*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Which File?

External commands need a **path**: the shell turns the first word into a file to start




*you type*

```
$ ../tools/hello
Hello from tools!
```

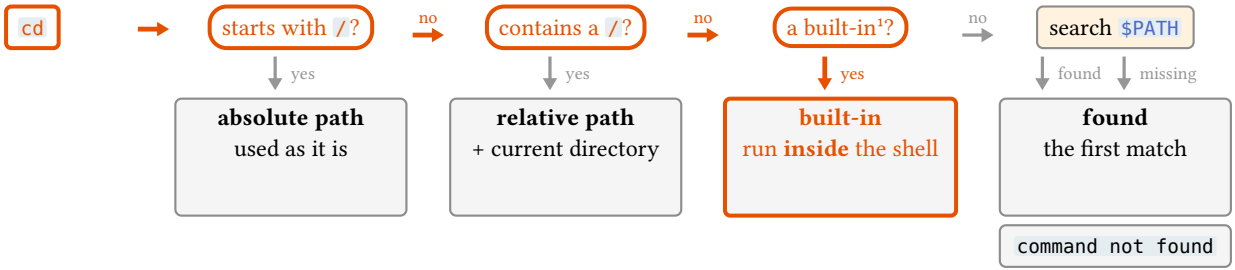
✅ `/home/alice` + `../tools/hello` → `/home/tools/hello`

[1] it is actually a *function*, *alias* or *builtin*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Which File?

External commands need a **path**: the shell turns the first word into a file to start




*you type*

```
$ cd /tmp
```

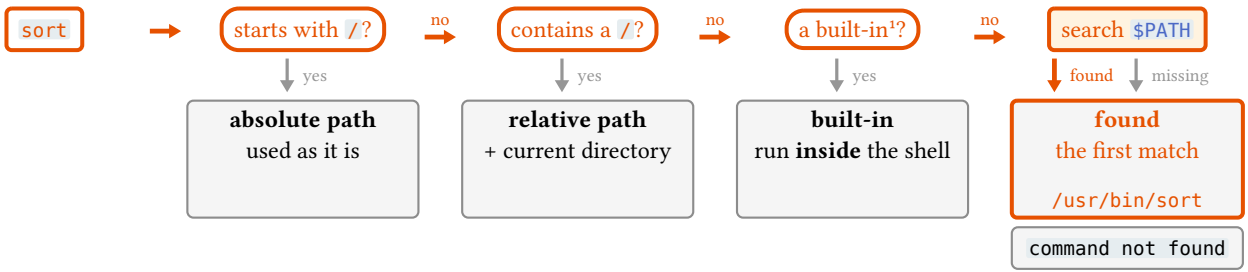
✅ `cd` is not a file, the shell does it itself

[1] it is actually a *function*, *alias* or *builtin*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Which File?

External commands need a **path**: the shell turns the first word into a file to start




*you type*

```
$ sort fruits.txt
```

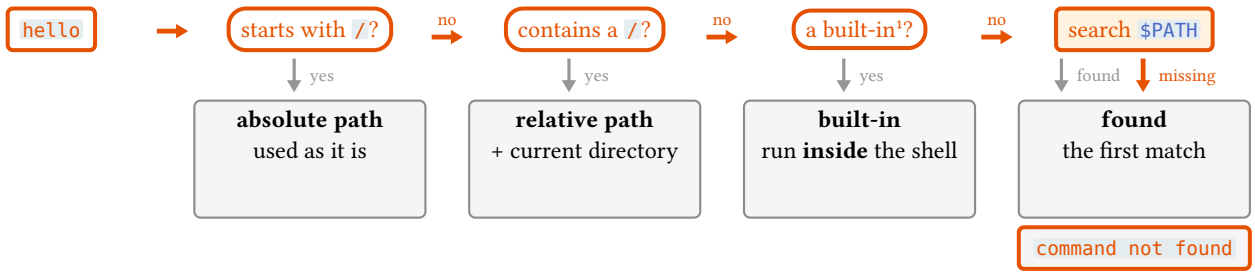
✅ the first match in `PATH`: `/usr/bin/sort`

[1] it is actually a *function*, *alias* or *builtin*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Which File?

External commands need a **path**: the shell turns the first word into a file to start




*you type*

```
$ hello
bash: hello: command not found
```

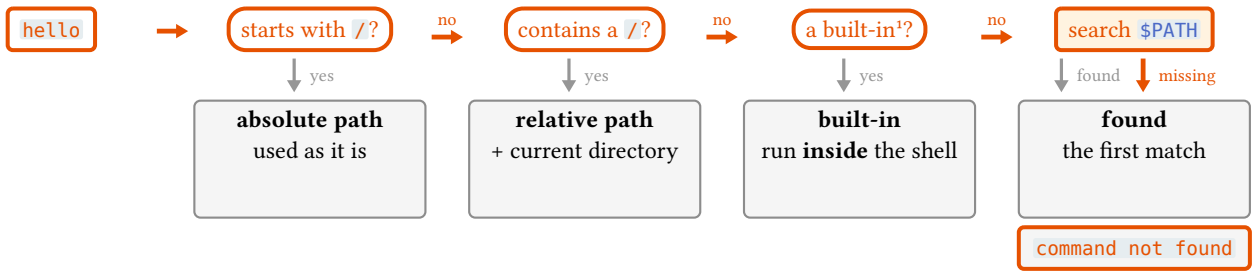
❌ no `hello` in `$PATH`: the shell does **not** look in the current directory

[1] it is actually a *function*, *alias* or *builtin*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# Which File?

External commands need a **path**: the shell turns the first word into a file to start




*you type*

```
$ hello
bash: hello: command not found
```

🐧 `exec` only starts a path: turning a name into a path is the job of the shell[2]

[1] it is actually a *function*, *alias* or *builtin*

[2] It is actually the job of `libc`, the C library, the shell uses the `execvp` function

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# `$PATH` Search

Where the shell looks for commands

```
$ sort fruits.txt
```

💡 a command name without a `/`: the shell has to find its executable file

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# `$PATH` Search

Where the shell looks for commands

```
$ sort fruits.txt
```

↓ *`sort` has no `/`: look it up in the environment variable `PATH`*

```
PATH=/home/alice/.local/bin:/usr/local/bin:/usr/bin:/bin
```

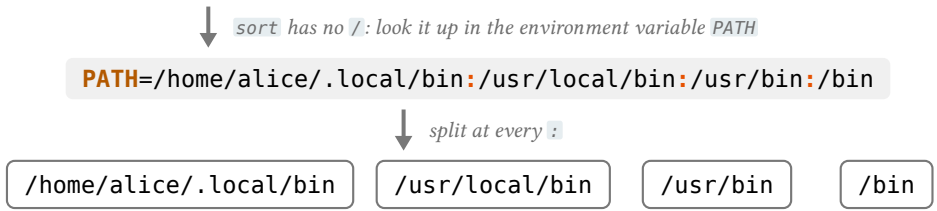
💡 `PATH` is an environment variable: a list of directories, separated by `:`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# `$PATH` Search

Where the shell looks for commands

```
$ sort fruits.txt
```



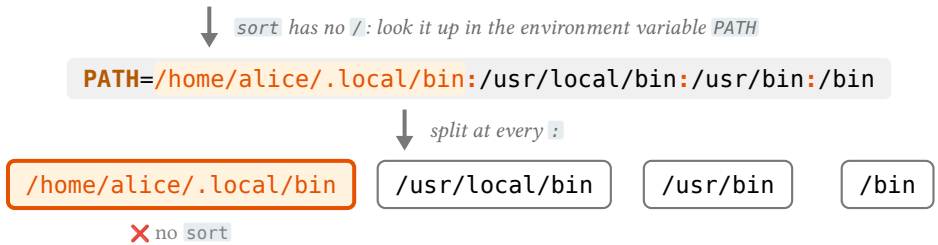

💡 the shell splits it into directories

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# `$PATH` Search

Where the shell looks for commands

```
$ sort fruits.txt
```



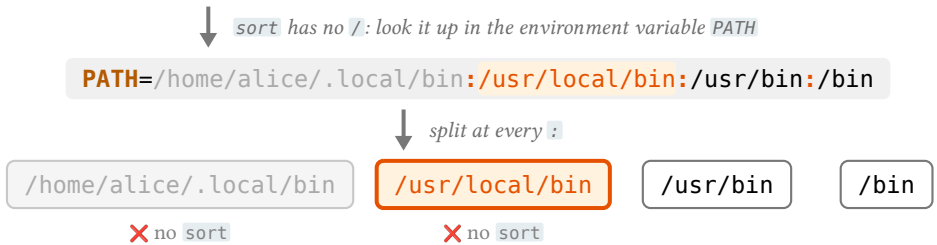

💡 it checks them **from left to right**: there is no `sort` in `/home/alice/.local/bin`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# `$PATH` Search

Where the shell looks for commands

```
$ sort fruits.txt
```




💡 no `sort` in `/usr/local/bin` either

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# `$PATH` Search

Where the shell looks for commands

```
$ sort fruits.txt
```



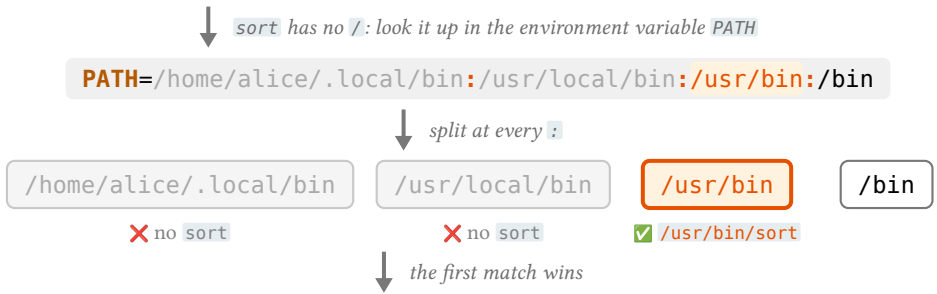

💡 `/usr/bin/sort` exists: the search stops, `/bin` is never checked

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# $PATH Search

Where the shell looks for commands

```
$ sort fruits.txt
```




```
$ /usr/bin/sort fruits.txt
```

💡 the shell runs the file it found

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# which

Do the `$PATH` search, without running the command

- prints the file the shell would run

```
$ which sort
/usr/bin/sort
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# `which`

Do the `$PATH` search, without running the command

- prints the file the shell would run
- `-a` prints **every** match, in the order of `PATH`

```
$ which sort
/usr/bin/sort
```

```
$ which -a sort
/usr/bin/sort
/bin/sort
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# `which`

Do the `$PATH` search, without running the command

- prints the file the shell would run
- `-a` prints **every** match, in the order of `PATH`
- nothing found: no output, exit status `1`

```
$ which sort
/usr/bin/sort
```

```
$ which -a sort
/usr/bin/sort
/bin/sort
```

```
$ which cd
$ echo $?
1
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

## `which`

Do the `$PATH` search, without running the command

- prints the file the shell would run
- `-a` prints **every** match, in the order of `PATH`
- nothing found: no output, exit status `1`

## Why does `which` not find `cd`?

```
$ which sort
/usr/bin/sort
```

```
$ which -a sort
/usr/bin/sort
/bin/sort
```

```
$ which cd
$ echo $?
1
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# `which`

Do the `$PATH` search, without running the command

- prints the file the shell would run
- `-a` prints **every** match, in the order of `PATH`
- nothing found: no output, exit status `1`

## Why does `which` not find `cd`?

💡 `which` is an **external** command: it only searches `$PATH`, it does not know the built-ins of the shell

💡 `type cd` asks the shell itself:

```
cd is a shell builtin
```

```
$ which sort
/usr/bin/sort
```

```
$ which -a sort
/usr/bin/sort
/bin/sort
```

```
$ which cd
$ echo $?
1
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

# We talked about

- Terminal & Shell
- Foreground & Background
- Sessions and Terminals
- Signals
- Environment Variables
- The Command Line
- Quoting, Escaping and Globbing
- Running a Command

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0