

# Princípios da Computação

The Unix shell

# Shell metacharacters

# Shell metacharacters

- Special characters the shell interprets rather than passing to the command.
  - We have already used metacharacters for file substitution:
    - `ls ex*`
    - `ls ?er`
    - `ls [abc]*`

# Escaping Metacharacters

- Sometimes those special characters are supposed to be passed to the command: they must *escape* the shell.
  - Use the backslash \ before the character.
    - E.g. \\* will not interpret the \* (and a literal \* is passed to the command)
    - Try **echo \*** and **echo \\***
  - Single quotes ( ' ') protect all characters in between (none are interpreted).
  - Double quotes ( " ") protect all characters in between, except the backslash (\), dollar sign (\$) and grave accent (`).

# Bash Metacharacters

- A lot more metacharacters for:
  - Redirecting input/output
  - Expand variables
  - Conditional execution
  - ...

# Bash Metacharacters

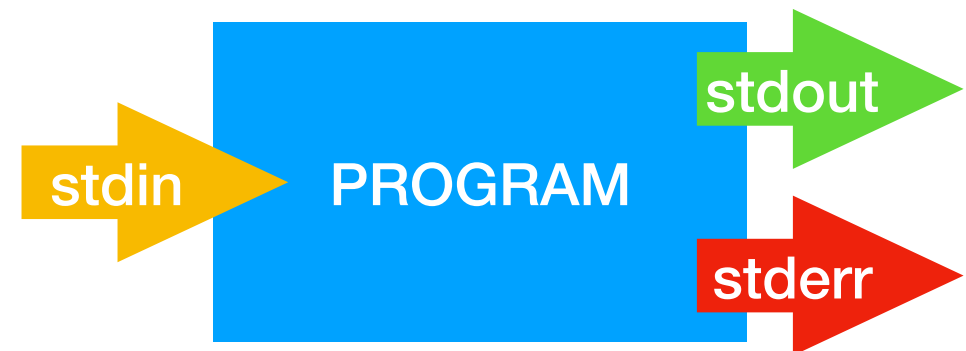
Symbol	Meaning
>	Output redirection
>>	Output redirection (append)
<	Input redirection
<<	Inline input redirection
*	File substitution wildcard; zero or more characters
?	File substitution wildcard; exactly one character
[ ]	File substitution wildcard; any one character between brackets
`cmd`	Command Substitution
\$(cmd)	Command Substitution
	Pipe commands

Symbol	Meaning
;	Sequence of commands
	OR conditional execution
&&	AND conditional execution
( )	Group commands, Sequences of Commands
&	Run command in the background
#	Comment
\$	Expand the value of a variable
\	Prevent or escape interpretation of the next character
~	User directory
..	Previous directory
.	Current working directory

# Redirecting input/output

# Standard input, output and error

- In Unix, each program has three predefined open streams:
  - standard input (stdin)
  - standard output (stdout)
  - standard error (stderr)





# Standard input (stdin)

- *File descriptor: 0*
- The program may read data from this stream...
  - as long as it is programmed to do so.
- Connected to the keyboard, by default.

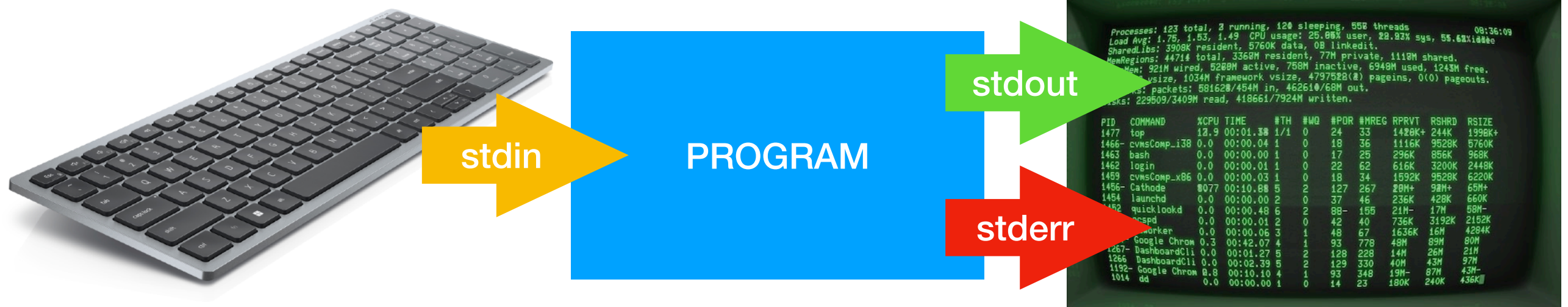
# Standard output (stdout)

- *File descriptor: 1*
- Where the program writes the regular output.
- Connected to the monitor, by default.

# Standard error (stderr)

- *File descriptor: 2*
- Where the program writes error messages.
- Connected to the monitor, by default.

# The standard case



# Redirecting streams

- Streams can be redirected to files!
  - You may want to feed your program with data from a file.
  - You may want to save output and/or error messages to a file.

# Output redirection

- `>`
  - Redirects all output and errors to a new (empty) file.
- `>>`
  - Redirects all output and errors, appending to a file.

# Output redirection

```
$ ls > my_files.txt
$ cat my_files.txt
Desktop
Documents
Downloads
Library
Movies
Music
Pictures
Public
my_files.txt
$ echo GOODBYE > my_files.txt
$ cat my_files.txt
GOODBYE
$
```

```
$ echo one > numbers.txt
$ cat numbers.txt
one
$ echo two >> numbers.txt
$ cat numbers.txt
one
two
$ echo three >> numbers.txt
$ cat numbers.txt
one
two
three
$
```

# Output redirection

- **1> or 1>>**
  - Redirects only output to a file: new or append, respectively.
- **2> or 2>>**
  - Redirects only errors to a file: new or append, respectively.



# Output redirection

```
$ my_program 1> output.txt 2>> my_errors.log  
$
```

Redirects output to the  
new (empty) file output.txt

Appends error messages  
to file my\_errors.log

# Input redirection

- `<`
  - The content of a file is redirected to the stdin of a command.
- `<<`
  - The command line arguments are directed to the stdin of a command.

# Input redirection

```
$ cat numbers.txt
```

```
one
```

```
two
```

```
three
```

```
$ sort < numbers.txt
```

```
one
```

```
three
```

```
two
```

```
$ sort < numbers.txt > sorted_numbers.txt
```

# Pipes

# Piping streams between commands

- The output stream of a command can be piped into the input stream of other command.
  - The pipe symbol `|` is placed between commands, connecting the output stream of the first to the input stream of the next command.
  - Example: **`command 1 | command2 | command3`**
- Both programs are running at the same time.
  - Output of the first command is immediately sent to the next command in the pipeline.

# Pipes

```
$ ls -l /bin | more
total 9584
-rwxr-xr-x  2 root  wheel   134240  2 Set 08:35 [
-r-xr-xr-x  1 root  wheel  1326752  2 Set 08:35 bash
-rwxr-xr-x  1 root  wheel   135408  2 Set 08:35 cat
-rwxr-xr-x  1 root  wheel   136960  2 Set 08:35 chmod
-rwxr-xr-x  1 root  wheel   152800  2 Set 08:35 cp
-rwxr-xr-x  2 root  wheel  1153408  2 Set 08:35 csh
-rwxr-xr-x  1 root  wheel   307248  2 Set 08:35 dash
-rwxr-xr-x  1 root  wheel   168032  2 Set 08:35 date
-rwxr-xr-x  1 root  wheel   185088  2 Set 08:35 dd
-rwxr-xr-x  1 root  wheel   151440  2 Set 08:35 df
-rwxr-xr-x  1 root  wheel   133952  2 Set 08:35 echo
-rwxr-xr-x  1 root  wheel   235296  2 Set 08:35 ed
-rwxr-xr-x  1 root  wheel   134800  2 Set 08:35 expr
-rwxr-xr-x  1 root  wheel   133952  2 Set 08:35 hostname
-rwxr-xr-x  1 root  wheel   134352  2 Set 08:35 kill
-r-xr-xr-x  1 root  wheel  2598896  2 Set 08:35 ksh
-rwxr-xr-x  1 root  wheel   394848  2 Set 08:35 launchctl
-rwxr-xr-x  2 root  wheel   134736  2 Set 08:35 link
-rwxr-xr-x  2 root  wheel   134736  2 Set 08:35 ln
-rwxr-xr-x  1 root  wheel   187120  2 Set 08:35 ls
-rwxr-xr-x  1 root  wheel   134128  2 Set 08:35 mkdir
-rwxr-xr-x  1 root  wheel   135552  2 Set 08:35 mv
-More-
```

# Environment variables

# Environment variables

- Environment variables store information about
  - the shell session,
  - the working environment, and
  - user defined data.



# Global and local variables

- Global variables are visible from the shell, but also from any program running inside the shell.
  - Usually store useful session data.
- Local variables are visible only from the shell (i.e. not visible to programs running inside the shell).
  - User defined variables are local.

# Seeing global variables

- Commands **env** and **printenv** display the session global variables.

```
$ printenv
[...]
SHELL=/bin/bash
HOME=/Users/johnny
LOGNAME=johnny
USER=johnny
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
PWD=/Users/johnny
OLDPWD=/Users/johnny
LANG=pt_PT.UTF-8
$
```

# Seeing local variables

- Command **set** displays all (global and local) variables.

```
$ set
(...)
SHELL=/bin/bash
HOME=/Users/johnny
LOGNAME=johnny
USER=johnny
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
PWD=/Users/johnny
OLDPWD=/Users/johnny
LANG=pt_PT.UTF-8
(...)
my_var='This is my local var.'
$
```

# Setting a user defined variable

- Assign a value to a variable using the = symbol.
  - No spaces are allowed around the = symbol!!!

```
$ my_word=Hello
$ my_sentence="Hi there"
$ set
(...)
my_sentence='Hi there'
my_word=Hello
$
```

# Using a variable

- Variables can be used on command lines.
- The value stored in the variable is obtained using the **\$** symbol (variable expansion).

```
$ echo $my_sentence I am $USER
Hi there I am johnny
$ ls /Users/johnny/docs
PRCMP_assignment3.pdf
PRCMP_assignment4.pdf
$ ls $HOME/docs
PRCMP_assignment3.pdf
PRCMP_assignment4.pdf
$
```

Name	Value
my_sentence	Hi there
HOME	/Users/home/johnny
USER	johnny

# Promote a variable to global

- The export command makes a variable global.

```
$ export DOCS_DIR=$HOME/docs
$ printenv
(...)
DOCS_DIR=/Users/johnny/docs
(...)
$
```

# Conditional execution

# Exit status

- When a program ends, it informs the shell about its exit status.
- The exit status is an integer:
  - 0 (zero) means SUCCESS
  - Any other value means FAILURE
- The exit status of the last command can be accessed in the variable `$?`
  - Try `echo $?`



# Conditional execution

- The exit status can determine the execution of the next program.
- **&&** executes the next program if the previous succeeded
- **||** executes the next program if the previous failed.

```
$ true && echo OK || echo Failed
OK
$ false && echo OK || echo Failed
Failed
$
```

```
$ true
$ echo $?
0
$ false
$ echo $?
1
$
```

# Conditional execution

```
$ ls && echo OK || echo FAILED
docs      numbers.txt      letters.txt
OK
$ ls no_file_here && echo OK || echo FAILED
ls: no_file_here: No such file or directory
FAILED
$
```

A more useful example. What is it supposed to do?

```
$ mkdir $HOME/docs && mv $HOME/Downloads/*.pdf $HOME/docs
$
```

# Job control

# Foreground jobs

- When you launch a program, it runs in the foreground.
  - The shell loads the program and suspends, waiting for the program to finish.
  - When the program ends, the shell returns.
- However, it is possible to launch a program running in the background...

# Background jobs

- When a program is launched in the background, the shell does not suspend, returning immediately.
- To launch a program in the background, the **&** symbol is added at the end of the command line.

# Background jobs

- The command should not be interactive!
  - No keyboard input, no output for the monitor!
- Great for programs that run for a long time.

```
$ git clone git://git.kernel.org/pub/scm/.../linux.git my-linux &  
[1] 34451  
$
```

The shell returns immediately.

Job 1, with identifier 34451 started.

Run command in the background

# List jobs in the background

- The **jobs** command lists the uncompleted jobs.

```
$ jobs
[1]      running    git clone ...
[2]  -  running    vlc -I dummy -q ...
[3]  +  running    my_
$
```

# Switching between foreground and background

- **CTRL-Z** interrupts (suspends) the job in the foreground.
- The **bg** command puts a suspended job to run in the background.
- The **fg** command puts a suspended job to run in the foreground.



# Switching between foreground and background

```
$ sleep 60
```

```
^Z
```

```
bash: suspended sleep 60
```

```
$ jobs
```

```
[1] + suspended sleep 60
```

```
$ bg %1
```

```
[1] + continued sleep 60
```

```
$ jobs
```

```
[1] + running sleep 60
```

```
$ fg %1
```

```
[1] + running sleep 60
```

Pressing CTRL-Z suspends the job running in the foreground.

Job %1 (number 1) continues running in the background.

Job %1 is promoted to run in the foreground.