

This work is licensed under a Creative Commons license



Attribution-NonCommercial-NoDerivatives 4.0 International
(CC BY-NC-ND 4.0)

You are free to:

Share copy and redistribute the material in any medium or format.

Under the following terms:

Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial You may not use the material for commercial purposes.

NoDerivatives If you remix, transform, or build upon the material, you may not distribute the modified material.

Shell e terminali

Giovanni Lagorio

`giovanni.lagorio@unige.it`

`https://csec.it/people/giovanni_lagorio`

Twitter & GitHub: `zxgio`

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy



`www.zenhack.it`

Outline

- 1 Introduzione
- 2 Uso interattivo della shell
- 3 Scripting
- 4 Job control

Introduzione

Nei sistemi Unix-like

- La **shell** è un **interprete di comandi**, che potete usare sia in modalità interattiva, sia tramite degli *script*
- Il nome *shell* deriva dal fatto che si tratta di un **programma utente** che offre un'interfaccia ad alto livello alle **funzionalità del kernel**
- Noi faremo riferimento alla GNU **bash**, la *Bourne-Again SHell*, un gioco di parole su Stephen Bourne, autore dell'antenata *sh* di Unix
 - Solo un'introduzione, RTFM:
<https://www.gnu.org/software/bash/manual/>
 - In generale, se dovete imparare *una sola cosa di tutto il corso*: RTFM
 - L'unico comando che vi dovete ricordare: `man`
- Un libro gratuito sull'uso di shell e principali comandi è **The Linux Command Line**, liberamente scaricabile da:
<http://linuxcommand.org/tlcl.php>

Per interagire con la shell dovete utilizzare un **terminale**

Terminale = evoluzione di telescrivente (TeleTYpe)

Dispositivo meccanico, nato per il telegrafo alla fine del 1800 (!)



A teleprinter (teletypewriter, teletype or TTY)

<https://en.wikipedia.org/wiki/Teleprinter>

Evoluti negli anni '60/'70 in...



https://en.wikipedia.org/wiki/Computer_terminal

- In Unix, (quasi) tutto è un file e **file speciali corrispondono ai terminali** collegati
- Il nome TTY è rimasto: **/dev/tty** è sinonimo del terminale associato a un processo (vedere `tty(4)`)

Naturalmente, in pochi utilizzano ancora “veri” terminali...

Virtual console e pseudo-terminali

In Linux

- diverse **virtual console** (`ttyn`), sono terminali virtuali che condividono la tastiera e lo schermo
 - si può cambiare quella attiva premendo `ALT+Fn`, o
 - `CTRL+ALT+Fn` dalla modalità grafica
 - nelle versioni più recenti di Ubuntu `tty1` è il display/login manager, `tty2` l'ambiente grafico e solo le console da `tty3` a `tty6` sono in modalità testo
- negli attuali ambienti desktop, es. Gnome, si utilizza un **emulatore di terminale**, cioè un programma che emula un terminale testuale
 - il collegamento fra l'emulatore di terminale e un'istanza della shell avviene tramite l'uso di **pseudo-terminali**, AKA `pty`

pty(7) = pseudo-terminal = coppia di dispositivi a caratteri, detti *master* e *slave*, che permettono l'emulazione di terminali

- creando il lato “master” un processo, per es. Gnome-terminal, crea un nuovo terminale (virtuale)
- altri processi possono aprire il corrispondente “slave” `/dev/pts/...` e interagire con esso come fosse un vero terminale

Per dettagli vedere `pty(7)` e `pts(4)`

Standard file descriptor

Ogni processo usa tre **file descriptor**:

- 0 standard input (stdin/cin)
- 1 standard output (stdout/cout)
- 2 standard error (stderr/cerr)

una shell interattiva legge da stdin e scrive su stdout/stderr, che corrisponde a /dev/tty

- scrivere su `/dev/tty`
- ...e altri terminali
- `reptyr -l`
- `/proc/self/fd`

Parentesi: directory principali standard

/ radice

/bin e /sbin comandi essenziali e quelli per l'amministrazione

/boot file per il boot del sistema

/dev file speciali che corrispondono a dispositivi

/etc file di configurazione del sistema

/home e /root home degli utenti (non root) e root

/lib* librerie

/media e /mnt *mount-point* per i media rimovibili e altri FS

/proc e /sys FS virtuali, interfaccia alle strutture dati del kernel

/tmp file temporanei, spesso un ramdisk nei sistemi moderni

/usr gerarchia secondaria (/usr/[s]bin, /usr/lib*, ...), che può essere condivisa in sola lettura fra più host

Tra (pseudo-)terminale e processi, c'è la **disciplina di linea**

- Normalmente si usa la versione *cooked/canonical*; che gestisce il buffering, l'editing, l'echo, caratteri speciali (backspace, Ctrl+C/D/H/Q/S/Z...) ...vedere `stty(1)`
- Alcune applicazioni, per esempio `vi`, utilizzano la versione *raw*
- Provate, ad esempio `stty -icanon && cat`

Per approfondimenti:

blog.nelhage.com/2009/12/a-brief-introduction-to-termios e

<http://www.linusakesson.net/programming/tty/>

Sequenze di escape

- https://en.wikipedia.org/wiki/ANSI_escape_code
a standard for in-band signaling to control the cursor location, color, and other options on video text terminals and terminal emulators. Certain sequences of bytes, most starting with Esc (ASCII character 27) and '[', are embedded into the text, which the terminal looks for and interprets as commands

per esempio, provate: `echo -e '\x1b[38;5;123mciao'`

- alcuni comandi, per esempio `ls`, di default usano i colori solo se il loro standard output è un terminale
 - provate, per esempio: `ls -l /etc` e `ls -l /etc | less`
 - è possibile usare `less` e avere i colori?

Outline

- 1 Introduzione
- 2 **Uso interattivo della shell**
- 3 Scripting
- 4 Job control

Cosa fa la shell?

- in modalità interattiva, stampa un **prompt** (=una sequenza di caratteri che indica che è in attesa di comandi)
 - per un prompt più carino/informativo: **Liquid prompt**
<https://github.com/nojhan/liquidprompt>
- legge l'input
 - lo spezza in *token*: **parole** e **operatori**
 - espande gli alias
 - fa il *parsing* in comandi semplici e composti
- esegue varie **espansioni** (`{}` ~ `$` `*?[]`)
- esegue le eventuali **redirezioni dell'I/O**
- esegue il “comando”, tipicamente **un eseguibile/script** esterno (ma può essere una funzione o un comando *built-in*)
- tipicamente, ne aspetta la terminazione
- ricomincia

I comandi più importanti

- `man`
- `help`
- spesso, opzione `-h` o `--help`
- `type`

Per visualizzare/capire i comandi più complessi, può essere utile il sito [Explain Shell](https://explainshell.com/) <https://explainshell.com/>

Per colorare

- le pagine di manuale, considerate `Bat` come *pager*:
<https://github.com/sharkdp/bat>
- l'output di alcuni comandi, `Generic Colouriser`
<https://github.com/garabik/grc>

Escaping e quoting

Alcuni caratteri hanno significati speciali (per esempio, lo spazio) e devono essere messi fra virgolette o preceduti da un backslash per essere utilizzati

- i singoli apici permettono l'inserimento di qualsiasi carattere, a parte gli apici stessi
- le doppie virgolette trattano in maniera speciale \$ ` e \
- la forma \$'...' permette di espandere i \... alla ANSI-C; per esempio: `echo $'ciao\nmondo\x21'` → `ciao<newline>mondo!`
 - che differenza c'è fra:
`echo -e 'ciao\nmondo'`
`echo 'ciao\nmondo'`
`echo $'ciao\nmondo' ?`

Tab-completion e History

- tab-completion (usare `ctrl+V` tab, per inserire un tab)
- freccia su e giù
- history
- `!!`, `!n` e `!str`
- `ctrl+R`



Type a simple
command into
the console



Press up key
dozens of times
until you find it.

<https://twitter.com/wehackpurple/status/1453771155337191424>

Variabili

- creare/aggiornare: *name=value*
 - *attenzione*: senza spazi, `A = 10` cerca di invocare `A`, passando come argomenti `"=`" e `"10"`
- leggere: *\$varname* o *\${varname}*
 - la seconda forma è necessaria per accedere agli array; es:
`${BASH_VERSINFO[*]}`, `${BASH_VERSINFO[0]}`,
`${BASH_VERSINFO[1]}`, ...
 - sono possibili varie espansioni/sostituzioni, per esempio:
 - `${varname:-value}`
 - `${varname:=value}`
 - `${varname/pattern/string}`
 - `${varname:offset}` e `${varname:offset:len}`
 - `${#varname}`
 - ...
- `$$` corrisponde al PID della bash

Variabili d'ambiente

- `export name[=value]` per specificare una **variabile d'ambiente**
 - senza `export` si definisce una variabile della shell, a meno che la variabile sia già di ambiente
 - per esempio, `PS1` configura il prompt principale di `bash`
- usate per specificare impostazioni; per esempio:
 - `PATH` specifica alla shell dove cercare i comandi (quando il nome non contiene `/`)
 - `MANPAGER` indica a `man` quale programma utilizzare per visualizzare le pagine di manuale
 - ...
- ogni processo ha la sua copia delle variabili d'ambiente
- il comando `env`, invocato senza argomenti, le elenca

Startup script

Per fare in modo che gli export (o altri comandi) vengano eseguiti ogni volta che aprite una shell, dovete aggiungerli a uno script

Tipicamente `~/.bashrc`, ma anche/invece in `.[bash_]profile`, o ...
www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html

Altri comandi da considerare:

- `shopt -s autocc`
- `set -o nounset` o, equivalentemente, `set -u` per far sì che sia un errore leggere una variabile inesistente
 - stranezza: `+o` per disabilitare
- `set -o errexit` o, equivalentemente, `set -e` per far sì che uno script termini al primo errore

Dopo il *parsing*:

- $s_1\{x,y,z\}s_2$ e $s_1\{x..y[..incr]\}s_2$
- \sim e $\sim user$
- $\$var$, $\${var}$, ...
- $\$(cmd)$ e $'cmd'$
- $\$((expr))$
- le espansioni che non sono avvenute all'interno di doppie virgolette (e contengono spazi, tab o newline) vengono spezzate in parole separate
- *pattern-matching* sui nomi di file con *wildcard* ($*$ $?$ $[...]$)
 - i nomi che iniziano con punto sono “nascosti”, per convenzione
 - $**$ espande ricorsivamente nelle directory, se l'opzione `globstar` è abilitata (di default non lo è)

- redirezione input: `<fname`
- redirezione output: `>fname` e `>>fname`
- due file speciali che possono tornare comodi:
 - `/dev/null`
 - `/dev/zero`

vedere `NULL(4)`, così come il comando `yes`

Outline

- 1 Introduzione
- 2 Uso interattivo della shell
- 3 Scripting**
- 4 Job control

- **semplici**: una sequenza di parole, separate da blank
- **pipeline**: sequenza di comandi, separati da `|` o `|&`
- **liste**: sequenze di pipeline, separati da `;`, `&`, `&&`, o `||`, opzionalmente terminate da `;`, `&`, o newline
 - possono essere racchiuse fra `(` e `)`, o `{` e `}`, per applicare un'unica redirectione a tutti i comandi nella lista

Exit status

- Ogni comando restituisce un **exit status**, che finisce in \$?
 - Si può uscire dalla bash con `exit n`
 - Per convenzione, 0 OK, non-0 errore
 - Nei contesti “booleani” 0 → True, non-0 → False
 - No, non mi sono sbagliato, è proprio 0 true, non-zero false (!!!)
- && e || possono essere usati per comporre *pipelines*

- i comandi possono essere *tipicamente* interrotti premendo **Ctrl+C**
- quando un terminale è in modalità canonica, coi settaggi di default, Ctrl+C corrisponde a inviare al comando un **segnale SIGINT (2)**
 - tecnicamente, inviato al *gruppo* in *foreground*; ne parliamo dopo
 - per l'elenco dei segnali vedere `signal(7)`

il comportamento di default, in risposta al segnale, è la terminazione

- per inviare segnali, si può usare il comando **kill**
 - di default invia il segnale di terminazione, **SIGTERM (15)**
 - di default, termina il processo
 - fra i vari segnali, **SIGKILL (9)**, termina il processo
 - non può essere catturato, bloccato o ignorato
- l'**exit-status** di un processo **terminato per un segnale s** è **$(s + 128)$**
- **SIGSEGV (11)**, che avrete certamente incontrato ☺ → 139

```
function name { cmd-list }
```

```
name () { cmd-list }
```

- gli argomenti sono \$1, \$2, ...
 - il loro numero è \$#
- il comando `return` può essere usato come in C/Python
- `local` può essere usato per dichiarare variabili locali
- esempio (malvagio 😊), una *fork-bomb*: `:(){ :|:& };;:`

Condizioni

- c'erano una volta test e [
 - [1 -eq 2]
 - [1 = 2]
 - [1 == 2]
 - [-f /etc/passwd]
 - [-w /etc/passwd]
 - test 1 -ne 2
- versione moderna (bash built-in): `[[...]]`
 - si possono usare < e > per confrontare *stringhe*, attenzione ai tipi:
[10 < 2] vs [[10 < 2]] vs [[10 -lt 2]]
 - le stringhe vuote non sono un problema; per esempio, provate:
[-f \$VAR_NON_ESISTENTE] vs
[[-f \$VAR_NON_ESISTENTE]]
 - ...

Costrutti condizionali e cicli

- if

```
if tst-commands; then
    consequent-commands;
[elif more-tst-commands; then
    more-consequents;]
[else alternate-consequents;]
fi
```

- for name in words; do commands; done

- while tst-commands; do consequent-commands; done

- for ((expr1 ; expr2 ; expr3)) ; do commands ; done

- ...

Un esempio

```
#!/bin/bash
for i in {0..255}; do
    printf "\x1b[38;5;%dm%d " $i $i
done
echo
```

Peculiarità della shell

Una serie di errori comuni sono descritti in:

<http://mywiki.woledge.org/BashPitfalls> e un'utility che controlla e suggerisce migliorie: <https://github.com/koalaman/shellcheck>

Python

Quando le cose si complicano, meglio passare a un linguaggio di scripting più moderno; per esempio, Python <https://www.python.org/>

Outline

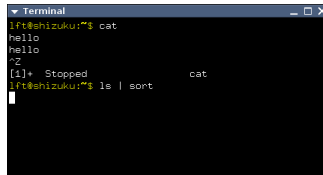
- 1 Introduzione
- 2 Uso interattivo della shell
- 3 Scripting
- 4 Job control**

Meccanismo che permette di **gestire più lavori con uno stesso terminale**, sospendendo/riprendendo l'esecuzione di gruppi di processi, i **job**

- importante coi terminali “veri”, meno con gli emulatori
- i job sono raggruppati in **sessioni**, una per terminale
- per ogni pipeline che inserite nella shell, viene creato un job

- quando apriamo l'emulatore di terminale, lui
 - crea un pty e gestisce la parte *master*
 - crea un nuovo processo *p*, che
 - crea una nuova sessione; *p* diventa *session-leader*
 - apre lo *slave*, che diventa il *terminale di controllo* della sessione; *p* è detto anche *controlling process*
 - esegue la shell
- quando chiudiamo la finestra corrispondente, viene “disconnesso”
 - il kernel manda il segnale SIGHUP al *session-leader*, la shell
 - la shell, a sua volta, lo manda a tutti i job che gestisce
 - di default, SIGHUP termina i processi

Job control — un esempio



```
ift@shizuku:~$ cat
hello
hello
^Z
[1]+  Stopped                  cat
ift@shizuku:~$ ls | sort
```

potrebbe corrispondere a:

Session 100		Session 101			
Job 100		Job 101	Job 102	Job 103	
XTerm (100)		bash (101)	cat (102)	ls (103)	sort (104)
stdin: -		stdin: /dev/pts/0	stdin: /dev/pts/0	stdin: /dev/pts/0	stdin: pipe0
stdout: -		stdout: /dev/pts/0	stdout: /dev/pts/0	stdout: pipe0	stdout: /dev/pts/0
stderr: -		stderr: /dev/pts/0	stderr: /dev/pts/0	stderr: /dev/pts/0	stderr: /dev/pts/0
PPID: ?		PPID: 100	PPID: 101	PPID: 101	PPID: 101
PGID: 100		PGID: 101	PGID: 102	PGID: 103	PGID: 103
SID: 100		SID: 101	SID: 101	SID: 101	SID: 101
TTY: -		TTY: /dev/pts/0	TTY: /dev/pts/0	TTY: /dev/pts/0	TTY: /dev/pts/0

Preso da: <http://www.linusakesson.net/programming/tty/>

Job in fore/back-ground

- ogni terminale può avere
 - un job in **foreground** che può
 - leggere dal terminale
 - ricevere segnali se l'utente usa `^C`, `^Z`, ...
 - tanti job in **background**
 - una pipeline terminata da `&` viene eseguita in background; `$!` è il PID del job
- `^Z` sospende un processo, che si può continuare la sua esecuzione...
 - in foreground, con **`fg [n]`**
 - in background, con **`bg [n]`**

il comando `jobs` elenca i *job* attivi

- con alcuni comandi built-in (es. `kill`, `wait`, `disown`) i job possono essere identificati con **`%n`**; l'ultimo fermato quando era in foreground, o fatto direttamente partire in background, è **`%%`**

<https://www.gnu.org/software/bash/manual/bash.html#Job-Control>