Sistemi Operativi

Riassunto delle lezioni di Laboratorio A.A. 2018/2019

Matteo Franzil

7 marzo 2019

Indice

1 Comandi Bash

La seguente è una lista, non esaustiva, dei comandi visti a lezione per l'uso della shell Bash.

1.1 Comandi

- `free` → memoria libera
- `pwd` → cartella corrente
- `df` → partizioni
- `dirname` → ottiene il nome della cartella di un dato file

1.2 Piping

- `;` → esecuzione in sequenza
- `&&` `||` → esecuzione in sequenza con corto circuito
- `|` → piping classico
- `>` → redirect su file classico
- `1>` → redirect di `stdout`
- `2>` → redirect di `stderr`

1.3 Intestazioni

- `#! /bin/bash` → intestazione
- `#Script "Hello World"` → intestazione nominativa

1.4 Scripting

- `var=VALORE` → assegnazione (sono sempre stringhe)
- `${var}` → stampa (con eventuale esecuzione)
- `$@` → equivalente ad `argv`
- `$#` → equivalente ad `argc`
- `$1`, `$2`, `$3`, `$n` → i primi 9 parametri passati
- `shift` → cestina il primo argomento nella lista
- `" ... "` → crea un comando / stringa interpolando variabili
- `$(( ... ))` → contengono espressioni aritmetiche: se all'interno uso una `$var`, viene sostituita come fosse una macro, se uso il singolo contenuto di `var` allora viene inserito il valore come avesse le parentesi.
- `bc` → comando che supporta il piping in entrata, per eseguire operazioni in float
- `# Commenti` → commento classico di singola riga
- `$?` → valore di ritorno globale, usato dagli script (tipo return 0 in C): ha significato booleano (**0** niente errori, arriva fino a **256**)

1.5 Booleani

- `test ...` → si aspetta un espressione booleana, e internamente modifica il registro booleano visto prima:
- `-eq, -ne, -lt, -gt` → operandi booleani utilizzati
- `[ ... ]` → sintassi di testing alternativa (gli spazi sono importanti!); attenzione che le parentesi quadrate sono considerate come ultimo comando eseguito
- `[[ ... ]]` → raggruppamento di espressioni booleani per utilizzare operatori comuni (>, < ...)-
- `-f (file) -d (directory)` → verificano l'esistenza di un dato file/cartella.

1.6 Cicli

- `if [ ... ]; then ... ; else ...; fi` → costrutto if standard
- `case $var in; a|b) ... ;; c) ... ;; esac` → costrutto switch standard
- `for (( init ; case; step )); do; ...; done` → costrutto for standard
- `until [[ ... ]] ; ...; done` → while negato
- `while [[ ... ]] ; ...; done` → while standard

1.7 Funzioni

- `func() { ... }` → accedibile come fossero degli script (`func arg1 ... argn`)

1.8 Varie

- `$( ... )` → sottoshell che esegue comandi in un processo separato
- `BASH_SOURCE[0]` → contiene il nome dello script in esecuzione
- `exit n` → uscita con codice d'errore
- `1> 2> ... n>` → redirezionamento dei diversi canali sui file, come visto prima (1 = stdout, 2 = stderr); è possibile redirezionare stderr su stdout e viceversa tramite il comando `2>&1`, oppure mettendo tutto su file: `1>output 2>&1`. L'ordine in cui vengono interpretati i redirect sono da destra a sinistra.

2 Make

2.1 Introduzione

Make è un tool utilizzato per automatizzare processi all'interno di sistemi Unix. Viene principalmente usato per automatizzare la compilazione dei file.

I Makefile sono composti da regole, composte da un **identificativo** (o nome) e da una **ricetta** (una serie di comandi indentati con una tabulazione `\t` rispetto al nome della ricetta):

```
regola:
    echo "Ciao"
```

I file vengono poi eseguiti tramite il comando `make -f nome_file.makefile`. Notare come i comandi vengono anche stampati su `stdout` oltre a essere eseguiti. Questa funzionalità può essere sfruttata per stampare a video anche i procedimenti che vengono eseguiti dal file, ma può risultare fastidioso: si può quindi inserire una chiocciola `@` che impedisce la stampa del comando.

Il nome del file è opzionale e se non presente viene cercato all'interna della cartella corrente un nome corrispondente a `Makefile`.

In ogni ricetta, ogni riga è trattata singolarmente come un singolo processo, caricando la giusta cartella di lavoro di volta in volta. Bisogna quindi fare attenzione all'uso di `cd`.

All'inizio di ogni regola, si può inoltre specificare una o più *dipendenze* che devono essere rispettate prima di avviare la regola corrispondente. A tal scopo si usa generalmente dare il nome del file che verrà generato alla regola stessa, se la regola ne genera qualcuno:

```
file.cc: dipendenza1 dipendenza2...
    comandi per generare file.cc...
```

In tutti gli altri casi si parla di *pseudo-regole* ed il nome può essere attribuito di fantasia.

2.2 Definizione di macro

All'interno dei makefile si possono definire delle macro, che possono essere utilizzate con una sintassi analoga a bash. Ciò risulta problematico in quanto `$` è interpretato prima da Makefile e poi da Bash.

```
MACRO1=Pippo
main:
    @echo Hello $(MACRO1)
```

Le macro possono inoltre essere sovrascritte al momento dell'avvio di `MAKE` con la medesima sintassi (ma è necessario che la macro stessa sia definita all'interno del file).

Con la sintassi `$(...)` è possibile accedere a una sotto-shell, analoga a quella di Bash, per eseguire comandi all'interno del Makefile stesso. In questo sotto-comando tutti gli `\n` vengono tramutati in spazi.

3 gcc

3.1 Introduzione

`gcc` è un compilatore multi source/target per compilare i file sorgenti C. Viene utilizzato in combinazione con Make per compilare velocemente più sorgenti.

3.2 Flag

- `gcc main.c -S` → compila in codice Assembly
- `gcc main.c -E` → esegue solo il preprocessore
- `gcc main.c -c` → compila senza linkare
- `gcc main.c -o` → genera i file oggetto binari