Sistemi Operativi

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

Matteo Franzil

14 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
- `gcc main.c -g` → compilazione con mapping per `gdb`
- `gcc main.c -DNOME=VAL` → compilazione con definizione di macro NOME=VAL

3.3 gdb

`gdb` è un debugger usato in combinazione con `gcc`. In questo ambiente controllato, possiamo tenere traccia dello stato dell'esecuzione del programma mappando opportunamente le variabili (in quanto i nomi vengono persi durante la compilazione).

Per iniziare, si usa il comando `run` nella shell interattiva che viene presentata da `gdb`.

Punto centrale del debugging è il **breakpoint**; si tratta di un punto, indicato manualmente con `break *`, che causerà l'interruzione dell'esecuzione del programma una volta raggiunto. A questo punto è possibile o usare `step <count>` e `next <count>` per fare uno o più step generici o principali, oppure interferire volontariamente in maniera più o meno pesante, ad esempio usando `print` per ottenere i valori delle variabili oppure `set variable vname=value` per modificare il valore. Si può infine continuare l'esecuzione con `continue`.

4 C

4.1 Introduzione

Il lingaggio C è un linguaggio di "basso livello", alla base dei sistemi Unix e dotato di una struttura minimale e ben definita, che velocizza le prestazioni omettendo funzioni di alto livello presenti invece in altri linguaggi.

Prima della compilazione vi è un *pre-processore* il cui compito è interpretare le direttive e gli alias definiti all'inizio del file (tramite l'apposito carattere `#`), generando il cosiddetto file "sorgente direttivo". Le direttive sono varie, seguono le principali:

- `#include <lib>` → inclusione di codici libreria
- `#include "file"` → inclusione di file `.h` esterni
- `#define NOME TESTO` → macro
- `#define FUNC(X,Y) (X) (Y)` → macro con funzioni
- `#if`, `#ifdef NOME`, `#ifndef NOME`, `#else`, `endif` → controlli

Il linguaggio viene quindi compilato e analizzato una sola volta: prima viene trasformato in codice intermedio (Assembly), poi in file oggetto `.o` e infine linkato ai file libreria e trasformato in linguaggio macchina `.out`.

4.2 Sintassi

- `int printf(const char *format, ...)` → stampa con formattazione: `%[flags][width][.precision][length]specifier`; alcuni specifier sono `%c`, `%d`, `%li`, `%s`, rispettivamente `char`, `int`, `long`, `char*`
- Si può ovviare alla mancanza dei booleani tramite un apposita macro: `#define BOOL char; #define FALSE 0; #define TRUE 1;`. Si noti che questa variabile può comunque, analogamente a un char, contenere valori fino a 255; è quindi cura del programmatore utilizzarla in maniera corretta e non confusionaria.

4.3 Organizzazione della memoria

Data l'immagine del processo in memoria, partendo dall'indirizzo più basso, troviamo:

- OS Kernel
- Text segment: istruzioni della CPU
- Data segment: variabili globali
- BSS: variabili non inizializzate
- Heap e stack: allocazione dinamica automatica e personalizzata

In C, non c'è garbage collection e la memoria allocata dinamicamente va esplicitamente deallocata per evitare fastidiosi memory leak. AL momento dell'allocazione in memoria, distinguiamo tra **variabili statiche**, corrispondenti alle variabili globali dichiarate all'inizio, **variabili dinamiche**, ovvero le variabili dichiarate con `new` e con puntatori e che devono essere esplicitamente deallocate, e **variabili auomatiche**, locali a una funzione.

4.4 Stringhe, puntatori, tipi

In C, una **stringa** è un vettore di caratteri che termina con uno `\0`.

Un **puntatore** è una variabile che punta a una certa casella di memoria. `int* a` rappresenta un puntatore (non inizializzato) a una variabile di tipo `int`. L'asterisco viene inoltre utilizzato per accedere alla locazione `*a`; `&a` rappresenta invece la locazione di memoria puntata in esadecimale.

Importante è la notazione: infatti il simbolo `*` si lega al nome della variabile e non al tipo, per cui scrivere `int* a, b` definirà una variabile `a` di tipo puntatore a int e `b` di tipo int, e non due puntatori come si potrebbe pensare.

In C è inoltre possibile costruire tipi di dati derivati e rinominarne altri con i costrutti `struct` e `typedef`. La prima permette di creare "tipi aggregati", in maniera analoga a una classe in un linguaggio Object-Oriented; la seconda permette di ridefinire, in maniera analoga a una macro, un certo tipo di dato oppure una struct con un nuovo nome.

5 Processi

Per la teoria riguardante i processi, consultare gli appunti di teoria del corso.

La seguente è una lista, non esaustiva, di comandi utili alla gestione dei processi di sistema Unix tramite Bash.

- `command&` → lancia un processo in background, restituendo il suo PID
- `ps` → elenca i processi attivi su sistema
- `ps -o pid,ppid,pgid,sid,uid,euid` → elenca i processi attivi con informazioni extra

La seguente è una lista degli identificatori di processo:

- `PID` → attributo di identificazione univoco per istante temporale
- `PPID` → id del processo genitore
- `PGID` → id di un gruppo di processi, che possono collaborare in maniera stretta tra loro; al momento della creazione i figli entrano nel gruppo di un genitore
- `SID` → insieme di gruppi di processi; alla creazione un processo è associato alla sessione del padre
- `RUID, EUID` → rispettivamente l'id utente del creatore ed esecutore del processo