

Sistemi Operativi, Secondo Modulo, Canale M–Z

Riassunto della lezione del 27/03/2017

Igor Melatti

La Bash, per davvero

- Pipelining: concatenazione di processi, con collegamento input-output
 - finora, abbiamo visto che si possono dare comandi in sequenza nei seguenti modi: con il `;` o l'andata a capo (equivalenti); con `&&` e il `|`; mettendoli o no dentro parentesi tonde o graffe
 - in tutti questi esempi, quello che succede è semplicemente che prima si esegue un comando, e poi (eventualmente) un altro; input ed output sono scollegati, a meno di redirezioni comuni nel caso di sottoshell o di group command
 - preziosa opportunità un più: collegare gli stdout agli stdin per le sequenze non condizionali (quindi, equivalenti alla separazione con `;` o con l'andata a capo)
 - ovvero, anziché scrivere `cmd1 >file1; cmd2 <file1; rm -f file1`, si scrive `cmd1 | cmd2`
 - lo stdout del processo a sinistra viene rediretto nello stdin del processo a destra
 - scrivendo `|&` anziché `|`, allora anche lo stderr viene rediretto nello stdin
 - se viene mandato in background, con `jobs -l` si vede cosa succede: tutti i processi corrispondenti ai comandi in pipelining sono stati lanciati
 - il pipelining, se applicato ad un group command (comandi tra graffe) o ad una subshell (comandi tra tonde) ha effetto sull'input/output combinato di tutti i comandi del gruppo
 - ad esempio, anziché scrivere `{ cmd1; cmd2; } > out; cmd3 < out; rm out`, si può scrivere `{ cmd1; cmd2; } | cmd3`
 - **esercizio:** senza usare `/dev/null`, né alcun file temporaneo, fare in modo che il comando `ls -l fileesistente filenonesistente` scriva solo le informazioni sul file esistente

- **esercizio:** anziché scrivere `awk '{print}' 0< file 1<> file`, risolvere il problema usando il pipelining ed il comando `tee` (vedere il manuale), assumendo che l'esecuzione dei comandi in pipelining sia da sinistra a destra (nota: in realtà questo non è vero; per essere proprio sicuri usare `sponge` anziché `tee`)
- **esercizio:** senza usare un file temporaneo, far sì che `awk` mostri i soli file della directory attuale che abbiano il permesso di lettura abilitato su tutti e 3 i gruppi, abbiano un nome lungo almeno 10 caratteri ed una dimensione tra i 100 ed i 1000 bytes; il risultato dev'essere la stessa linea ritornata da `ls`
- **esercizio:** senza usare un file temporaneo, far sì che `awk` mostri i soli file in una qualsiasi sottodirectory della directory attuale che abbiano il permesso di lettura abilitato su tutti e 3 i gruppi, abbiano un nome lungo almeno 10 caratteri ed una dimensione tra i 100 ed i 1000 bytes; il risultato dev'essere la stessa linea ritornata da `ls`, ma con il nome sostituito dal path completo
- **esercizio:** senza usare un file temporaneo, far sì che vengano salvate su un file `risultato.txt` tutte le righe del `man bash` che contengono la stessa parola 2 volte, senza usare `awk`
- **esercizio:** senza usare un file temporaneo, far sì che vengano salvate su un file `risultato.txt` tutte le righe del `man bash` che contengono la stessa parola 2 volte, usando `awk`

Script in Bash

- Primi rudimenti di shell scripting
 - prendere un comando qualsiasi (ad esempio, `ls`), e scriverlo su un file di testo; sia `filename` il nome di tale file
 - è possibile eseguire tale file in 4 modi:
 1. `chmod u+x filename; ./filename` (non proprio ortodosso: ci vorrebbe una prima riga, detta *shabang*, fatta in un certo modo; ci ritorneremo)
 2. `bash filename`
 3. `source filename`
 4. `. filename`
 - `filename` è un *bash script* (spesso, si usa l'estensione `.sh` o `.script`)
 - eseguirlo nei primi 2 modi equivale a lanciare una sottoshell (sempre, anche se c'è dentro un solo comando) che esegue uno alla volta i comandi contenuti nello script
 - invece, nei secondi 2 modi *non* si lancia una sottoshell, e l'esecuzione avviene nel contesto della shell corrente

- in realtà, la bash permette di avere un vero e proprio linguaggio di programmazione Turing-completo, i cui comandi base sono, essenzialmente, i comandi della shell visti finora più gli assegnamenti e i controllori di flusso (vedere sotto)
 - quindi, è possibile compiere decisioni (ad esempio, con l'`if`) e cicli (ad esempio, con il `for` e il `while`)
 - quindi, alcuni comandi potrebbero non essere eseguiti, o eseguiti più volte
 - se ci sono errori di sintassi: la parte che precede l'errore viene sempre eseguita
 - la parte che segue l'errore potrebbe essere eseguita o no, a seconda della gravità dell'errore
 - tutto quello che si scrive sulla bash interattiva può essere messo in uno script; al posto del `;`, per separare i comandi, si può usare l'andata a capo
 - il viceversa non è vero, ma solo perché le andate a capo possono essere solo negli script
 - infatti, nella shell interattiva, premere invio vuol dire “esegui il comando”...
 - c'è però l'eccezione già menzionata sopra (se ci sono parentesi o apici aperti)
 - altra eccezione: mettendo un backslash alla fine del comando (ma proprio alla fine, senza spazi successivi), l'effetto è lo stesso di quando ci sono apici o parentesi aperte e non chiuse
- Parametri e variabili
 - per poter essere veramente Turing-completi, è necessario avere anche delle *variabili*, esattamente come nei linguaggi di programmazione
 - le variabili sono definite come nei linguaggi di programmazione: un *identificatore* cui si assegna, e dal quale si può recuperare, un valore
 - niente tipi: di default tutte stringhe, ma possono essere interpretate come interi in opportune circostanze; con opportuna sintassi, possono essere anche degli array (associativi o standard)
 - formalmente, in bash ci sono le variabili (definite come sopra), i *parametri posizionali* e i *parametri speciali*
 - inoltre, sempre formalmente, tutte e 3 le categorie appena definite sono definite genericamente come *parametri*
 - solo le variabili possono essere soggette ad *assegnamento*; per i parametri posizionali, si può usare il comando builtin `set`
 - tutti i parametri possono essere soggetti ad *espansione* (ovvero, si può usare il valore che contengono)

- Variabili

- si assegnano con `identificatore=espressione`
- niente spazi prima e dopo l'uguale!!!!
- si espandono con `$identificatore`, oppure con `${identificatore}`
- quindi, per vedere quanto valgono, è sufficiente dare un comando come `echo ${identificatore}` (il meccanismo che c'è sotto, chiamato *espansione*, sarà più chiaro nel seguito di questa e delle prossime lezioni)
- anche, ma più complicato: `echo niente | awk -v var=${identificatore} 'print var'`
- la versione con le graffe è importante quando ci sono 2 variabili, i cui nomi sono l'uno il prefisso dell'altro
- senza graffe, il nome della variabile finisce non appena non ci sono né caratteri alfanumerici né underscore; altrimenti, il nome della variabile finisce con la `}`
- nel caso delle stringhe (default), si possono concatenare a piacimento stringhe costanti e stringhe variabili, senza usare nessun operatore di concatenazione
- espandere una variabile mai definita prima dà luogo alla stringa vuota (o al valore 0, nelle espansioni aritmetiche)
- dentro alle parentesi graffe si possono in realtà fare molte cose, ma per ora soprassediamo (vedere “Parameter expansion” nel `man bash` e Tabella 4 più avanti)
- per le espressioni aritmetiche, ci ritorneremo

- Variabili locali e d'ambiente

- tutte le variabili assegnate come descritto sopra sono *locali*
- ovvero, un processo lanciato da uno script, dopo che una variabile è stata settata, non può vedere il valore di quella variabile
- fanno eccezione le sottoshell tra parentesi tonde (ovviamente anche i compound group tra graffe, ma lì non viene lanciata nessuna sottoshell), nonché ovviamente gli script lanciati con `source` o `.` (che non creano un nuovo processo)
- se si vuole far sì che un qualsiasi (nuovo) processo lanciato da uno script possa accedere alle variabili v_1, \dots, v_n , precedentemente settate ai valori a_1, \dots, a_n , occorre:
 - * usare il comando `export nomevar` (o anche `declare -x nomevar`) prima di lanciare il processo: `export v_1, \dots, export v_n`
 - * oppure, usare la seguente sintassi: `v_1=a_1 \dots v_n=a_n comando`

Table 1: Alcune variabili d’ambiente predefinite o comunque usate da bash (vedere `man bash` per la lista completa)

Nome	Significato
PATH	Lista di directory, ognuna separata da <code>:</code> . Ogni qualvolta viene dato un comando <code>cmd</code> senza path (ovvero, senza usare neanche uno slash <code>/</code>), la bash cerca in ciascuna di tali directory un file eseguibile di nome <code>cmd</code> . Viene eseguito il file che si trova nella prima directory (da sinistra) dove il file stesso viene trovato. Se <code>cmd</code> non viene trovato, allora l’errore è “Command not found” (exit code 127). Il comando <code>which</code> si limita a trovare tale file, senza eseguirlo; <code>which -a</code> trova tutte le occorrenze.
HOME	Path assoluto della home directory dell’utente attualmente loggato.
IFS	Internal Field Separator, usato dal word splitting. Attenzione: quando si lancia uno script (a meno che non lo si faccia con <code>source</code> o <code>.</code>) questo valore viene sempre resettato al suo valore di default, che è <code>\t\n</code>
BASHPID	Pid della bash corrente.
PS1	Formattatore per il prompt.
PWD	Path assoluto dell’attuale cwd; ogni comando <code>cd</code> ne cambia il valore.
OLDPWD	Path assoluto del cwd precedente quella attuale; ogni comando <code>cd</code> ne cambia il valore.

- * nel primo caso, le variabili v_1, \dots, v_n saranno disponibili a tutti i processi che verranno lanciati in seguito
 - * nel secondo caso, le variabili saranno visibili solo al processo creato da `comando`
 - * in ogni caso, eventuali modifiche non avranno effetto sulla bash padre
- il comando `export` ha proprio l’effetto di trasformare una variabile locale in variabile d’ambiente
 - il comando `unset nomevar` annulla una `export nomevar`
 - esempio: `bash lancia_var_ambiente.sh`
 - inizialmente, ci sono svariate variabili predefinite nell’ambiente (e altre se ne possono aggiungere mettendole nei file di configurazione); le più importanti sono descritte in Tabella 1
 - **esercizio:** scrivere uno script che modifichi la variabile d’ambiente `PATH`, cancellando il precedente contenuto e facendo sì che contenga solo la directory corrente, e poi ne stampi il valore. Verificare che, lanciandolo come nuovo processo, la modifica avviene effettivamente all’interno dello script, ma nella bash di partenza (quella che invoca lo

script) la variabile `PATH` non è stata modificata. Verificare anche che, lanciandolo all'interno dello stesso processo della `bash`, la modifica risulta permanere anche nella `bash` invocante. Modificare lo script in modo tale che, dopo la modifica, la variabile `PATH` venga ripristinata al suo valore precedente.

- Parametri posizionali
 - si vedono solo negli script, o nei corpi delle funzioni
 - vengono espansi con il carattere `$` seguito da un numero intero positivo
 - `$n` sta per l'`n`-esimo argomento dato allo script o alla funzione
 - se `n` necessita più di una cifra decimale per essere scritto, allora l'espansione va fatta con le parentesi graffe: `${n}` (altrimenti...)
 - con `set {valore}` si possono cambiare i valori di questi parametri: se vengono specificati `n` valori, allora il primo viene assegnato a `$1`, il secondo a `$2`, ..., l'`n`-esimo a `$n`
 - con `shift [n]` si possono cambiare i valori di questi parametri: quelli da `$1` a `$n` avranno come valore la stringa vuota (quindi, saranno non assegnati), mentre `$(n+1)` avrà il vecchio valore di `$1`, `$(n+2)` avrà il vecchio valore di `$2` e così via fino all'ultimo. L'argomento di default di `shift` è 1.
 - nelle chiamate a funzione, i valori dei parametri posizionali vengono modificati solo temporaneamente, in modo da avere i valori passati alla chiamata stessa
 - una volta finita la chiamata, riprendono i valori precedenti (ci ritorneremo)
 - **esercizio:** scrivere uno script che stampi il valore dei suoi argomenti, usando solo `$1` come parametro posizionale. Assumere che verranno sempre passati esattamente 5 argomenti.
 - **esercizio:** scrivere uno script che stampi il valore di alcuni suoi argomenti, usando solo `$1` come parametro posizionale. Quali argomenti stampare è deciso dal primo argomento: assumendo che sia un intero positivo `n`, occorrerà stampare gli argomenti in posizione `1 + in`, con $i \in \{1, 2, 3, 4, 5\}$.
- Parametri speciali, ci limitiamo ai seguenti
 - `$0`: il nome dello script, come è stato avviato (o quasi, rivedere Tabella 1 della lezione 8)
 - `$*`: lista di tutti i parametri posizionali (da 1)
 - `@`: lista di tutti i parametri posizionali (da 1); la differenza con il precedente la fa il word splitting (ci ritorneremo)

- `$#` : numero di parametri posizionali (da 1), separati da uno spazio
- `$?` : exit status dell'ultimo processo terminato
- `$!` : pid dell'ultimo processo avviato in background (terminato o no)
- `$$` : pid della bash (o del parent della bash, se si tratta di una sottoshell)
- **esercizio:** scrivere uno script che stampi il valore del suo ultimo argomento.
- **esercizio:** scrivere 2 script `a.sh` e `b.sh` , con `a.sh` che lancia `b.sh` . Fare in modo che `a.sh` stampi il valore del suo i -esimo argomento (che esista o no), dove i è l'exit status di `b.sh`