Sistemi Operativi, Secondo Modulo, Canale M–Z
Riassunto della lezione del 23/03/2017

Igor Melatti

La Bash, per davvero

- Finora, uso molto limitato della Bash
  - solo comandi singoli (in foreground o in background), e poi invio
  - l'output quasi sempre su schermo, tranne che per i comandi che permettono di specificare un file dove mettere l'output
    - per esempio `patch`...
  - l'input quasi sempre da file, oppure da tastiera quando i file non sono specificati
- Tutte e tre queste caratteristiche possono essere modificate
  - si possono specificare sequenze di comandi
  - si possono specificare varie condizioni in dipendenza delle quali eseguire un comando oppure un altro
  - se l'output è su schermo, si può dire di scriverlo su un file
  - se l'input è da tastiera, si può dire di prenderlo da un file
  - si può far sì che l'input di un comando sia l'output di un altro
- 3 diversi tipi di bash (e in generale di shell):

  login shell è una shell interattiva per accedere alla quale occorre fornire username e password (ad esempio, quelle accessibili tramite CTRL+ALT+Fn, con $n \in \{1, \ldots, 6\}$). Fanno eccezione le shell aperte con `bash -l` o `bash --login`: non chiedono l'autenticazione, ma sono di login.

  interactive shell è una shell interattiva per accedere alla quale non occorre fornire username e password

  non-interactive shell è una (sotto)shell invocata per eseguire uno script (che potrebbe contenere comandi interattivi...)
- *Configurazione* della bash

Table 1: Tipi di bash e file di configurazione. Nel caso di "Extended", i seguenti file vengono usati: `/etc/profile`, poi il primo che esiste ed è accessibile in lettura tra `~/.bash_profile`, `~/.bash_login`, `~/.profile`; al logout, viene eseguito `~/.bash_logout`. Nel caso di "Restricted", i seguenti file vengono usati: `/etc/bash.bashrc`, poi `~/.bashrc`. Nel caso "Nothing", nessun file di configurazione viene usato.

Tipo	**Invocazione**	**echo $0**	**Configurazione**
login senza autenticazione	`bash -l`	`bash`	Extended
	`bash --login`		
	`(exec -a "-bash" bash)`	`-bash`	
	`(exec -l bash)`		
login con autenticazione	CTRL+ALT+Fn		
	`ssh utente@nomemacchina`		Extended o Restricted
	`ssh utente@localhost`		
	`su - utente`	`-su`	Extended
	`su -l utente`		
interactive	`bash [-i]`	`bash`	Restricted
non-interactive	`bash nomefile`	`nomefile`	Nothing
	`bash -l nomefile`		Extended
	`bash --login nomefile`		
sottoshell		uguale shell padre	Nothing

  - *system-wide*, scelta dall'amministratore del sistema e che si applica ad ogni utente; basata sui files `/etc/profile` e `/etc/bash.bashrc`
  - una configurazione definibile dall'utente, che può anche sovrascrivere alcune impostazioni della configurazione system-wide; basata sui file `~/.bash_profile`, `~/.bash_login`, `~/.profile`, `~/.bashrc`
  - una configurazione di uscita può essere scritta su `~/.bash_logout`

- La situazione è un po' caotica; la Tabella 1 dovrebbe fare un po' di chiarezza

- Un po' di storia delle (principali!) shell:

  - `sh`, detta *Bourne Shell*, dal nome del ricercatore che la ideò, nel 1977 ai Bell Labs. Le shell che si ispirano ad essa hanno il prompt che termina in `$`
  - `csh`, detta *C Shell*, ideata nel 1978 da Joy a Berkeley per la BSD. Oggi la si usa come `tcsh`. Le shell che si ispirano ad essa hanno il prompt che termina in `%`
  - `bash`, detta *Bourne Again Shell*, `sh` reimplementata, e migliorata, per GNU (Fox, 1989). Come la `sh`, ma con le caratteristiche interattive (ad es., la history) della `csh`

- Comandi della shell
  - ogni comando viene dato immettendo da tastiera una serie di parole, separate da spazi
  - la prima è il comando (da cercare nel filesystem, o interni alla bash), poi seguono opzioni ed argomenti
  - per alcuni comandi, opzioni ed argomenti possono essere mischiati (ad es. `ls`)
  - per altri (la maggioranza), prima le opzioni, poi gli argomenti (ad es. `find`)
  - un comando viene considerato completato:
    - quando si trova un `;` (esecuzione in foreground)
      - il `;` può anche essere usato per separare comandi: `c1; c2` significa che prima deve essere eseguito `c1`, e una volta che questo termina va eseguito `c2`
    - quando si trova un `&` (esecuzione in background)
    - quando viene premuto invio, il che fa partire l'esecuzione del comando (o dei comandi)
      - questo tuttavia vale solo per comandi in cui non ci sono parentesi o apici aperti ma non chiusi, nel qual caso il prompt cambia (diventa `>`) e occorre immettere la conclusione della riga
  - qualsiasi comando può essere inserito tra parentesi tonde. Il significato è che quel comando non va eseguito dal processo corrispondente alla bash corrente; bensì, verrà lanciato un nuovo processo bash (comunemente detto *sottoshell*), all'interno del quale viene eseguito quel comando
    - se il comando è unico, allora il nuovo processo è costituito da quel comando
    - altrimenti, se è una lista di comandi (separati, ad esempio, da `;`), allora il nuovo processo è una bash che esegue quei comandi
    - utilità: poter raggruppare tutte le redirezioni in una volta sola (vedere sotto)
    - utilità: poter raggruppare tutto l'input/output da mandare in pipelining in una volta sola (vedere sotto)
    - utilità: poter raggruppare più comandi in esecuzione condizionale (vedere sotto), facendo sì che non ci sia effetto sulla bash corrente
  - qualsiasi comando può essere inserito tra parentesi graffe (*group command*). Il significato è che quel comando va eseguito dal processo corrispondente alla bash corrente
    - se il comando è unico, mettere le parentesi graffe è inutile (anzi, complica le cose, vedere sotto)

    - altrimenti, l'utilità sta nel fatto di poter raggruppare tutte le redirezioni in una volta sola (vedere sotto)
    - notare che serve lo spazio dopo la prima parentesi graffa, e che serve il `;` dopo l'ultimo comando
    - altra utilità: poter raggruppare tutto l'input/output da mandare in pipelining in una volta sola (vedere sotto)
    - altra utilità: raggruppare più comandi in esecuzione condizionale (vedere sotto), facendo sì che l'effetto sia sulla bash corrente
  - provare a dare i comandi `(cd ..)` e { `cd ..;` }

- Ogni comando genera un processo. Tale processo, terminando, restituisce un *exit code* alla bash: storicamente, 0 indica *tutto ok*, mentre un valore tra 1 e 255 indica un errore
  - se va bene, va bene
  - se va male, ci possono essere molte cause (ma non più di 255...)
  - se un comando non è riconosciuto, viene restituito 127
  - se un comando è costituito da una sequenza di comandi separati da `;`, allora l'exit code è quello dell'ultimo comando eseguito
  - se un comando viene eseguito in background, l'exit code è 0; per prendere il suo vero exit code, occorre usare il comando builtin `wait`

- La bash permette l'esecuzione *condizionale* dei comandi
  - un comando viene eseguito solo se una certa condizione è vera
  - molti modi per farlo; il più semplice è condizionare un comando alla corretta (o sbagliata) esecuzione di un comando precedente
  - dove "corretta" equivale a "l'exit code è 0" e "sbagliata" equivale a "l'exit code è diverso da 0"
  - sintassi:
    - `comando1 && comando2`: `comando2` viene eseguito solo se `comando1` è corretto
    - `comando1 || comando2`: `comando2` viene eseguito solo se `comando1` è sbagliato
    - si possono concatenare più di 2 comandi, e anche usare le parentesi, sia tonde che graffe (vedere più sotto per la differenza tra queste 2 opzioni)
  - **esercizio:** capire se l'exit code di `ls` è 0 o diverso da 0 nei seguenti casi:
    - nessun argomento
    - un file esistente come argomento
    - un file esistente ma non accessibile in lettura come argomento

    - una directory esistente ma non accessibile in lettura come argomento
    - un file non esistente come argomento
    - un file esistente e uno non esistente come argomento
  - **esercizio:** capire se l'exit code di `find` è 0 o diverso da 0 nei seguenti casi:
    - le opzioni non sono corrette (ad esempio: `-name file1 file2`)
    - non trova nessun file
    - trova 1 file
    - trova più di un file
  - **esercizio:** verificare che il comando `exit 12;` di `awk` fa sì che l'exit code di awk sia diverso da 0

- Ogni comando genera un processo cui vengono subito associati 3 *stream*:
  - *standard input* o `stdin`, con *file descriptor* 0 (di default: la tastiera)
  - *standard output* o `stdout`, con *file descriptor* 1 (di default: lo schermo)
  - *standard error* o `stderr`, con *file descriptor* 2 (di default: lo schermo)
  - un file descriptor è un intero intero non negativo associato o ad uno stream (come sopra) o ad un file vero e proprio
  - agendo su tale numero, è come se si agisse sullo stream e/o file corrispondente
  - comando `lsof +f g -ap pid`: lista dei file aperti dal processo `pid`
  - settando `pid` al PID della bash (si può usare la variabile `BASHPID`), si vede che tutti e 3 gli stream sono collegati ad un file speciale: `/dev/pts/n`, se l'attuale bash è stata l'`n`-esima ad essere aperta
  - le sottoshell hanno gli stessi file descriptor della shell genitrice...
  - vedere anche il contenuto di `/dev/fd/`, e vederlo da bash diverse...

- Ciascuno degli stream dati sopra può essere *rediretto*. Le regole generali per la *redirection* "dirette" sono riportate in Tabella 2. Tenere anche presente la Figura 1
  - le redirezioni avvengono sempre *prima* che il comando sia eseguito
  - supponendo che un file di nome `file` sia non vuoto, provare a dare il comando `awk '{print}' < file > file`: dopo, `file` sarà vuoto
  - perché >, come prima cosa, tronca il file (è come aprire un file in scrittura)
  - le redirezioni, se applicate ad un group command (comandi tra graffe) o ad una subshell (comandi tra tonde) hanno effetto su tutti i comandi del gruppo

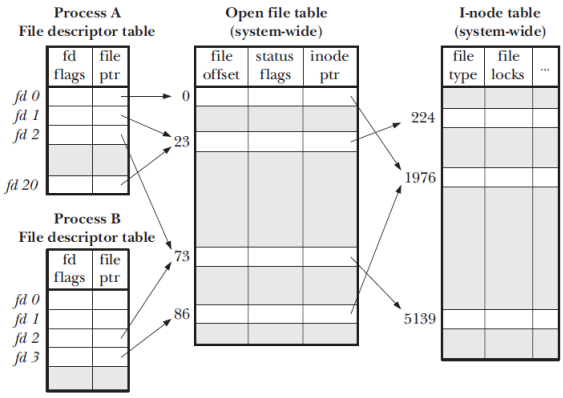

Figure 1: Organizzazione interna di file descriptors e i-nodes

Table 2: Regole per la redirection. Il numero `n` dev'essere un file descriptor. Le redirection possono trovarsi in *qualsiasi punto* di un comando (anche prima del comando stesso).

Operatore	**Azione**
`[n]>nomefile`	Redirige il file con descriptor `n` sul file `nomefile`. Se non esiste, lo crea, se esiste, lo sovrascrive
`[n]>>nomefile`	Come sopra, ma appende il contenuto in coda al file `nomefile` creandolo se non esiste. Default `n`=1 (stdout)
`[n]<nomefile`	Redirige il contenuto del file `nomefile` sul file descriptor `n`. Default `n`=0 (stdin)
`[n]<>nomefile`	Può essere usato sia per ridirigere il contenuto del file `nomefile` sul file descriptor `n` (redirezione di input), che per il contrario (redirezione di output). Se usato per ridirigere l'output su un file esistente, ne sovrascrive solo i primi bytes, lasciando inalterati gli altri. Di default, `n`=0 (stdin, quindi redirezione di input).
`&>nomefile`	Redirige lo stdout e stderr sul file `nomefile`
`>&nomefile`	(come sopra)
`&>>nomefile`	Come sopra, ma appende stdout e stderr alla fine del file `nomefile`

- ad esempio, anziché scrivere `cmd1 > out; cmd2 >> out`, si può scrivere { `cmd1; cmd2;` } `> out`
- **esercizio:** tenendo conto che esiste un file speciale `/dev/null` all'interno del quale si può riversare qualsiasi output, senza che la dimensione di questo file cresca, fare in modo che il comando `ls -l fileesistente filenonesistente` scriva solo le informazioni sul file esistente
- **esercizio:** scrivere `awk '{print}' 0< file 1<> file` risolve il problema del troncamento di `file`? Se sì, funziona con qualsiasi programma venga fornito ad `awk`?
- **esercizio:** usando 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:** usando 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:** usando 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:** usando 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`

- I file descriptor possono essere *creati*, *duplicati* e *chiusi*, secondo le regole di Tabella 3

Table 3: Regole per i file descriptor. Possono essere combinate con quelle in Tabella 2; in tal caso, vanno valutate da sinistra a destra.

Operatore	**Significato**	**Commento**
`n< file`	Apertura in lettura	Se usato in questo modo: `exec n< file`, apre in lettura il file con nome `file` sul file descriptor `n`. Default `n`=0 (stdin). Dopo tale comando, si può usare `n` nelle redirezioni di lettura, e l'effetto sarà quello di leggere da `file` (finché `n` non viene chiuso).

Continuazione di Tabella 3.

Operatore	**Significato**	**Commento**
`n> file`	Apertura in scrittura	Se usato in questo modo: `exec n> file`, apre in scrittura il file con nome `file` sul file descriptor `n`. Se il file esiste, come prima cosa viene troncato a dimensione 0. Default `n`=1 (stdout). Dopo tale comando, si può usare `n` nelle redirezioni di scrittura, e l'effetto sarà quello di scrivere su `file` (finché `n` non viene chiuso).
`n<> file`	Apertura in lettura e scrittura	Se usato in questo modo: `exec n<> file`, apre in lettura e in scrittura il file con nome `file` sul file descriptor `n`. Default `n`=0 (stdin). Dopo tale comando, si può usare `n` nelle redirezioni sia di lettura che di scrittura, e l'effetto sarà quello di leggere e/o scrivere su `file` (finché `n` non viene chiuso).
`n <& m`	Duplicazione in lettura	Duplica il file descriptor `m` in `n`. L'effetto sarà quello che letture chieste al file descriptor `n` verranno invece fatte dal file descriptor `m`. Il default per `n` è 0 (stdin)
`n >& m`	Duplicazione in scrittura	Duplica il file descriptor `n` in `m`. L'effetto sarà quello che scritture effettuate sul file descriptor `n` verranno invece fatte sul file descriptor `m`. Il default per `n` è 1 (stdout)
`n <&-`	Chiusura in lettura	Chiude il file descriptor `n`: non potrà più essere usato (ovvero, l'input non verrà più rediretto al file o al device collegato in precedenza ad `n`). Il default per `n` è 0 (stdin). Messo all'interno di un comando qualsiasi, ha effetto solo su quel comando; per chiudere definitivamente il file descriptor `n`, occorrerà il comando `exec n <& -`
`n >&-`	Chiusura in scrittura	Chiude il file descriptor `n`: non potrà più essere usato (ovvero, l'output non verrà più rediretto al file o al device collegato in precedenza ad `n`). Il default per `n` è 1 (stdout). Come sopra per la chiusura definitiva.
`n <& m-`	Spostamento in lettura	Combinazione di duplicazione in lettura seguita da chiusura di `m`. Come sopra per la chiusura definitiva.
`n >& m-`	Spostamento in scrittura	Combinazione di duplicazione in scrittura seguita da chiusura di `m`. Come sopra per la chiusura definitiva.

- caso più comune: redirigere stderr in stdout, ovvero `2>&1`
- Chiarimento di cosa vuol dire "copiare" un file descriptor: con riferimento alla Figura 1, un comando del tipo `2>&1` prende il "file ptr" relativo al file descriptor 2 e lo copia nel "file ptr" relativo al file descriptor 1 (con ciò sovrascrivendo il valore precedente)
- **esercizio:** perché il comando `ls nonesiste 2>&1 > /dev/null` mostra comunque il messaggio di errore su schermo (considerare la didascalia di Tabella 3 e quanto appena detto nel punto precedente)? Come occorre correggere (senza usare `>&` o `&>`), se si vuole che non venga mostrato alcun messaggio d'errore?
- **esercizio:** perché il comando `ls nonesiste 2>&1 > /dev/null` mostra comunque il messaggio di errore su schermo (considerare la didascalia di Tabella 3)? Come occorre correggere (senza usare `>&` o `&>`), se si vuole che non venga mostrato alcun messaggio d'errore?
- **esercizio:** usando un file temporaneo, ma usando solo creazioni e duplicazioni di file descriptor (quindi, usando solo operatori presi dalla Tabella 3), 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:** usando un file temporaneo, ma usando solo creazioni e duplicazioni di file descriptor (quindi, usando solo operatori presi dalla Tabella 3), 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:** usando un file temporaneo, ma usando solo creazioni e duplicazioni di file descriptor (quindi, usando solo operatori presi dalla Tabella 3), 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:** usando un file temporaneo, ma usando solo creazioni e duplicazioni di file descriptor (quindi, usando solo operatori presi dalla Tabella 3), 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`