

Sistemi Operativi, Secondo Modulo, Canale A–L
e Teledidattica
Riassunto della lezione del 17/03/2021

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, anche organizzando i comandi in cicli
 - 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 $n \in \{1, \dots, 6\}$). 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 (tranne i 2 casi appena detti)
 - non-interactive shell* è una (sotto)shell invocata per eseguire uno script

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	<code>bash -l</code>	<code>bash</code>	Extended
	<code>bash --login</code>	<code>-bash</code>	
	<code>(exec -a "-bash" bash)</code>		
	<code>(exec -l bash)</code>		
login con autenticazione	<code>CTRL+ALT+Fn</code>	<code>-su</code>	Extended o Restricted
	<code>ssh utente@nomemacchina</code>		
	<code>ssh utente@localhost</code>		
	<code>su - utente</code>	Extended	
	<code>su -l utente</code>		
interactive	<code>bash [-i]</code> o invocazione terminale	<code>bash</code>	Restricted
non-interactive	<code>bash nomefile</code>	<code>nomefile</code>	Nothing
	<code>bash -l nomefile</code>		Extended
	<code>bash --login nomefile</code>		
sottoshell		uguale shell padre	Nothing

- *Configurazione della bash*
 - *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` (si ricorda che `~` è la directory home dell’utente attualmente loggato)
 - 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**
- Le shell bash vengono aperte a richiesta dell'utente
 - alcune in modo “automatico”: se si apre un terminale grafico (come l'**LXTerminal** della macchina virtuale dei laboratori, o il **gnome-terminal** standard di Ubuntu), il processo che gestisce tale terminale crea anche una bash con il comando **bash -i** (vedere Tabella 1)
 - altre in modo esplicito: o perché si preme CTRL+ALT+Fn, o perché da dentro una shell si esegue uno dei comandi della seconda colonna di Tabella 1
- Le shell bash vengono chiuse a richiesta dell'utente o per necessità di sistema
 - richiesta utente: basta usare il comando **exit [n]**
 - richiesta utente: basta usare il comando **logout**, ma solo se è una shell di login
 - necessità di sistema: una bash che cerca di fare qualche operazione proibita grave può essere terminata dal sistema operativo
 - necessità di sistema: in caso di bash in esecuzione remota (ad es., **ssh**), la bash può essere terminata se cade la connessione, o se l'utente supera la quota consentita di utilizzo
- 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, a meno che non sia interno alla bash), poi seguono opzioni ed argomenti
 - per alcuni comandi, opzioni ed argomenti possono essere mischiati (ad es. **ls**)
 - per altri, prima le opzioni, poi gli argomenti (ad es. **find**)
 - un comando viene considerato completato:
 - * quando si trova un **;** o un'andata a capo prima di 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 **&** prima di un **;** o un'andata a capo (esecuzione in background)

- * per le shell interattive, 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 parentesi graffa aperta, il ; dopo l'ultimo comando, e lo spazio prima della parentesi graffa chiusa
 - * 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, che non sia built-in, 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` (es., `wait -f %n` per il job `n`)
- 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
 - **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*:

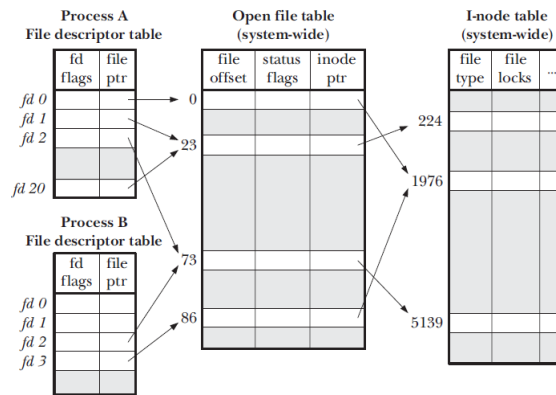


Figure 1: Organizzazione interna di file descriptors e i-nodes

- *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 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 `lsuf +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
 - * in realtà, vale lo stesso discorso fatto per i PID: se una bash viene chiusa, il suo indice viene "liberato" e potrà essere riusato per la prossima bash che verrà aperta
 - le sottoshell hanno gli stessi file descriptor della shell genitrice (però non leggono l'input della bash da `stdin`, come fa invece la shell aperta su un terminale: del resto, sono non-interattive)
 - vedere anche il contenuto di `/dev/fd/`, e vederlo da bash diverse...
- Ciascuno degli stream dati sopra può essere *rediretto*. Le regole generali per le *redirection* 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
 - 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 2

Table 2: Regole per le redirezioni. Possono essere combinate tra loro; in tal caso, vanno valutate da sinistra a destra. **Importante:** se messe dentro un comando, hanno effetto solo sul comando stesso; affinché abbiano effetto per tutti i comandi da un certo punto in poi, occorre usare il comando built-in `exec` (ad es.: `exec <file` fa sì che da questo punto in poi lo standard input diventi il contenuto del file `file`)

Operatore	Significato	Commento
<code>n< file</code>	Apertura in lettura	Apre in lettura il file con nome <code>file</code> sul file descriptor <code>n</code> . Ovvero, dopo tale redirezione, si può usare <code>n</code> nelle redirezioni di lettura, e l'effetto sarà quello di leggere da <code>file</code> (finché <code>n</code> non viene chiuso). Inoltre, se il file descriptor <code>n</code> era già aperto, tutte le letture fatte su <code>n</code> vengono fatte su <code>file</code> (a meno di sovrascrizioni). Il default per <code>n</code> è 0 (stdin).
<code>n> file</code>	Apertura in scrittura	Apre in scrittura il file con nome <code>file</code> sul file descriptor <code>n</code> . Se il file esiste, come prima cosa viene troncato a dimensione 0. Dopo tale redirezione, si può usare <code>n</code> nelle redirezioni di scrittura, e l'effetto sarà quello di scrivere su <code>file</code> (finché <code>n</code> non viene chiuso). Inoltre, se il file descriptor <code>n</code> era già aperto, tutte le scritture fatte su <code>n</code> vengono fatte su <code>file</code> . Il default per <code>n</code> è 1 (stdout).
<code>n>> file</code>	Apertura in scrittura (append)	Apre in scrittura il file con nome <code>file</code> sul file descriptor <code>n</code> . Se il file esiste, sposta il suo offset alla fine del file. Dopo tale redirezione, si può usare <code>n</code> nelle redirezioni di scrittura, e l'effetto sarà quello di scrivere a partire dalla fine di <code>file</code> (finché <code>n</code> non viene chiuso). Inoltre, se il file descriptor <code>n</code> era già aperto, tutte le scritture fatte su <code>n</code> vengono fatte alla fine di <code>file</code> . Il default per <code>n</code> è 1 (stdout).
<code>&> file</code>	Redirezione in scrittura	Redirige sia lo stdout che lo stderr.
<code>>& file</code>	Redirezione in scrittura	Redirige sia lo stdout che lo stderr.
<code>&>> file</code>	Redirezione in scrittura	Redirige sia lo stdout che lo stderr, ma anziché sovrascrivere <code>file</code> , appende alla fine di esso.

Continuazione di Tabella 2.

Operatore	Significato	Commento
<code>n<> file</code>	Apertura in lettura e scrittura	Apri in lettura e in scrittura il file con nome <code>file</code> sul file descriptor <code>n</code> . Dopo tale redirectione, si può usare <code>n</code> nelle redirectioni sia di lettura che di scrittura, e l'effetto sarà quello di leggere e/o scrivere su <code>file</code> (finché <code>n</code> non viene chiuso). Inoltre, se il file descriptor <code>n</code> era già aperto, tutte le operazioni fatte su <code>n</code> vengono fatte su <code>file</code> . Da notare che, in caso di operazione di scrittura, il file non viene troncato, ma vengono sovrascritti solo i caratteri interessati. Il default per <code>n</code> è 0 (stdin).
<code>n <& m</code>	Duplicazione in lettura	Duplica il file descriptor <code>m</code> in <code>n</code> . Dopo tale redirectione, l'effetto sarà quello che letture chieste al file descriptor <code>n</code> verranno invece fatte dal file descriptor <code>m</code> . Il default per <code>n</code> è 0 (stdin)
<code>n >& m</code>	Duplicazione in scrittura	Duplica il file descriptor <code>m</code> in <code>n</code> . Dopo tale redirectione, l'effetto sarà quello che scritture effettuate sul file descriptor <code>n</code> verranno invece fatte sul file descriptor <code>m</code> . Il default per <code>n</code> è 1 (stdout)
<code>n <&-</code>	Chiusura in lettura	Chiude il file descriptor <code>n</code> : non potrà più essere usato (ovvero, l'input non verrà più rediretto al file o al device collegato in precedenza ad <code>n</code>). Il default per <code>n</code> è 0 (stdin).
<code>n >&-</code>	Chiusura in scrittura	Chiude il file descriptor <code>n</code> : non potrà più essere usato (ovvero, l'output non verrà più rediretto al file o al device collegato in precedenza ad <code>n</code>). Il default per <code>n</code> è 1 (stdout).
<code>n <& m-</code>	Spostamento in lettura	Combinazione di duplicazione in lettura seguita da chiusura di <code>m</code> .
<code>n >& m-</code>	Spostamento in scrittura	Combinazione di duplicazione in scrittura seguita da chiusura di <code>m</code> .

- 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 1 e lo copia nel “file ptr” relativo al file descriptor 2 (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 2 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:** usando un file temporaneo, ma usando solo creazioni e duplicazioni di file descriptor (quindi, usando solo operatori presi dalla Tabella 2), 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 2), 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 2), 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 2), 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`
- **esercizio:** creare un file di testo `file.txt` con un contenuto qualsiasi (ma non vuoto), e provare a dare il seguente comando: `1<>file.txt awk '{print}' 0<&1`. Dopodiché, vedere nuovamente il contenuto di `file.txt`: cos'è successo? Provare poi con il seguente comando: `1<>file.txt echo ciao: adesso cos'è successo?`