BASH - Esame Sistemi Operati: Vittorio Ghini

La shell legge le righe scritte dall'utente e partendo a leggere i vari caratteri speciali espande in quest'ordine:

1. History Expansion
2. Brace Expansion
3. Tilde Expansion
4. Parameter and Variable Expansion
5. Arithmetic Expansion
6. Command substitution
7. Word Splitting
8. Pathname Expansion
9. Quote Removal

COMANDI SACRI

man: Permette di visualizzare tutte le informazioni relative al comando passato (man [comando])

--help: È un'opzione che hanno solo i comandi bash built-in e contiene una sorta di "riassunto breve" di cosa fanno quest'ultimi (comando --help)

man bash: È il più importante di tutti in quanto da informazioni su come funziona la bash in generale

- Utilizzando il simbolo **/** è possibile scrivere subito dopo la parola o la frase da cercare, così da ottenere tutte le informazioni dettagliate riguardo a un comando o al funzionamento di una particolare funzione all'interno della bash. (Esempio: una delle espansioni oppure le espressioni condizionali).

COMANDI BASE

echo: Stampa a video nel terminale la sequenza di caratteri passata fino all'invio/return

- Se si vuole andare a capo si può utilizzare:
  - **“** all'inizio e poi quando si termina si mette **“**
  - In una frase già iniziata invece metto **** prima di premere invio
- Mettere una frase tra **“”** o **‘’** permette di stampare anche caratteri speciali
- Mettere **** prima di un carattere considerato speciale (come **“** o **‘**) ne abilità la stampa disattivando la sua interpretazione da parte della bash
- **-e**: Permette di abilitare l'interpretazione dei \
  - Esempio: echo a\tb → Stampa “a b”
- **-n**: Permette di disabilitare la stampa su una nuova riga ogni volta
  - Esempio: [cicloqualunque]; do echo -n [indiceACrescereFinoa5] → Stampa 12345

cd: Cambia la directory

- Se lo si lancia senza parametri allora si tornerà alla directory home dell'utente corrente
- Se lo si lancia con un percorso (cd [percorso]) allora ci si sposterà in quella directory
- Se lo si lancia con un **.** (cd **.**) allora si rimarrà nella directory corrente
- Se lo si lanca con doppio punto **..** (cd **..**) allora si tornerà indietro alla directory padre, se è presente, della directory in cui ci si trova.

pwd: Stampa a video la directory corrente, ossia quella in cui ci si trova.

ls: Stampa a video il contenuto della directory corrente

- **-a** / **-all**: Mostra anche i file nascosti, ossia quelli che iniziano con **.**
- **-A**: Il funzionamento è uguale ad **-a** ma non include le directory **.** e **..**
- **-d**: Solitamente utilizzata insieme ad altre opzioni, mostra la directory corrente ma non il suo contenuto
- **-l**: Mostra i file e directory della directory corrente, compresi delle loro proprietà (lettura,scrittura, group...)

find: Serve a trovare qualcosa a partire da certe opzioni passate (percorso di partenza, parte del nome…). Esplora le cartelle in maniera ricorsiva.

- **-name**: Permette di cercare file o directory e trovarne il percorso
  - find [percorsoDiPartenza] -name [nomeDaCercare]

- **-iname**: Permette di cercare file o directory, ignorando il confronto tra maiuscole e minuscole, e trovarne il percorso
    - find [percorsoDiPartenza] -iname [nomeDaCercare]
- **-type**: Permette di cercare uno specifico tipo di cosa con il nome passato
    - find [percorsoDiPartenza] -iname/name [nomeDaCercare] -type **f** → cerca un file
    - find [percorsoDiPartenza] -iname/name [nomeDaCercare] -type **d** → cerca una directory
- **-maxdepth**: Permette di non esplorare ricorsivamente tutte le sottodirectory durante la ricerca, ma si ferma al sotto livello specificato con il parametro [n]
    - Con 1 è come praticamente un ls
    - find [percorsoDiPartenza] -maxdepth [n] -iname/name [nomeDaCercare]
- **-mindepth**: Restituisce solo i file che corrispondono alla ricerca ed appartengono a sottolivelli uguali o superiori a quello specificato dal parametro [n]
    - Con 1 è come praticamente un ls
    - find [percorsoDiPartenza] -mindepth [n] -iname/name [nomeDaCercare]
- **-exec**: Permette di applicare un comando a tutti i file che corrispondono alla ricerca eseguita
    - find [percorsoDiPartenza] [maxdepth/mindepth] -iname/name [nomeDaCercare] [opzioni] -exec [comando] '{}' \;
        - Al posto di '{}' verrà messo, ogni volta, il percorso di un file che corrisponde alla ricerca
        - -exec accetta solo o singoli comandi oppure uno script

COMMAND LIST

Cosa sono? È una sequenza di comandi separati da un delimitatore, che permette di eseguire più comandi in un'unica riga eseguiti in maniera consecutiva.

Separatore **;**

- Separa i comandi in una lista e li esegue uno dopo l'altro **senza condizione**, ed ogni comando viene eseguito indipendentemente dal successo o dal fallimento degli altri.

- Se viene fatto un ridirezionamento output allora in questo caso, tale ridirezionamento verrà fatto solo per l'ultimo comando
- Esempio: echo "ciao" **;** ls **;** pwd

Separatore **( )**

- Vengono utilizzate per **raggruppare più comandi** in un unico sotto-processo. Tutti i comandi all'interno delle parentesi vengono eseguiti **in un nuovo processo**.
- Se viene fatto un ridirezionamento output allora in questo caso, tale ridirezionamento verrà fatto per tutti i comandi, uno per volta.
    - ([comando1] ; [comando2]) | [comando3] -> Il [comando3] viene prima applicato a ciò che genera [comando1] e poi a ciò che genera [comando2]
- [pwd ; **(** cd / ; pwd **)** ; pwd]
    - Se la directory in cui ci si trova è /BUTTAMI allora dopo questo comando si sarà ancora nella stessa directory e perciò l'output di questo comando sarà: /BUTTAMI - / - /BUTTAMI

UTENTI E GRUPPI

Ogni User ha univoci:

- Username
- User ID

Ogni group ha univoci:

- Groupname
- Group ID

Un utente può far parte di più gruppi.

Ogni file e directory ha un proprietario e per questo è anche associato al gruppo di quest'ultimo.

- Ha dei permessi associati per:
    - User (Proprietario)
    - Group
    - Others

I vari permessi vengono impostati come:

- r - 4 : Per la lettura

- w - 2 : Per la scrittura
- x - 1 : Per l'esecuzione
- s: Se presente vuol dire che il file può essere eseguito con i permessi di root tramite → sudo [fileDaEseguire]
    - Se invece è presente S invece indica che c'è un problema nell'assegnazione dei permessi.

COMANDI PER I PERMESSI

chown: Permette al proprietario attuale del file di cambiare il proprietario

- chown [nuovoProprietario] [file]

chgrp: Permette di cambiare il gruppo al file attuale

chmod: Permette di cambiare i permessi di un file

- chmod **+x** [file] → Aggiunge all'utente corrente il permesso di esecuzione del file
- chmod +n (o +nnn) [file] → A seconda degli n passati cambia i vari permessi
    - 7 → Tutti
    - <7 → Dipende e toglie gli altri

COMANDI PER CREAZIONE E RIMOZIONE FILE E DIRECTORY

touch: Permette di creare dei file, anche più contemporaneamente inserendo uno spazio tra un nome e l'altro

- Se si mette un **.** prima del [nomeFile] allora quel nuovo file sarà nascosto
- Se il file non esiste lo crea, altrimenti se esiste ne aggiorna la data di ultima modifica.

nano: Apre un editor di testo

- Lanciato da solo crea un nuovo file
- Lanciato con un nome di file esistente permette di modificarne il contenuto se si hanno i permessi adatti
- Se si scrive uno script aggiungere come prima riga: #!/bin/bash

mkdir: Permette di creare una directory

rmdir: Permette di rimuovere una directory

rm: Permette di rimuovere un file

- **-r**: Permette di eliminare ricorsivamente tutto il contenuto presente nella directory passata
    - rm **-r** [nomeDirectory]

mv: Permette di spostare uno o più file da una directory all'altra

- mv [nomeFile] [nomeDirectory]
- mv [nomeFile] [nomeNuovoFile] → Permette in realtà di rinominare un file, spostando il suo contenuto in uno nuovo ed eliminandolo

cp: Permette di copiare uno o più file da una directory all'altra

- cp [nomeFile] [nomeDirectory]
- cp [nomeFile] [nomeNuovoFile] → Permette in realtà di creare una copia di un file.
- **-r**: Permette di copiare ricorsivamente tutti i file di una directory in un altra:
    - cp -r [nomeDirectory] [nomeDirectory]

ESECUZIONI COMANDI E FILE

I comandi vengono eseguiti invocando direttamente il loro nome

I file vengono eseguiti passando:

- Percorso relativo o assoluto
- Solo il nome se presenti nella variabile PATH

Una subshell con una copia dell'ambiente di esecuzione (variabili) attuale viene creata e poi cancellata quando:

- Esegue una command list
- Esegue uno script
- Esegue un processo in background

&: Permette di eseguire un processo in background

- [nomeScript]/[comando] &

source: Permette di eseguire un processo senza creare una subshell

- source [nomeProcesso]/[comando]
- Può essere molto utile per esempio per modificare variabili della shell padre.

VARIABILI

Le variabili possono essere:

- Locali: Non trasmesse alle subshell
- Ambiente: Trasmesse alle subshell
    - Però se modificate nella subshell, non cambiano anche in quella padre
        - Per farlo usa source

Per creare una variabile locale si fa: [nomeVariabile]=[valore]

Per utilizzare il valore contenuto in una variabile si fa: ${[nomeVariabile]}

Per vedere quanti caratteri contiene una variabile si fa: ${#[nomeVariabile]}

export: Permette di creare variabili di ambiente

- export [nomeVariabile]=[valore]

env: Permette di visualizzare l'elenco delle variabili d'ambiente

set: Se lanciato senza parametri stampa a video TUTTE le variabili locali, di ambiente e function

- Se lanciato con parametri serve a settare o resettare un opzione di comportamento della shell
    - set **-a**: Fa si che che tutte le variabili create o modificate siano d'ambiente
    - set **+a**: Fa si che tutte le variabili create o modificate siano locali (comportamento di default)

unset: Permette di eliminare una variabile

Comportamente speciale: Se definisco una variabile nella stessa riga di un comando allora se un altra variabile è presente ed ha lo stesso nome non viene considerata.

- export var=”inizio”
- var=”fine” [comando] → lui visualizza “fine”
- echo ${var} → lui visualizza “inizio”

VARIABILI SPECIALI

PATH: Contiene una sequenza di percorsi assoluti nel filesystem di alcune directory in cui sono contenuti gli eseguibili

- which: Mi dice se il comando passato in argomento è presente nei percorsi specificati dalla variabile PATH
    - Esplora in ordine le directory indicate da PATH e successivamente se è presente restituisce il percorso.

IFS: Contiene quei caratteri che sono considerati dediti alla separazione delle parole

- Si può modificare in questo modo: IFS=**$’**[caratteri]**’**

RANDOM: Permette di generare numeri casuali in un range tra 0 e 32k-1

- Per usarla si fa semplicemente ${RANDOM} e mi genera un numero random ogni volta che ne richiedo il valore
- Posso anche impostare dei range
    - $((${RANDOM} % [n])) -> Mi genera numeri random da 0 a [n]-1
    - $(([m] + (${RANDOM} % [n]))) -> Mi genera numeri random da [m] a ([m]+[n])-1
- Se imposto il valore di RANDOM ed inizio a generare dei numeri allora ogni volta che imposterò tale valore riceverò la medesima sequenza di numeri
    - È però buona prassi sfruttare questa cosa, per far si che in esecuzioni ravvicinate che utilizzino random non si ricevano le stesse sequenze. Abbiamo due modi per farlo:
        - ((RANDOM=$$))
        - RANDOM=$((`date +%s`))

RIFERIMENTI INDIRETTI A VARIABILI

Posso utilizzare valori di altre variabili, se la variabile inserita nel comando ha come valore il nome di un altra variabile, e l’operatore che fa questo è: ${!

- varA=pippo
- varB=varA

- echo ${!varB} → stampa “pippo”

MANIPOLAZIONI DI VARIABILI (stringhe)

${[nomeVariabile]:[indice]:[offset]} → Mi da esattamente (o finchè ci sono) caratteri indicati da [offsett] di una variabile a partire dalla posizione [indice]

- Esempio: VAR=colibri, echo ${VAR:1:3} → stampa “oli”
- Esempio: VAR=colibri, INDICE=4, echo ${VAR:1:${INDICE}} → stampa “olib”

${[nomeVariabile]%%[pattern]} → Permette di rimuovere il più lungo suffisso che fa match con il pattern passato

- Il pattern può contenere metacaratteri * o ?
- Esempio: VAR=” [13] qualcosa con [o] fine ”, ${VAR%%]*} → Restituisce [13, in quanto il resto era il suffisso più lungo

${[nomeVariabile]%[pattern]} → Permette di rimuovere il più corto suffisso che fa match con il pattern passato

- Il pattern può contenere metacaratteri * o ?
- Esempio: VAR=” [13] qualcosa con [o] fine ”, ${VAR%]*} → Restituisce [13] qualcosa con [o , in quanto era il suffisso più corto

${[nomeVariabile]##[pattern]} → Permette di rimuovere il più lungo prefisso che fa match con il pattern passato

- Il pattern può contenere metacaratteri * o ?
- Esempio: VAR=” [13] qualcosa con [o] fine ”, ${VAR##*[} → Restituisce o] fine, in quanto il resto era il prefisso più lungo

${[nomeVariabile]#[pattern]} → Permette di rimuovere il più corto prefisso che fa match con il pattern passato

- Il pattern può contenere metacaratteri * o ?
- Esempio: VAR=” [13] qualcosa con [o] fine ”, ${VAR#*[} → Restituisce 13] qualcosa con [o] fine , in quanto era il prefisso più corto

${[nomeVariabile]/[pattern]/[string]} → Permette di sostituire solo la prima occorrenza che matcha con il pattern con il string

- Esempio: VAR=”Sono alberto ed ho 21 anni”, ${VAR/”alberto”/”cristian”} → Restituisce come stringa “Sono cristian ed ho 21 anni”

${[nomeVariabile]//[pattern]/[string]} → Permette di sostituire ogni occorrenza che matcha con il pattern con il string

HISTORY EXPANSION

history: Permette di visualizzare in un elenco numerato tutti i comandi fatti dall'accensione della bash

- È un comando built-in
- I numeri sono fissi e non cambiano

Con **!** seguito da un [n] posso eseguire [n] comando dell'history

- **!**[n]
- Esempio: 47 echo "ciao"
    - !47 → viene stampato "ciao"

Posso anche far seguire **!** da un carattere [char], in questo caso verrà eseguito il comando più recente che inizia con tale [char]

- Esempio: 47 echo "ciao", 48 ls, 49 echo "peppino"
    - !e → viene eseguito echo "peppino"

set **+o** history: Disabilita la memorizzazione dei comandi
set **-o** history: Riabilita la memorizzazione dei comandi

BRACE EXPANSION

Tramite l'utilizzo di {} è possibile inserire al loro interno una serie di stringhe separate da virgole che andranno poi ad inserirsi oppure a sostituirsi all'interno dei comandi

- **{**[char],[char1],[char2]**}**
- Se voglio che sia presente uno spazio nelle stringhe inserite all'interno delle graffe allora devo fare quoting con \
    - Esempio: echo ciao{g\ iuseppe,peppe}com → stampa "ciaog iuseppecom ciaopeppecom"
- È anche possibile avere un preambolo prima di queste graffe ed un post dopo queste graffe, che andranno ad aggiungersi a tutte le varie opzioni delle graffe

    - Esempio: echo fo{rte,rti,rza}maximum → stampa “fortemaximum fortimaximum forzamaximum”

È possibile anche annidare più brace expansion fra loro

- Esempio: echo /usr/{ucb/{ex,sm},lib/{al,bp}} → stampa “/usr/ucb/ex /usr/ucb/sm /usr/lib/al /user/lib/bp”

Siccome vengono fatte prima le brace expansion rispetto la variable expansion, nelle brace è possibile inserire anche delle variabili

- Esempio: varA=c, varB=d, echo c{${varA},${varB}}p → stampa “ccp cdp”

Utilizzando l’operatore **{**[char/num]**..**[char/num]**}** sostituirà con tutte le lettere presenti da un estremo all altro (compresi)

- Esempio: echo c{a..c}d → stampa “cad cbd ccd”
- Esempio: echo c{2..5}d → stampa “c2d c3d c4d c5d”

TILDE EXPANSION

Ha 5 casi:

- Quelli in cui la tilde viene sostituita dal percorso assoluto della home directory dell’utente
    - ~
    - ~/
    - ~/[nomeFile]
- Quelli in cui la tilde è seguita da un nome di un utente, e quindi viene sostituita dal percorso assoluto della home directory
    - ~[nomeUtente]
    - ~[nomeUtente]/[nomeFile]

Per scriverla si usa la combinazione di tasti: ALT+126

PATHNAME EXPANSION (WILDCARDS)

Le wildcards sono interpretate dalla shell che cerca di sostituire con sequenze di caratteri per ottenere nomi di files nei filesystem

- * → Può essere sostituito da una qualunque sequenza di caratteri, anche vuota
- ? → Può essere sostituito da ESATTAMENTE un singolo carattere
- [elenco] → Può essere sostituito da un SOLO carattere tra quelli specificati nell'elenco

Cosa si può mettere nell'elenco:
- [lettere]
- [n-n] → Può essere sostituito con uno dei numeri del range in elenco
- [char-char] → Può essere sostituito con uno delle lettere del range in elenco
- [[:digit:]] → Può essere sostituito da un solo carattere numerico
- [[:upper:]] → Può essere sostituito da un solo carattere maiuscolo
- [[:lower:]] → Può essere sostituito da un solo carattere minuscolo

PARAMETER EXPANSION

È possibile passare argomenti a script facendo: [nomeScript] [p1] [p2] ..

Esistono variabili d'ambiente che contengono gli argomenti passati allo script:
- $# → Numero di argomenti passato
- $0 → Nome del processo in esecuzione
- $$ → PID del processo (subshell)
- $[n] → Argomenti passati
    - $1 primo argomento, $2 secondo argomento …
- $* → Tutti gli argomenti passati concatenati e separati da spazi
- $@ → Come $* ma se quotati, gli argomenti vengono quotati separatamente
    - "$*" → "$1 $2 $3"
    - "$@" → "$1" "$2" "$3"

IMPORTANTE: Per passare il numero di argomenti, se nello script successivo non si tiene conto del fatto che siano quotati, allora si può fare anche un'unica grande variabile dove gli elementi sono separati da spazio, e passare quella.

I parametri NON possono essere modificati

ARITHMETIC EXPANSION

È possibile valutare una stringa (ovviamente composta da cifre) come se fosse un'espressione costituita da operazioni aritmetiche tra numeri

- (()) → Racchiudendo tutta una riga che comprende espressioni più eventuale assegnamento
    - (([nomeVariabile]=${nomeVariabile}+1))
- $(()) → Valuta aritmeticamente solo una parte della riga di comando
    - Qui all'interno delle parentesi possono esserci spazi
    - echo pippo$((3+2)) → stampa pippo5

Dentro le parentesi posso usare altre parentesi () per indicare precedenze tra le varie operazione

Questa expansion funziona solo con le operazioni intere, se voglio avere quelle decimale allora devo passare il risultato in questo modo:
[script]/[comandoAritmetico] | bc -l -q

EXIT STATUS

Ogni programma e comando restituisce un valore alla shell, compreso tra 0 e 255 per indicare lo stato dell'esecuzione, e questo è proprio l'exit status.

Tale valore si trova nel: $?

Per ritornare uno specifico exit status si fa: exit [n]

EXIT STATUS PARTICOLARI

1 → General Error
2 → Uso sbagliato della shell builtins
126 → Il comando non può essere eseguito
127 → Comando non trovato
128 → Valore passato ad exit non valido (valore passato diverso da intero)
128+[n] → Fatal error signal "n"
130 → Script terminato per via di CTRL+C
255 → Exit status fuori dal range disponibile

SITUAZIONI PARTICOLARI

Le espressioni aritmetiche restituiscono:

- 0 tutto ok
- 1 errore

Nelle liste di comandi l'exit status restituito è quello relativo all'ultimo comando eseguito

COMMAND SUBSTITUTION

Permettono di sostituire a runtime, la riga del comando/programma con l'output prodotto dall'esecuzione del programma/comando stesso

L'operatori che fanno questo sono:

- `` (Backtick)
  - Si inseriscono con ALT+96
- $()

ATTENZIONE: Non è possibile annidare più command substitution

Esempio: var=`ls BUTTAMI/`

- for file in var; do …
  - Sostituisce a runtime var con tutti i file contenuti ottenuti da ls BUTTAMI/

QUOTING DELLE STRINGHE

" " → Quota in maniera parziale

- Disabilita l'utilizzo di ; per separare i comandi
- Disabilita la sostituzione di tutte le wildcard
- Permette la sostituzione a runtime delle variabili con il loro contenuto
- Permette le command substitution

' ' → Disabilita qualsiasi interpretazione

$' ' → Disabilita tutto tranne l'interpretazione dei caratteri speciali che iniziano con \

CONDITIONAL EXPRESSION

Sono operatori/comandi che valutano alcune condizioni e restituiscono un exit status

- 0 se è tutto corretto
- Diverso da 0 se c'è stato qualche errore

Esistono 3 operatori che permettono di fare espressioni condizionali

- [[ ]] → Il più nuovo
  - Utilizza → !, ||, &&
  - Permette di usare le parentesi tonde per gestire le precedenze nelle espressioni ed operatori
  - Si può andare a capo proseguendo l'espressioni
- [ ] e test
  - Utilizzano → !, -a (and), -o (or)
  - Non si possono usare le tonde
  - Si può andare a capo utilizzano per forza \ a fine riga

Sintassi (Uguale anche in []):

- [[espressione -opzione espressione]]
- Lo spazio dopo [[, prima e dopo le opzioni e prima di]] è SEMPRE necessario

All'interno delle espressioni condizionali non possono esserci comandi diretti ma si può inserire:

- Command substitution
- Parameter and Variable Expansion
- Arithmetic Expansion con $() → Quindi la versione che non quota l'intera riga
  - In queste però non possono esserci espressioni condizionali
- Process substitution
- Quote removal con gli operatori

NON possono essere annidate più espressioni condizionali

I confronti che possono essere fatti possono essere:

- Aritmetici
- Stringhe (Condizioni lessicografiche)
- File

(Per vedere quali sono fare: man bash e poi /CONDITIONAL EXPRESSION)

OPZIONI PARTICOLARI

-e: Controlla se il file esiste
-d: Controlla se il file esiste ed è una directory
-f: Controlla se il file esiste ed è un regular file
-r: Controlla se l'effective user ha il diritto di lettura per il file
-ot: Dati due file esistenti (i nomi), controlla se quello a sinistra è più vecchio di quello a destra

- [[[file] -ot [file2]]]

-nt: Dati due file esistenti (i nomi), controlla se quello a sinistra è più nuovo di quello a destra

- [[[file] -nt [file2]]]

-O: Controlla se il file esiste ed è posseduto dall'effective user
-z: Da true se la stringa ha lunghezza uguale a 0
-n: Da true se la stringa ha lunghezza non uguale a 0

FILE

I file descriptor è un astrazione che permette di accedere ai file

- Sono rappresentati con degli interi

Ogni processo punta alla propria file descriptor table che contiene un puntatore, uno per file, per accedere alla tabelle che contiene le informazioni di tale file

- I file descriptor di base sono:
  - 0 → stdin
  - 1 → stdout
  - 2 → stderr
- Una subshell eredita la tabella dei file della shell padre

    - Attenzione a non lavorare sullo stesso file, in scrittura, in contemporanea.

EXEC - APERTURA FILE

exec: Permette di aprire un file in diverse modalità

- exec n[opzione] [percorso]→ Gli passiamo direttamente un file descriptor
    - Se quel fd era già associato ad un altro file, allora si andrà a sostituire tale file, ed il file precedente non sarà più accessibile con questo fd
- exec {nomeVar}[opzione] [percorso] → Il s.o scegli in automatico un file descriptor libero.

Le opzioni stabiliscono in che modo apriremo il file:

- exec {nomeVar}< [pecorsoFile] → Apre in sola lettura
- exec {nomeVar}> [pecorsoFile] → Apre in sola scrittura
- exec {nomeVar}>> [pecorsoFile] → Apre in sola aggiunta in coda
- exec {nomeVar}<> [pecorsoFile] → Apre in lettura e scrittura

Ogni volta che apro un file devo SEMPRE chiuderlo e si fa:
n/{nomeVariabile}>&-

COMPORTAMENTE PARTICOLARE

Se voglio posso ridirigere l'output di un particolare file con la seguente sintassi:

- [fileDescriptor]>&n/${nomeVar}

Nella directory /proc sono presenti delle directory chiamate solo con numeri che rappresentano i PID dei processi in esecuzione sulla shell

- Contengono a loro volta molti file e directory, ma in particolare la directory fd contiene i file descriptor dei file aperti, e non chiusi, in quel processo.

ps: Mi permette di stampare, sotto forma di lista, i processi eseguiti attualmente nella bash su cui lo lancio .

- **-a**: Mi da tutti i processi in esecuzione sull'intera MACCHINA

- **-ux:** Mi da ulteriori informazioni sugli utenti legati ai vari processi

df: Mostra lo spazio totale ed utilizzato su tutti i file system disponibili
- **-h**: Mostra le dimensioni in potenze di 1024

READ

read: Permette di leggere dallo stdin una sequenza di caratteri fino ad invio
- read [nomeVar] [nomeVar2]...
    - Legge la sequenza e la suddivide per parole le inserisce nelle variabili
        - Se le variabili non esistono allora vengono create
        - Se il numero di parola è maggiore rispetto alle variabili disponibili allora il resto delle parole vengono concatenate nell'ultima variabile disponibile
        - Se il numero di prole è minore rispetto alle variabili disponibili allora il resto delle variabili saranno vuote
    - Se nella lettura non si incontra il carattere di terminazione riga, l'utente verrà avvisato attraverso exit status diverso da 0.
- **-n**: Mi permette di indicare il numero di caratteri massimo da poter leggere
    - Se i caratteri disponibili sono meno del massimo non c'è problema
    - Una read con quest'opzione rifatta sullo stesso output, fa sì che si riprenda dal carattere subito successivo rispetto all'ultima read, sempre se ce ne sono.
- **-N**: Mi permette di leggere ESATTAMENTE [N] caratteri
    - L'unico scenario in cui vengono letti meno caratteri rispetto quelli indicati è perché si incontra la fine dell'output.
- **-r**: Mi permette di NON interpretare i \ incontrati nel file come caratteri di escape
- **-u**: Mi permette di leggere input da un file passato
    - read -u ${nomeVarContenenteFD}
    - Per fare questo in maniera più semplice si può utilizzare il ridirezionamento
    - Continua fino a fine file
        - A fine file si ha sempre exit status 0, ma può capitare prima di non incontrare caratteri di terminazione riga, per questo

motivo devo controllare di aver letto effettivamente sempre qualcosa, per farlo si può fare così:

- [[$? == 0 || -n ${varRigaLetta}]] → Se è diverso da 0 controlla che non sia comunque stata letta una stringa diversa da vuota.

RIDIREZIONAMENTO FILE

In che consiste? Consiste nel mandare l'output dello stdin, stdout o stderr verso un altro file.

Gli operatori base che permetto di fare ciò sono:

- < → Permette di ricevere input da file
- > → Permette di mandare output su un file eliminandone il contenuto
- >> → Permette di mandare output su un file aggiungendolo al vecchio contenuto
- | → Permette di mandare output di un programma come input di un altro.
  - Posso mettere infiniti programmi in pipe
  - I programmi a destra della pipe vengono eseguiti in una subshell figlia per questo modifiche a variabili locali vengono cancellate appena è terminato il programma

È possibile ridirigere l'input anche per gani insiemi di comandi:

- Esempio: while read A B C; do echo $B; done; < file.txt → La read verrà eseguita avente come input il contenuto di file.txt

CASISTICA PARTICOLARE

É possibile avere una concatenazione di input in una casistica di questo modo:

- [scriptCheGeneraInput] | ([comando] ; [comando] …)
  - Esempio : cat [file.txt] | (read RIGA1; echo $RIGA1 ; read RIGA2; echo $RIGA2) -> La prima read riceva la prima riga mentre la seconda riceve la seconda

VARIANTI DI OPERATORI

- [program] < [file_input.txt] > [file_output.txt] → Permette di ridirezionare contemporaneamente sia input che output
- [program] &> [file_err_output] → Permette di ridirezionare stderr e stdout sullo stesso FILE
- [program] &> [file_err] > [file_output] : Permette di ridirezionare stderr e stdout su due FILE diversi.
- [program] |& [program] → Permette di ridirezionare stdout e stderr come input del programma a destra.
- [n]>&[n] → Permette di ridirezionare un file descriptor verso un altro.
    - Rende quello a destra come FD
    - Esempio: 1>&2 → Permette di ridirezionare lo stdout nel stderr.
- <<[parola] → Permette di passare come input tutte le righe che si trovano prima della [parola]
    - [parola] deve trovarsi all'inizio di una riga senza nulla che la preceda
    - I metacaratteri non vengono espansi
        - Esempio: <<F?NE → Cerca proprio la parola F?NE
- <<< → Permette di passare in input una SINGOLA riga

In generale se davanti agli operatori visti fino ad ora si mette un file descriptor specifico allora il ridirezionamento viene fatta da o verso quest'ultimo

- [n]/${FD}[operatore]
    - Esempio: 2>

NOTA BENE: I vari redirezionamenti vanno inseriti prima se bisogna avviare un processo in background

- Esempio: ./script.sh [argomenti] 1>&2 & → Avvia quello script in background, ma il suo stdout lo manda in stderr

COMANDI VARI

date: Permette di stampare la data corrente

- **-d**: Permette di passargli una stringa che indica che data vogliamo
    - date -d "tomorrow"/"yesterday"
    - date -d "today + 10days"
- **%s**: Dice i secondi passati a partire dal 1970-01-01 00:00 UTC
    - Si usa di solito così [var]=$((`date +%s`))

        - Permette di creare nomi univoci per variabili, oppure per quanto riguarda la variabile RANDOM permette di aumentare la probabilità realistica di non ricevere la stessa sequenza di numeri in due esecuzione ravvicinate

cat: Permette di leggere il contenuto di tutti i file passati, concatenarlo e stamparlo sullo stdout

- Se gli si passa solo un file stampa semplicemente l'output di quest'ultimo
- Più file vanno passati tramite |
    - Esempio: cat [file1] | [file2] | …

wc: Permette di stampare insieme, a partire da un input:

- Numero di linee
- Numero di parole
- Numero di byte
- Numero di caratteri
- **-l** : Stampa solo linee
- **-w** : Stampa solo le parole
- **-c** : Stampa solo i byte
- **-m** : Stampa solo i caratteri
- **-L** : Stampa il numero di caratteri della linea più lunga

grep: Permette di cercare tra lo stdin le righe contenenti [parola]

- grep [parola] -> Legge dallo stdin
    - [scriptCheGeneraOutput] | grep [parola]
- grep [parola] [nome_file.txt] -> Permette di applicare grep ma considerando come input il contenuto del file passato
- grep [parola] [percorso]* → Legge da tutti i file contenuti in quel percorso
    - grep '*' /usr/include/* → Applica grep su tutti i file della directory /usr/include/, non esplora in maniera ricorsiva
- **-d**: Permette di permette di fare qualcosa prima di applicare grep se i file sono delle directory
    - grep -d **skip** [parola] → Non applica grep su i file che SONO directory

- **-v**: Permette di selezionare e stampare solo le righe che NON contengono la parola cercata
- **-i**: Permette di ignorare la politica di “case sensitive”
    - Esempio: file contiene due righe “pippo asino” e “PIppo elefante”, grep “pippo” -i [file_conIeRighe] → Verranno stampate entrambe le righe
- **-c**: Stampa SOLO il numero di righe che hanno la parola cercata
- **-m**: Permette di fermare il comando grep dopo un [n] di righe lette
    - Esempio: grep ‘pippo’ -m 10 → Dopo aver letto esattamente 10 righe contenenti “pippo” si fermerà

tail: Permette di stampare solo le 10 righe finali di un output

- Posso anche passargli un file ed applicarlo su quello
    - tail [nome_file.txt]
- **-n**: Permette di far stampare solo le [n] righe finali di un output
    - tail **-n** [n]
- **-f**: Permette di far rimanere in attesa il comando tail, e stampa man mano le righe che vengono aggiunte al file su cui viene chiamato
    - tail **-f** [file_output.txt]
    - L’esecuzione si termina con CTRL+C
    - Solitamente le righe vengono aggiunte su un altra shell oppure da script in esecuzione in background che generano costantemente nuovo output (i log)

head: Permette di stampare solo le prime 10 righe di un output

- Posso anche passargli un file ed applicarlo su quello
    - head [nome_file.txt]
- **-n**: Permette di far stampare solo le prime [n] righe di un output
    - head **-n** [n]

tee: Permette di stampare l’output ricevuto su stdout e di salvare contemporaneamente su n file

- [comando_script_CheGeneraOutput] | tee [file1.txt] [file2.txt]

sed: È un editor di stringhe che permette di fare svariate cose

- **‘s’**: Permette di sostituire solo la prima occorrenza di certe cose con altre in ogni riga (ricordare che si può passare solo una cosa da

sostituire per comando sed, se bisogna togliere più cose allora utilizzare vari sed in pipe |)
    - sed ‘s/[cosaDaSostituire]/[cosaCheSaràInserita]/’
    - Se viene messo un [n] alla fine allora verrà sostituita l’ennesima occorrenza
        - sed ‘s/[cosaDaSostituire]/[cosaCheSaràInserita]/[n]’
    - Se viene messo un [n] ad inizio della riga verrà sostituita la prima occorrenza dell’ennesima riga
        - sed ‘[n] s/[cosaDaSostituire]/[cosaCheSaràInserita]/’
- **‘g’**: Messo in combo con ‘s’ permette di sostituire OGNI occorrenza di certe cose in ogni riga
        - sed ‘s/[cosaDaSostituire]/[cosaCheSaràInserita]/g’
- **-i** o **--in-place**: Permette di fare ciò che gli indichiamo su uno specifico file
    - sed -i [operazione] [nome_file.txt]
        - Esempio: sed -i ‘s/[cosaDaSostituire]/[cosaCheSaràInserita]/g’ [nome_file.txt]
- **^**: indica l’inizio di una riga
    - ^. -> Indica che il primo carattere può essere qualsiasi
        - Esempio: sed ‘s/^.//g’ -> Permette di eliminare il primo carattere di ogni riga
- **$**: Indica la fine di una riga
    - .$ -> Indica che l’ultimo carattere può essere qualsiasi
    - Esempio: sed ‘s/.$//g’ -> Permette di eliminare l’ultimo carattere di ogni riga
- È possibile concatenare più sed in questo modo ‘[opzione]**/**[cosaDaSostituire]/[cosaCheSaràInserita]**/**[opzione]**;**[opzione]**/**[cosaDaSostituire]/[cosaCheSaràInserita]**/**[opzione]’
    - Esempio: echo "${RIGA}" | sed 's/?/\\?/g;s/*/*/g;s/\[/\\\[/g;s/\]/\\\]/g’ → Permette di sostituire ? * [ ] con \? * \[\]

cut: Permette di estrarre una specifica parte di un input ricevuto e stamparlo

- **-b** : Permette di estrarre un [n] carattere specifico, un certo range di caratteri oppure più intervalli di caratteri
    - cut -b [n] -> Estrapola l’ennesimo carattere
    - cut -b [n]-[n] -> Estrapola i caratteri nel range specificato, estremi compresi

        - cut -b -[n] -> Estrapola i caratteri dall'inizio fino all'ennesimo compreso
        - cut -b [n]- -> Estrapola i caratteri a partire dall'ennesimo carattere fino alla fine
    - cut -b [n],[m]-[m],... -> Estrapola i caratteri richiesti dalle varie operazione specificate
- Si può passare direttamente un file, ed allora cut si applica ad ogni riga di tale file
    - cut -b [n] [nome_file.txt]

sort: Ordina le linee a partire da un input

- sort [file.txt]
- < [file.txt] | sort → Permette di ordinare un file prima di mandarlo in input ad altro
- **-k** [n]: Ordina il file confrontando elementi della specifica colonna [n]
- **-R**: Ordina gli elementi in ordine inverso

PROCESSI E JOBS

Ogni processo creato eredita il terminale di controllo del padre.

- Tutti i processi che ereditano lo stesso terminale di controllo formano un "gruppo di processi"
    - Quando il terminale viene chiuso, quest'ultima prima di terminare manda un comando kill a tutti i processi del gruppo di processi, per far si che anche questi terminino.

Processo in foreground: È un processo che quando viene avviato assume il controllo della shell interattiva, rendendola così "inagibile" all'utente.

Processo in background: È un processo che quando viene avviato, condivide stdin, stdout e stderr con la shell interattiva, ma non la blocca all'utente.

- L'utente è così in grado di lanciare altri comandi
- [comando] & → Permette di avviare tale comando in background
    - Tale comando essendo figlio diretto della shell su cui viene lanciato, sarà detto job di tale shell
        - Se tale job ne starta un altro allora però quest'ultimo NON sarà job della shell padre iniziale.

- Se si lanciano più comandi che attendono un output da tastiera bisogna ricordare che NON si saprà chi leggerà prima.
- Quando si avvia un processo in background vengono stampati il suo JobID ed il suo PID
    - Quando il processo termina, non appena l'user preme un tasto, viene ristampato il JobID del processo terminato, e le istruzioni di tale processo

Il JobID è:

- Univoco per ogni singola shell
- [JobId]**+** → Indica che il job appena terminato era stato l'ultimo a essere "creato"
- [JobId]**-** → Indica che il job appena terminato era stato il penultimo a essere "creato"

jobs: Permette di visualizzare la lista di tutti i jobs in esecuzione sulla shell chiamante

$! → Contiene il PID del comando messo in background più di recente

CTRL+Z: Blocca un processo in foreground e lo rende un job sospeso

fg: Permette di riattivare il job in sospeso più recente rendendolo un processo in foreground

- %[JobId]: Rende processo in foreground il job avente il JobId specificato.
- CASO PARTICOLARE
    - Dopo aver riattivato un processo sospeso e messo in background, si può comunque riportarlo in foreground utilizzando fg o fg %[JobId]

bg: Permette di riattivare il job in sospeso più recente rendendolo un processo in background

- %[JobId]: Rende processo in background il job avente il JobId specificato.

kill: Permette di fermare un processo o un job, manda un messaggio predefinito TERMINATE

- Lo stato del processo sarà “Terminated”
- kill [PID]
- kill [JobId]
- **-9** o **-SIGKILL**: Manda un messaggio di KILL, e termina SEMPRE il processo
    - Lo stato del processo sarà “Killed”

nohup: Permette di creare un processo sganciato dal terminale

- nohup [comando]
- Questo dissociamento permette di far vivere i processi anche dopo la chiusura del terminale

disown: Permette di sganciare l’ultimo processo creato dal terminale

- -r: Permette di sganciare dalla shell tutti i job in stato running
- -a: Permette di sganciare dalla shell tutti i job in stato running e sospesi
- %[JobId]: Permette di sganciare dalla shell il job avente [JobId]
- Questo dissociamento permette di far vivere i processi anche dopo la chiusura del terminale

wait: Permette di attendere la terminazione di tutti i processi attivi sulla shell chiamante

- wait [PID]: Permette di attendere la terminazione del processo avente [PID]
- wait %[JobId]: Permette di attendere la terminazione del processo avente [JobId]
- Accetta solo figli diretti come argomenti
- Restituisce l’exit status del processo che si sta attendendo
    - Se l’exit status restituito è 127, allora questo vuol dire che il PID o JobId passato come argomento o non è un figlio diretto della shell oppure non esiste

ps: Mi permette di stampare, sotto forma di lista, i processi eseguiti attualmente nella bash su cui lo lancio .

- **-a**: Mi da tutti i processi in esecuzione sull’intera MACCHINA

- **-ux:** Mi da ulteriori informazioni sugli utenti legati ai vari processi

Processo zombie: È un processo figlio terminato per cui il padre non ha ancora fatto la wait

- Wait serve anche al sistema operativo per eliminare completamente i processi dalla macchina
- Si può incontrare un errore che indica che non sono più disponibili process identifier da assegnare in quanto sono presenti troppi processi zombie nella shell