Interfaccia a caratteri e Bash scripting

Vittorio Ghini

A cura di Giovanni Maria Rava.

Indice

11. **Script e Automazione**
    - Esempi di scripting
    - Ridirezione e gestione dei flussi I/O
12. **Operatori e Condizioni**
    - Espressioni condizionali
    - Confronti su stringhe, numeri e file
13. **Gestione dei Processi**
    - Foreground e background
    - Controllo dei job e comandi utili (`jobs`, `kill`)
14. **Comandi GNU Coreutils**
    - Comandi principali (`head`, `tail`, `cut`, `sed`, ecc.)
    - Esempi e applicazioni
15. **File Descriptor e Stream**
    - Apertura, gestione e chiusura
    - Ridirezione e utilizzo di `exec`
16. **Processi e Terminale**
    - Concetti di zombie process e controllo terminale
    - Uso di `nohup` e `wait`
17. **Il comando `find`**
    - Opzioni principali (`-type`, `-exec`, ecc.)

Introduzione

L'interfaccia utente a caratteri rappresenta un potente strumento per interagire con i sistemi operativi Unix e Linux. Tramite la **Command Line Interface (CLI)**, possiamo impartire comandi e gestire operazioni complesse in modo diretto ed efficiente. Questo tipo di interazione è particolarmente utile per attività tecniche avanzate come la programmazione, il debugging e la gestione di file e processi. La shell Bash, una delle più popolari, è un'interprete di comandi che consente di eseguire comandi e script, gestendo operazioni basilari e avanzate.

Struttura di un sistema operativo

Un sistema operativo è organizzato in strati, dove ogni livello ha compiti specifici:

1. **Hardware**: Include CPU, memoria e dispositivi I/O.

2. **Kernel**: Il cuore del sistema operativo, che si occupa di:

   - Gestione della memoria e dei processi.

   - Comunicazione con l'hardware tramite driver.

   - Fornitura di interfacce per le applicazioni utente.

3. **Librerie di sistema**: Offrono funzioni di base, come l'input/output, utili per gli sviluppatori.

4. **Utility e applicazioni utente**: Strumenti come editor di testo, compilatori e altre applicazioni.

Questa struttura stratificata garantisce modularità, sicurezza e facilità d'uso.

Il kernel e le librerie

Il kernel funge da intermediario tra hardware e software. Per esempio, quando un'applicazione deve leggere un file, non interagisce direttamente con il disco rigido, ma utilizza una chiamata di sistema (`system call`) gestita dal kernel. Questa astrazione consente agli sviluppatori di concentrarsi sullo sviluppo delle applicazioni senza preoccuparsi dei dettagli dell'hardware.

Le **librerie di sistema** come `libc` forniscono API che semplificano l'uso delle chiamate di sistema. Un esempio comune è `printf` in C, che invoca indirettamente la chiamata di sistema `write` per scrivere dati sul terminale.

Modi di esecuzione della CPU

Nei sistemi operativi moderni, la CPU opera in diversi anelli di privilegio:

- **Ring 0**: Il livello più alto, riservato al kernel.

- **Ring 3:** Livello per le applicazioni utente.

Questo modello garantisce sicurezza: un'applicazione utente non può accedere direttamente all'hardware, evitando interferenze con il sistema operativo.

Interfaccia a linea di comando (CLI)

La CLI è uno strumento potente per il controllo del sistema. Permette agli utenti di:

- Navigare nel file system.

- Gestire file e directory.

- Automare operazioni tramite script.

A differenza dell'interfaccia grafica, che punta sulla facilità d'uso, la CLI offre precisione, flessibilità e controllo avanzato.

Shell e scripting

La shell Bash è un interprete che legge comandi, li interpreta ed esegue. Oltre a eseguire comandi interattivi, consente di creare **script** per automatizzare sequenze di operazioni. Le espansioni come `brace expansion` e `parameter substitution` rendono la shell estremamente potente.

Esempio di scripting:

```
#!/bin/bash

echo "Hello, World!"
```

Questo script stampa "Hello, World!" sul terminale.

Standard POSIX

È una famiglia di standard sviluppato dall'IEEE. POSIX.1 include alcune primitive della C standard Library, però non tutte le primitive della C Standard Library fanno parte di POSIX.1. Se un sorgente C è conforme a Posix può essere eseguito, una volta ricompilato su qualunque sistema Posix. Per scrivere un programma conforme a questo standard bisogna includere gli header files richiesti dalle varie primitive usate.

man nome_primitiva: comando usato per ottenere la lista completa degli header necessari ad utilizzare ciascuna primitiva.

Tipi primitivi di POSIX:

I tipi di dato interi sono definiti negli header *stdint.h e inttypes.h int8_t int16_t int32_t uint8_t uint16_t uint32_t, intptr_t, uintptr_t.* Lo header *sys/types.h* definisce i tipi dei dati di sistema primitivi.

File system

È l'organizzazione del disco rigido che permette di contenere i file e le loro le informazioni.

Lo spazio di ciascun disco rigido è suddiviso in una o più parti dette partizioni. Queste contengono:

- Dei contenitori di dati detti files
- Dei contenitori di files, dette directories

In Windows le partizioni del disco sono viste come logicamente separate e ciascuna è indicata da una lettera (C; B: Z: ...).

In Linux e Mac le partizioni sono viste come collegate tra loro. Esiste una partizione principale il cui nome è / (slash). Ciascun utente di un computer ha a disposizione dello spazio su disco per contenere i suoi files. Per ciascun utente esiste una directory in cui sono contenuti i files. Quella directory viene detta home dell'utente. per accedere ad un computer ciascun utente deve utilizzare farsi riconoscere mediante un nome utente (account) e autenticarsi mediante una password. Questa operazione si chiama login. L'autenticazione permette che il sistema operativo protegga i files di un utente impedendo l'accesso da parte di altri utenti o di persone esterne.

pwd: comando utilizzato dall'utente per sapere in quale directory si trova logicamente in quel momento

cd: comando utilizzato per spostarsi tra directory.

.. : indica la directory superiore.

BASH E COMANDI

La shell legge ciascuna riga di comando, e la interpreta eseguendo alcune operazioni che servono:

- Per capire dove finisce un comando e dove inizia il successivo
- Ad individuare le parole e a indentificare le parole riservate del linguaggio e a identificare la presenza di costrutti come for, while, if
- Sostituire alcune parti della riga di comando, interpretando i caratteri speciali

Successivamente la shell opera alcune sostituzioni nella riga di comando effettuando le expansion. Le espansioni principali, elencate in ordine di effettuazione, sono:

- History expansion
- Brace expansion
- Tilde expansion
- Parameter and variable expansion
- Arithmetic expansion
- Command substitution
- Word splitting
- Pathname expansion
- Quote removal

echo: comando che permette di visualizzare a video la sequenza dei caratteri scritti subito dopo a echo e fino al primo carattere di andata a capolinea.

Il comando:

echo pippo pippa pippi

Visualizza

pippo pippa pippi

Se ho bisogno di far stampare a video anche caratteri speciali come punti e virgola, andare a capo, devo inserire sia prima che dopo la stinga il separatore ".".

Variabili

Sono simboli dotati di un nome e un valore. Le variabili di una shell possono essere stabilite e modificate sia dal sistema operativo sia dall'utente che usa quella shell. Alcune variabili vengono definite d'ambiente e vengono impostate subito dal sistema operativo non appena viene iniziata l'esecuzione della shell. Altre variabili possono essere create ex-novo dall'utente in un qualunque momento dell'esecuzione della shell. La shell riconosce i nomi delle variabili contenuti negli ordini digitati, e cambia il contenuto dell'ordine sostituendo al nome della variabile il valore della variabile. Per fare in modo che la shell distingue il nome di una variabile questa deve essere preceduta dalla coppia di caratteri ***${***e seguita dal carattere***}.***

NB: quando si assegna un valore ad una variabile, NON sono consentiti spazi ne prima ne dopo il simbolo "="

PATH→ variabile d'ambiente importantissima. Viene impostata dal sistema operativo già dall'inizio dell'esecuzione della shell. L'utente può cambiare il valore di questa variabile. Contiene una sequenza di percorsi assoluti nel filesystem di alcune directory in cui sono contenuti degli eseguibili**.**

Usando l'interfaccia utente a linea di comando possono essere eseguiti:

- **Comandi built-in →** sono implementati e inclusi nella shell stessa e quindi non esistono come file separati. Sono forniti dal sistema operativo
- **File binari eseguibili →** file che contengono codice macchina e che si trovano nel filesystem
- **Script →** sono file di testo che contengono una sequenza di nomi di comandi, binari e altri script che verranno eseguiti uno dopo l'altro.

Nei sistemi Unix esistono:

- Le astrazioni di un utente (*user*), che è caratterizzata da una stringa chiamata username che contiene il nome utente e da un identificatore numerico chiamato userID.
- Gruppo di utenti (*group*) che è caratterizzato da una stringa chiamata groupname che contiene il nome del gruppo e da un identificatore numerico chiamata groupID.

Ogni file ha un proprietario ed un gruppo proprietario.

chown nuovoproprietario nome file → comando tramite il quale il proprietario di un file può cambiare il proprietario.

Ciascun file mantiene diversi diritti di lettura, scrittura ed esecuzione assegnati al proprietario.

Lettura (valore 4, simbolo r):

- File: lettura
- Directory: elenco file/directory nella cartella.

Scrittura (valore 2, simbolo w):

- File: modifica
- Directory: creazione, eliminazione, cambio nome file

Esecuzione (valore 1, simbolo x):

- File: esecuzione
- Directory: accesso all'interno della directory e proprietà di suoi file e directory

chmod, chgrp → comandi usati per cambiare i permessi del proprio file.

Quando scriviamo il comando "ls -al" su un terminale la prima parte delle righe recita: rwxrwxr-x (o simili). Dividendo in triplette vediamo che permessi ha ogni utente, se è presente un meno quell'utente non ha quel permesso. Con "r" si indica la lettura e ha valore 4, con "w" si indica la scrittura e ha valore 2, con "x" si indica la possibilità di eseguire quel file e ha valore 1. Se voglio cambiare i permessi dei vari utenti/gruppi utilizzo il comando chmod. Posso scrivere:

chmod 741 nome_file ed ottengo che rwxr----x, perché assegno tutti i permessi al proprietario 4+2+1=7, solo lettura =4 agli altri utenti, solo possibilità di eseguire ai gruppi=1.

Tutte le informazioni dei permessi e proprietà dei file sono salvate nelle directory nel file system e non direttamente sul file.

In Linux esiste il concetto di file nascosto, il cui nome inizia con ".". i file nascosti non vengono visualizzati dal comando **ls** a meno che non aggiunga l'opzione "-a" al comando.

Come eseguire da Terminale a linea di comando?

I comandi possono essere eseguiti semplicemente invocandone il nome.

I file eseguibili (binari o script) devono essere invocati specificandone o il percorso assoluto (a partire dalla root) oppure il percorso relativo (a partire dalla directory corrente), oppure il solo nome, a condizione che quel file sia contenuto in una directory specificata nella variabile d'ambiente PATH.

Una SUBSHELL è una shell (shell figlia) creata da un'altra shell (shell padre). Questa viene creata in caso di:

- Esecuzione di comandi raggruppati
- Esecuzione di script
- Esecuzione di processo in background

NB → L'esecuzione di un comando built-in avviene nella stessa shell padre.

Ogni shell ha una propria directory corrente, ha delle proprie variabili, eredita dalla shell padre una copia delle variabili d'ambiente ma non eredita le variabili locali della shell padre.

Ogni shell supporta due tipi di variabili:

- **Variabili locali** → non trasmesse da una shell alle subshell da essa create. Vengono utilizzate per computazioni locali all'interno di uno script.
- **Variabili d'ambiente** → vengono trasmesse dalla shell alla subshell. Viene creata una copia della variabile per la suibshell. Se la subshell modifica la sua copia della variabile la variabile originale nella shell non cambia. Alcuni esempi: $HOME; $PATH; $USER; $SHELL; $TERM

env (enviroment) → comando utilizzato per visualizzare l'elenco delle variabili di ambiente.

export nomevariabile → comando utilizzato per trasformare una variabile locale già dichiarata in una variabile d'ambiente.

Se una shell lancia uno script questo viene eseguito in una subshell con l'interprete di comandi indicato nella prima riga speciale dello script #!/bin/bash se esiste quella riga, o con lo stesso interprete della shell chiamante, se quella riga non esiste

source nomescript / **./nomescript** → comando utilizzato per eseguire uno script senza creare la suibshell in cui lo script dovrebbe essere eseguito.

Creazione di variabili d'ambiente da passare a subshell e Campo di visibilità di variabili

È possibile definire una o più variabili nell'ambiente (nella subshell) di un eseguibile che si sta per fare eseguire, senza farle ereditare a quelli successivi, scrivendo le assegnazioni prima del nome del comando.

var = "stringa" comando #comando vede e può usare var

echo ${var} #echo non vede la variabile var

se esiste già una variabile d'ambiente con quello stesso nome, il comando lanciato vede solo la variabile nuova, alla fine del comando torna visibile la variabile vecchia.

export var = "contenutoiniziale"

var="stringa" comando # comando vede var che vale "stringa"

echo ${var} #echo vede la variabile che vale "contenutoiniziale"

Variabile vuota → esiste ma è stato assegnato come valore la stringa vuota ""

Variabile che non esiste → non è mai stata dichiarata

unset nomevariabile → comando utilizzato per eleminare una variabile esistente.

Una variabile vuota o non esistente può provocare errori sintattici a runtime negli script in cui si confronta il valore delle variabili.

Funzionalità history/history expansion

Memorizza i comandi lanciati dalla shell e permette di visualizzarli ed eventualmente rilanciarli. La storia dei comandi viene mantenuta anche dopo la chiusura della shell. Il comando built-in history visualizza un elenco numera dei comandi precedentemente lanciati dalla shell, con numero crescente dal comando più vecchio verso il più recente. Il numero assegnato ad un comando, quindi, non cambia quando lanio altri comandi. Il numero serve a rilanciare quel comando.

Il comando **set**

Lanciato senza nessun parametro visualizza tutte le variabili della shell, sia quelle locali d'ambiente. Inoltre, visualizza le funzioni implementate nella shell.

Lanciato con dei parametri serve a settare o resettare una opzione di comportamento della shell in cui viene lanciato

Alcuni esempi

set +o history → disabilita la memorizzazione di ulteriori comandi eseguiti nel file di history. I comandi lanciati prima della disabilitazione rimangono nel file history.

set -o history → abilita la memorizzazione dei comandi eseguiti mettendoli nel file di history

set -a → causa il fatto che le variabili create o modificate vengono immediatamente fatte diventare variabili d'ambiente e vengono ereditate dalle eventuali shell figlie.

set +a → causa il fatto che le variabili create sono variabili locali. È il modo di default.

Parametri a riga di comando passati al programma

Sono un insieme ordinato di caratteri, separati da spazi, che vengono passati al programma che viene eseguito in una shell, nel momento iniziale in cui il programma viene lanciato. Quando da una shell digito un comando da eseguire, gli argomenti devono perciò essere digitati assieme al nome del programma per costituire l'ordine da dare alla shell

Tipologie di Shell:

- **Shell non interattiva** → shell figlia che esegue script, lanciata con argomenti *-c percorso_script_da_eseguire.* Non esegue i comandi contenuto in nessune dei file che customizzano il comportamento della shell interattiva.
- **Shell interattiva NON di login** → quella che vediamo all'inizio nella finestra di terminale lanciata senza nessuno degli argomenti *-c -l –login*. Quando incomincia la sua esecuzione cercare di eseguire solo il file *".bashrc"* nella home directory dell'utente;
- **Shell interattiva di login** → è come la shell non di login ma inizia chiedendo user e password lanciata con argomenti *-l* oppure *– login*. Nel momento in cui comincia la sua esecuzione, cerca di eseguire i seguenti file:
    1. Il file "/etc/profile"
    2. Poi uno solo (il primo che trova) tra i file "*.bash_profile*", "*.bash_login*" e "*.profile*" collocati nella home directory dell'utente e che risulti essere disponibile
    3. Poi il file "*.bashrc*" collocato nella home directory dell'utente

Brace expansion - generazione di stringhe

La bash, quando passo una riga di comando da eseguire, la bash cerca di capire se nella riga sono presenti delle coppie di parentesi graffe che rappresentano un ordine di generare delle stringhe secondo delle regole.

Un ordine di brace expansion è una stringa di testo, racchiusa tra separatori quali spazi, tab o andate a capo, al cui interno compaiono una coppia di graffe precedute da un $ e al cui interno non ci sono spazi bianchi o tab. questa stringa che ordina brace expansion è formato da tre parti, di cui la prima (*preambolo*) e l'ultima (*postscritto*) possono non essere presenti. La parte di mezzo è quella racchiusa all'interno di una coppia di parentesi graffe. Dentro le graffe sono presenti una o più stringhe separate da virgole.

Ciascuna stringa rappresenta una possibile scelta di stringhe che possono essere aggiunte al preambolo e seguite dal postscritto per formare delle nuove stringhe di testo.

echo va{caga,ffancu,ammori,catihafat}lo

Produce come output

Vaacagalo vaffanculo vaamorilo vacatihafatlo

Alcune regole:

- Possono mancare preambolo o postscritto
- Non possono essere presenti spazi bianchi o tab altrimenti alla brace expansion non viene riconosciuta e non viene applicata l'espansione.
- Se voglio far generare stringhe con spazi bianchi, devo proteggere i singoli spazi con dei caratteri backslash
- Non posso proteggere tutto l'ordine di brace expansion con doppi apici perché disabilitano l'interpretazione dell'espansione, ma posso proteggere ciascuna singola stringa

È possibile inserire degli ordini di brace expansion all'interno di altri ordini di brace expansion

echo /usr/{ucb/{ex,edit},lib/{bin, sbin}}

Ottiene come output

/usr/ucb/ex /usr/ucb/edit /usr/lib/bin /usr/lib/sbin

È possibile inserire delle variabili negli ordini di brace expansion all'interno di altri ordini di brace

A=bin

=log

C=boot

echo ${A}{%{B}${C},%{C},${A}${B}}a

Visualizza

binlogboota binboota binbinloga

Esiste una particolare forma di brace expansion che consente di generare stringhe elencando un intervallo di caratteri che possono essere usati come possibili scelte

echo a{b..k}m

ottiene come output

abm acm adm aem afm agma ahm aim ajm akm

Tilde Expansion

Riguarda 5 casi essenziali:

1- Un carattere tilde isolato
2- Un carattere tilde seguito da slash non quotato
3- Un carattere tilde seguito da slash non quotato seguito da altri caratteri
    - In questi casi la tilde viene sostituita dal percorso assoluto della home directory dell'utente che sta eseguendo la riga di comando, l'effective user
4- Una parola che inizia con la tilde seguita da un nome utente
5- Una parola che inizia con la tilde seguita da un nome utente, seguita da uno slash non quotato a cui possono seguire caratteri
    - In questi casi la tilde più nome utente viene sostituita dal percorso assoluto della home directory dell'utente specificato dal nome.

Wildcards ***** - **?** - **[...]**

***** → può essere sostituito da una qualunque sequenza di caratteri, anche vuota

ls /home/vittorio/.exe*

visualizza il nome di quei file della directory vittorio il cui nome termina per .exe (cioè primo.exe)

? → può essere sostituito da esattamente un singolo carattere

*ls /home/vittorio/primo**

visualizza i nomi di quei file della directory vittorio il cui nome inizia per primo, cioè primo.c primo.exe

[elenco] → può essere sostituito da un solo carattere tra quelli specificati in elenco

ls /home/vittorio/pri?o.c

visualizza il nome del file primo.c di directory vittorio

[abk**]** puo' essere sostituito da un solo carattere tra a b oppure k.

[1-7] puo' essere sostituito da un solo carattere tra 1 2 3 4 5 6 oppure 7.

[c-f**]** puo' essere sostituito da un solo carattere tra c d e oppure f.

[[:digit:]**]** puo' essere sostituito da un solo carattere numerico (una cifra).

[[:upper:]**]** puo' essere sostituito da un solo carattere maiuscolo.

[[:lower:]**]** puo' essere sostituito da un solo carattere minuscolo.

Comandi della bash

pwd →mostra directory di lavoro corrente .

cd percorso_directory →cambia la directory di lavoro corrente .

mkdir percorso_directory→ crea una nuova directory nel percorso specificato

rmdir percorso_directory→ elimina la directory specificata, se è vuota

ls-alh percorso→ stampa informazioni su tutti i files contenuti nel percorso

rm percorso_file →elimina il file specificato

echo sequenza di caratteri →visualizza in output la sequenza di caratteri specificata

cat percorso_file →visualizza in output il contenuto del file specificato

env →visualizza le variabili ed il loro valore

which nomefileeseguibile →visualizza il percorso in cui si trova (solo se nella PATH) l'eseguibile

mv percorso_file percorso_nuovo →sposta il file specificato in una nuova posizione

ps aux →stampa informazioni sui processi in esecuzione

du percorso_directory →visualizza l'occupazione del disco.

kill-9 pid_processo →elimina processo avente identificativo pid_processo

killall nome_processo→ elimina tutti i processi con quel nome nome_processo

bg →ripristina un job fermato e messo in sottofondo

fg →porta il job più recente in primo piano

df →mostra spazio libero dei filesystem montati

touch percorso_file→ crea il file specificato se non esiste, oppure ne aggiorna data.

more percorso_file →mostra il file specificato un poco alla volta

head percorso_file →mostra le prime 10 linee del file specificato

tail percorso_file →mostra le ultime 10 linee del file specificato

man nomecomando→ è il manuale, fornisce informazioni sul comando specificato

find →cercare dei files

grep→ cerca tra le righe di file quelle che contengono alcune parole

read nomevariabile →legge input da standard input e lo inserisce nella variabile specificata

wc →conta il numero di parole o di caratteri di un file

true →restituisce exit status 0 (vero)

false→ restituisce exit status 1 (non vero)

Parameter Expansion

Esistono variabili d'ambiente che contengono gli argomenti passati allo script:

- $# il numero di argomenti passati allo script
- $0 il nome del processo in esecuzione
- $1 primo argomento, $2 secondo argomento
- $* tutti gli argomenti passati a riga di comando concatenati e separati da spazi
- $@ come $* ma se quotato gli argomenti vengono quotati separatamente

I parametri non possono essere modificati. Il programma vede i parametri così come sono diventati dopo la eventuale sostituzione dei metacaratteri * e ?.

Posso chiedere di vedere e fare delle modifiche al contenuto prima di farlo vedere, senza fare una modifica alla variabile.

${VAR : offset : lenght}

Fa vedere la sottostringa di lunghezza "lenght" a partire dal carattere "offset".

Suffisso= parte finale della stringa

Prefisso = parte iniziale della stringa

Pattern può essere una variabile ma può contenere wildcard che cercano di matchare con sottostringhe della stringa passata

Rimuove il più lungo suffisso che fa match con stringa originale

${VAR%%pattern}

Rimuove il più corto suffisso che fa match con la stringa originale

${VAR%pattern}

Rimuove il più lungo prefisso che fa match con la stringa originale

${VAR##pattern}

Rimuove il più corto prefisso che fa match con la stringa originale.

${VAR#pattern}

Esprime la lunghezza in byte del contenuto di VAR

${#VAR}

Cerca nel contenuto di Var la sottostringa più lunga che fa match con il pattern specificato e lo sostituisce con string. Se sono possibili più sostituzioni viene sostituita quella piu vicino all'inizio.

${VAR/pattern/string}

Valutazione di espressioni con soli interi: operatori **(( ))** e **$(( ))**

È possibili valutare una stringa come se fosse una espressione costituita da operazioni aritmetiche tra soli numeri interi.

L'operatore (( )) racchiude tutta una riga di comando semplice, che deve essere una espressione più un eventuale assegnamento. Esegue la riga di comando che racchiude valutando aritmeticamente gli operandi.

Ad esempio:

((NUM=3+2))

Assegna 5 alla variabile NUM

L'operatore *$(( ))* invece serve se dovete racchiude solo una parte di una riga di comando, che deve essere una espressione. La bash prima valuta aritmeticamente l'espressione racchiusa dall'operatore $(( )), poi nella riga di comando, mette al posto dell'operatore il risultato calcolato. Infine, esegue la riga di comando modificata.

Le valutazioni aritmetiche possono contenere:

- Degli operatori aritmetici + - * / %
- Degli assegnamenti
- Delle parentesi tonde () per accorpare operazione e modificare precedenze

Riferimenti indiretti a Variabili – operatore **$ {!**

Avendo una variabile varA che contiene un valore qualunque, supponiamo di avere un'altra variabile il cui valore è proprio il nome della prima variabile. Voglio, quindi, usare il valore della prima variabile sfruttando solo il nome della seconda variabile il cui valore è proprio il nome della prima variabile: si sfrutta questo operatore.

varA=pippo

nomevar=varA

echo ${!nomevar} //stampa a video pippo

Exit Status

Ogni programma o comando restituisce un valore numerico compreso tra 0 e 255 per indicare se c'è stato un errore durante l'esecuzione oppure se tutto è andato bene. Un risultato 0 indica tutto bene mentre un risultato diverso da zero indica errore. Tale risultato non viene visualizzato sullo schermo bensì viene passato alla shell che ha chiamato l'esecuzione del programma stesso. In tal modo il chiamante può controllare in modo automatizzati il buon andamento dell'esecuzione dei programmi.

In C per restituire il risultato si usa l'istruzione return.

In uno script bash per restituire il risultato si usa il comando exit.

exit 9 //fa terminare lo script e restituisce 9 come risultato

Per catturare il risultato di un programma chiamato da uno script si usa una variabile d'ambiente predefinita $? Che viene modificata ogni volta che un programma o un comando termina e in cui vinee messo il risultato numerico restituito.

Exit Code Number	Meaning	Example	Comments
1	catchall for general errors	let "var1 = 1/0"	miscellaneous errors, such as "divide by zero"
2	misuse of shell builtins, according to Bash documentation		Seldom seen, usually defaults to exit code 1
126	command invoked cannot execute		permission problem or command is not an executable
127	"command not found"		possible problem with `$PATH` or a typo
128	invalid argument to exit	exit 3.14159	**exit** takes only integer args in the range 0 – 255
128+n	fatal error signal "n"	**kill –9** `$PPID` of script	**$?** returns 137 (128 + 9)
130	script terminated by Control–C		Control–C is fatal error signal 2, (130 = 128 + 2, see above)
255*	exit status out of range	exit –1	**exit** takes only integer args in the range 0 – 255

La valutazione aritmetica tramite gli operatori $(( )), oppure (( )), restituisce un valore di Exit Status che sarà:

__ **Diverso da zero** se durante la valutazione aritmetica è accaduto un **errore.**

((ls 5)) non valutabile aritmeticamente $? Conterrà 1

__ **Uguale a zero** se la valutazione aritmetica fornisce un risultato logico **true.**

((5 >= 2)) $? Conterrà 0

__ **Diverso da zero** se la valutazione aritmetica fornisce un risultato logico **false.**

((5 >= 2)) $? Conterrà 1

__ **Uguale a zero** se la valutazione aritmetica fornisce un risultato **intero diverso** da **zero.**

((4)) $? Conterrà 0

((VAR=5+3)) $? Conterrà 0

__ **Diverso da zero** la valutazione aritmetica fornisce un risultato **intero uguale** a **zero.**

((0)) $? Conterrà 1

*((VAR=6-2*3))* $? Conterrà 1

__ Se l'assegnamento è interno all'espressione (cioè non è eseguito per ultimo), allora il risultato dell'espressione è quello del confronto eseguito per ultimo.

*(((VAR=6-2*3) != 1))* assegna 0 a VAR ma $? Conterrà 0

Il risultato Exit Status restituita da una lista di comandi è l'Exit Status restituito dall'ultimo comando che è stato lanciato dalla lista di comandi stessa.

NB: poiché la sequenza condizionale e le espressioni condizionali contengono dei comandi semplici e possono essere terminate senza avere eseguito tutti i comandi, in funzione dei risultati restituiti dai comandi eseguiti in precedenza, può capitare che l'ultimo comando eseguito non sia l'ultimo a destra elencato nella lista.

Compound Commands – Comandi composti

for varname in elencoword ; do list ; done

for ((expr1 ; expr2 ; expr3)) ; do list ; done

if listA ; then listB ; [alif listC ; then listD ;] ... [else listZ ;] fi

while list; do list; done

Sequenze non condizionali → il meta carattere ";" viene utilizzato per eseguire due o più comandi in sequenza ed indica la fine degli argomenti passati a ciascun comandi riga di comando

date : ls /usr/giovanni/ ;pwd

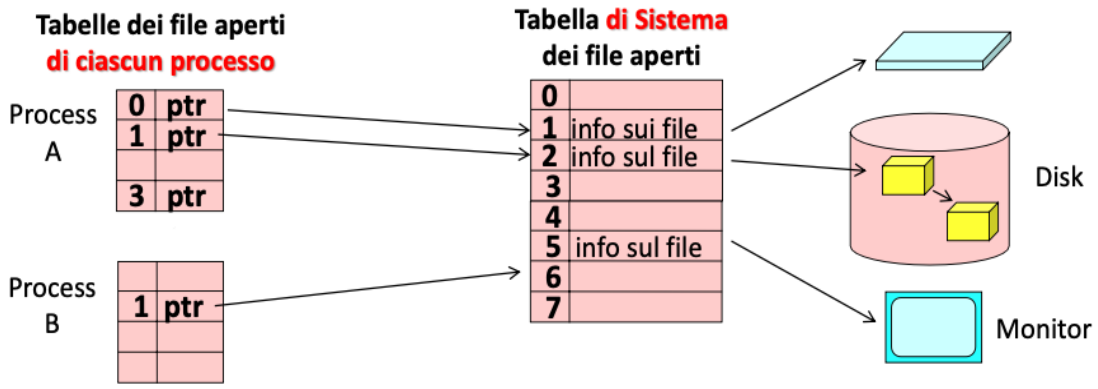
Sequenza condizionali → il metacarattere "||" viene utilizzato per eseguire due comandi in sequenza, ma il secondo viene eseguito solo se il primo termina con un exit code diverso da 0 (failure)

"&&" → viene utilizzato per eseguire due comandi in sequenza, ma il secondo viene eseguito solo se il primo termina con un exit code uguale a 0 (success).

File **descriptor**

È un'astrazione per permettere l'accesso ai file. Ognuno è rappresentato da un integer. Il concetto di file descriptor esiste sia usando il linguaggio C ed anche in bash, perché è un concetto fornito dal sistema fornito.

Ogni processo ha la propria file descriptor table che contiene le informazioni sui file attualmente utilizzati dal processo stesso dal processo stesso. Questa table contiene un file descriptor per ciascun file usato dal processo. Per ciascun file descriptor, la tabella del processo punta ad una tabella di sistema che contiene le informazioni su tutti i file attualmente aperti dal processo stesso



Stream di I/O predefiniti dei processi

Quando un programma entra in esecuzione l'ambiente del sistema operativo si incarica di aprire 3 flussi di dati standard:

1- **Standard Input** (stdin) → serve per l'input normale. Identificato con una costante =0.

echo "stringa"
printf "%f %.6s" 3.99999 piripicchio

2- **Standard Output** (stdout)→ serve per l'output normale. Identificato da 1.

ls filechenonesiste

produce

"ls: cannot access filechenonesiste: no such file or directory"

3- **Standard Error** (stderr) → serve per l'output che serve a comunicare messaggi di errore all'utente. Identificato con 2.

read nomevar

Una shell B figlia di una shell padre A eredita una copia della tabella dei file aperti del padre; quindi, padre e figlio leggono e scrivono sugli stessi stream. Vengono ereditati gli stream predefiniti ed anche tutti gli stream.

Comand substitution

Dobbiamo sostituire a run time, in uno script, la riga di comando di un programma con l'output su stdout prodotto dall'esecuzione del programma stesso.

OUT= `./primo.exe`

echo "l'output del processo è ${OUT}

" " → double quote per escape wildcards e spazi (quoting parziale di stringhe). Impedisce di usare ; come separatore di comandi, impedisce la sostituzione di wildcards ma permette di sostituire le variabili con il loro contenuto e permette l'esecuzione di comandi.

' '→ single quote escape wildcards, command substitution, variable substitution. Impedisce di usare ; come separatore di comandi, impedisce la sostituzione di wildcards, impedisce di sostituire le variabili con il loro contenuto e impedisce l'esecuzione di comandi.

!! la bash processa l'output prodotto dalla command substitution !!

Espressioni condizionali

Sono dei particolari comandi che valutano alcune condizioni e restituiscono un exit status di valore 0 per indicare la verità dell'espressione o di valore diverso da zero per indicarne la falsità.

Ciascuna espressione condizionale si scrive mettendo le condizioni da valutare tra **doppie parentesi** quadre [[…]]. Le condizioni che possono essere inserite in una espressione condizionale sono condizioni create appositamente, e NON possono essere comandi.

All'interno le condizioni possono essere composte tra loro mediante degli operatori logici !, &&, || o mediante altre parentesi per stabilire l'ordine di valutazione.

La bash dentro le espressioni condizionale [[ ]] effettua solo le interpretazioni:

- variable expansion
- arithmetic expansion con $(( )) ma non (( ))
- command substitution
- process substitution and
- quote removal, ma solo negli operandi

Sono disabilitate le interpretazioni di:

- word splitting
- brace expansion
- pathname expansion

Gli operatori di condizione per essere riconosciuti e valutati correttamente devono essere NON quotati.

Dentro una espressione condizionale posso:

- effettuare confronti aritmetici con **operatori -eq -ne -le lt -ge -gt**
- verificare condizioni non aritmetiche che riguardano stringhe con **== = != < <= > >=**
- verificare se delle stringhe sono vuote con operatori -z -n
- verificare condizioni sui file con operatori **-d -e -f -h -r -s -t -w -x -O -G -L**
- confrontare le date di ultima modifica di due files con **-nt -ot**

NON posso eseguire assegnamenti a variabili.

Operatore	Significato	Esempio
-eq	Equal to	5 -eq 5 → vero
-ne	Not equal to	5 -ne 3 → vero
-le	Less than or equal to	3 -lt 5 → vero
-lt	Less than	5 -le 5 → vero
-ge	Greater than or equal to	7 -gt 5 → vero
-gt	Greater than	5 -ge 5 → vero

Esempi di espressioni corrette o sbagliate

[[ls /]] **syntax error**

[[$((**"11" > "2"**))]] **Fraintende. produce exit status 0 (true) sempre**

[[(("11" > "2"))]] **Fraintende**. produce exit status 1 (false) perché le tonde parentesi non valutano aritmeticamente e la valutazione è lessicografica (primo 1 a sinistra precede primo 2 a destra).

*[[**-e** /usr/include/stdio.h]]* OK. Esiste il file /usr/include/stdio.h ? true va bene

[["-e /usr/include/stdio.h"]] **Fraintende**, restituisce true ma per sbaglio, infatti...

[["-e /usr/include/stdio"]] **Fraintende**, restituisce true ma stdio non esiste

[["-e" /usr/include/stdio.h]] **syntax error**

Gli operatori (( )) e $(( )) valutano aritmeticamente tutto il comando che specifico all'interno, quindi, l'operatore di valutazione aritmetica:

- può contenere solo espressioni valutabili aritmeticamente
- può comporre espressioni aritmetiche mediante operatori logici &&, || e !
- non può contenere espressioni condizionali
- non può contenere comandi

SI (($VAR * (3 + $MUL) < 4 && 7 > $VAR))
NO (($VAR < 3 + $MUL && [[$NUM -lt 5]])) errore sintattico

CI sono tre operatori per le espressioni condizionali, al cui interno specificare gli operatori per Condizioni di file, Confronto tra stringhe, Confronto aritmetico.

L'operatore [[ ]] permette di comporre tra loro più condizioni, utilizzando gli operatori logici. Al suo interno si possono usare parentesi tonde per raggruppare espressioni e modificare la precedenza degli operatori e posso andare a capo proseguendo le espressioni.

Diversamente gli operatori **[ ]** e ***test*** permettono di comporre tra loro più condizioni utilizzando però operatori logici diversi, !(negazione), -a (and), -o (or). Con questi operatori, NON posso usare parentesi tonde per raggruppare espressioni e modificare la precedenza degli operatori e NON posso andare a capo se non usando il \ a fine riga.

Le precedenze degli operatori logici decrescono in questo ordine: !, &&, ||

Operatori per verificare condizioni sui file

Operatore	Significato
-d file	True if file exists and is a directory
-e file	True if file exists
-f file	True if file exists and is a regular file
-h file	True if file exists and is a symbolic link
-r file	True if file exists and is readable
-s file	True if file exists and has a size greater than zero
-t fd	True if file descriptor fd is open and refers to a terminal
-w file	True if file exists and is writable
-x file	True if file exists and is executable
-O file	True if file exists and is owned by the effective user id
-G file	True if file exists and is owned by effective group id
-L file	True if file exists and is a symbolic link (deprecated)
File1 -nt file2	True if file1 is newer than file2 or file1 exists and file 2 not
File1 -ot file2	True if file1 is older than file2, or if file2 exists and file1 not

Operatori per verificare condizioni su stringhe, aritmetiche e altre varie

Operatore	Significato
-o optname	True if shell option optname is enabled.
Operazione su stringhe	
-z string	True if the lenght of string is zero
-n string	True if lenght of string is non-zero
String1 == string2/string1 = string2	True if the strings are equal
String1 != string2	True if the strings are not equal
String1 < string2	True if string1 sorts before string2 lexicographically
String1 > string2	True if string1 sorts after string2 lexicographically

Quoting $'stringa'

Parole aventi forma $'charsequence' sono trattate in modo speciale. La charsequence può contenere backslash-escaped characters.
La parola viene espansa in una stringa single-quote, cioè perde il $ all'inizio ma mantiene i due ' all'inizio e alla fine.
Le **backslash-escaped characters**, se presenti, sono sostituite come specificato nello standard ANSI C

\a alert (bell)	\b backspace
\e \E an escape character	
\f form feed	\n new line
\r carriage return	\t horizontal tab
\v vertical tab	\\ backslash
\' single quote	\" double quote

\nnn the eight-bit character whose value is the octal value nnn (one to three digits)
\xHH the eight-bit character whose value is the hexadecimal value HH (one or two hex digits)
\cx a control-x character, as if the dollar sign had not been present.

Variabile IFS → contiene i caratteri che fungono da separatori delle parole negli elenchi. Di default contiene uno spazio bianco, un tab, un newline (a capo).
Se devo lanciare dei comandi in cui devo trattare dei nomi di file che contengono degli spazi bianche, se non posso fare diversamente devo:

- usare gli elenchi separati da newline o tab
- togliere dai separatori lo spazio bianco

Esempio importante:
IFS=$' \t\n'
eseguirò il comando che dovevo lanciare e dopo rimetterò lo IFS come era prima.
Es: directory che contiene due files, "aa bb.txt " e "aa cc.txt"
Vedere che succede se lancio

for name in `ls aa*` ; do echo ${name} ; done

Visualizzo

aa
bb.txt
aa
cc.txt

TRUCCO OSCENO MA FUNZIONA

```
OLDIFS=${IFS}
IFS=$'\t\n'
for name in `ls -1 aa*` ; do echo ${name}; done
IFS=${OLDIFS}
```

I separatori contenuti in IFS sono utilizzati per individuare in modo specifico le separazioni tra gli elenchi di nomi. I separatori usati per individuare la separazione tra parole chiave del linguaggio sono invece non modificabili e sono sempre gli spazi bianchi, i tab e le andate a capo.

Lettura da standard input – il comando **read**

Uno script per leggere dallo standard input delle sequenze di caratteri utilizza il comando read.

Il comando riceve la sequenza di caratteri digitate da tastiera fino alla pressione del tasto INVIO e mette i caratteri ricevuti in una variabile che viene passata come argomento alla read stessa.

Se invece lo standard input è stato ridiretto da un file, allora la read legge una riga di quel file ed una eventuale read successiva legge la riga successiva.

Il return della read è:

- 0 se non si arriva a fine file e viene letto qualcosa
- >0 se si arriva a fine file

Può accadere che la lettura incontri la fine del file senza incontrare l'andata a capo, in quel caso la read mette, nella variabile, tutti i caratteri non ancora letti che precedono la fine del file, e poi restituisce valore <0 per indicare che il file è stato usato fino alla fine.

Visto che quando la read dice che è arrivata a fine file non è detto che la stringa contenga dei caratteri bisogna controllare se nella variabile letta c'è qualcosa dentro.

La stringa ${#VAR} viene interpreta dalla bash sostituendola con la stringa di cifre che rappresenta il numero di carattere id cui la variabile VAR, se esiste, è formata.

Qualche esempio di controllo sulle read:

1

```
while read RIGA; if (( $?==0 )); then true; elif (( ${#RIGA} != 0)); then true; else false; fi ;
do echo read "${RIGA}" ; done
```

2

```
while read RIGA ; [[ $? == 0 || ${RIGA} != "" ]] ; do echo "read ${RIGA}" ; done
```

3

```
while read RIGA ; [[ $? -eq 0 || ${#RIGA} > 0 ]] ; do echo "read ${RIGA}" ; done
```

4

```
while read RIGA ; [[ $? == 0 ]] || [[ -n ${RIGA} ]] ; do echo "read ${RIGA}" ; done
```

Se al comando read vengono passati come argomenti una o più variabili, allora il contenuto della variabile IFS viene usato per separare la linea letta in parole e per assegnare alle variabili passate le parole lette.

i) Se la riga letta contiene più parole del numero delle variabili passate alla read, allora ciascuna variabile viene riempita con una parola estratta dalla linea letta, tranne l'ultima variabile che riceve tutto quello che resta della linea

ii) Se invece la riga letta contiene meno parole del numero di variabili passate alla read, allora solo le prime variabili ricevono una parola, alle altre è assegnato il valore vuoto.

Opzione -N → permette di specificare il numero di caratteri che devono essere letti.

Se durante la lettura viene raggiunta la fine riga, la read termina anche se non ho letto tutti i caratteri richiesti. In tal caso, nella variabile trovo meno caratteri di quelli che avevo richiesto. La read successiva partirà dalla riga successiva.

Stream di I/O – **Apertura, File Descriptor, Accesso**

È possibile aprire un altro file da disco, ottenere un altro file descriptor che lo rappresenta ed utilizzare quel nuovo file descriptor per accedere al file aperto.

Modo apertura	Utente sceglie fd	Sistema sceglie fd libero e lo inserisce in variabile
Solo lettura	exec n< PercorsoFile	exec {NomeVar}< PercorsoFile
Scrittura	exec n> PercorsoFile	exec {NomeVar}> PercorsoFile
Aggiunta in coda	exec n>> PercorsoFile	exec {NomeVar}>> PercorsoFile
Lettura e Scrittura	exec n<> PercorsoFile	exec {NomeVar} <> PercorsoFile

Nel comando read specifico l'opzione -u seguita dal file descriptor del file aperto per indicare al comandò read da quale file aperto deve essere effettuata la lettura.

Variabile $$ → mi dice il process identifier della shell corrente.

Nella directory /proc/ esiste una sotto-directory per ciascun processo in esecuzione del processo stesso.

ls /proc/$$

nella sotto-directory propria di ciascun processo, esiste una sotto-directory fd in cui sono presenti dei file speciali che sono i file aperti da quel processo.

Qualunque sia il modo di apertura con cui ho aperto un file, la chiusura di un file può essere effettuata utilizzando il comando exec con il seguente operatore:

exec n>&-

n=file descriptor da chiudere

NB: se dopo la chiusura del file utilizzo il file descriptor, al bash produce un errore.

Ridirezionamenti di Stream di I/O

Quando lanciamo un comando o processo o script all'interno di una bash il processo figlio ottiene una copia di tutti i file descriptor del padre. il padre può decider di cambiare gli stream da far usare al figlio associando, a ciascun file descriptor da passare al figlio, uno stream diverso. In tal caso, i file descriptor passati al figlio avranno lo stesso identificatore numerico di quelli del padre, ma saranno associati a stream diversi.

Un processo può associare un proprio file descriptor ad un altro stream, i file descriptor ridirezionati mantengono il loro valore ma sono associati a stream diversi. Da quel momento, il processo usando i vecchi file descriptor accederà ai nuovi stream.

Comando **exec**→comando utilizzato per effetuare apertura, chiusura e ridirezionamento.

Ridirezionamenti:

1) < → ricevere input da file
2) > →mandare std output verso file eliminando il vecchio contenuto del file
3) >> →mandare std output verso file aggiungendolo al vecchio contenuto del file
4) | → ridirigere output di un programma nell'input di un altro programma

L'utente può utilizzare lo standard input di un programma per dare input non solo da tastiera ma anche da file, ridirezionando il contenuto di un file sullo standard input del programma, al momento dell'ordine di esecuzione del programma.

program<nome_file_input

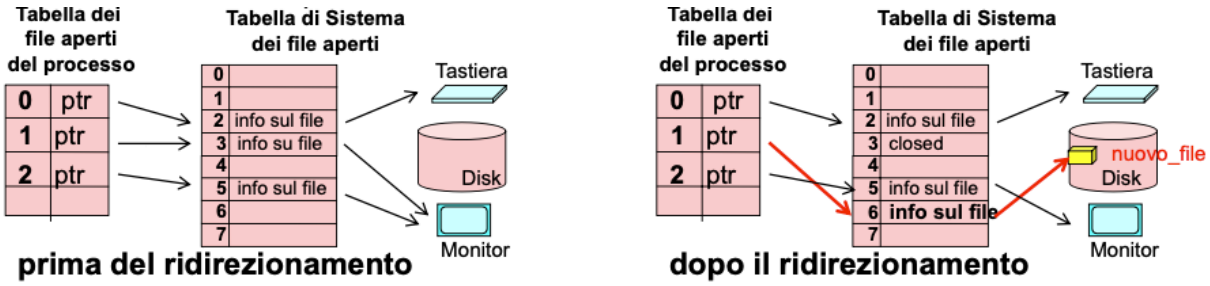
il programma vedrà il contenuto del file nome_file_input come se venisse digitato da tastiera.

Ctrl+D→ serve per emulare la fine di input da tastiera perché se non faccio il ridirezionamento dell'input e fornisco l'input da tastiera, non incontro mai una fine del file di input.

Analogamente lo standard output di un programma può essere ridirezionato su un file, su cui sarà scritto tutto l'output del programma invece che a video

program> nome_file_output

In nome_file_output troveremo i caratteri che il programma voleva mandare a video. Il precedente contenuto del file verrà perso ed alla fine dell'esecuzione del programma del file troveremo solo l'output generato dal programma stesso.



Il ridirezionamento dello standard output può essere fatto senza eliminare il vecchio contenuto del file di output bensì mantenendo il vecchio contenuto ed aggiungendo in coda al file l'output del programma

program >>nome_file_output

i due ridirezionamenti (input e output) possono essere fatti contemporaneamente

program < nome_file_input> nome_file_output

o analogamente

program >nome_file_output < nome_file_input

inoltre, su bash si può ridirezionare standard output e standard error su due diversi file, sovrascrivendo il vecchio contenuto:

program &> nome_file_error_output

Si può ridirezionare standard output e standard error su due diversi file, sovrascrivendo il vecchio contenuto:

program 2> nome_file_error > nome_file_output

Se una bash ha uno stream (un file aperto) e quello stream è identificato da un file descriptor di valore N, allora, è possibile ridirigere su un altro file lo stream indicato da quel file descriptor, usando una sintassi che specifica il valore intero N di quel file descriptor.

N> NomeFileTarget → ridireziona il file descriptor N sul file con come NomeFileTarget, se si omette N si intende standard output.

<N NomeFileSource → ridirezione il file con nome NomeFileSource sul file descriptor N del programma specificato alla sinistra dell'operatore.

L'operatore **|** - PIPE

È possibile far eseguire contemporaneamente due processi mandando lo standard output del primo nello standard input del secondo, mediante l0operatore pipe |

program1 | program2

I due processi eseguono in contemporanea. Quando il primo processo termina o chiude il suo standard output, il secondo processo vede chiudersi il proprio standard input.

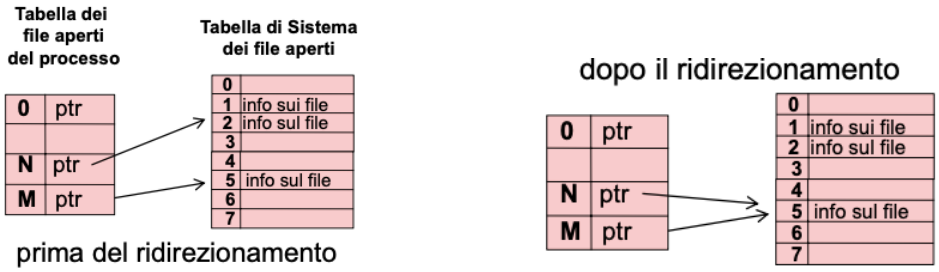
In uno script dove compare una pipe, Se un comando a destra o a sinistra della pipe è un comando composto (dei loop) oppure è uno script, allora quel comando esegue in una shell bash figlia.

Operatore **>&** - ridirezionamento tra file descriptor

Se una bash ha due stream (due file aperti) entrambi di output (o entrambi di input), e ciascun stream è identificato da un file descriptor di valore n ed M rispettivamente, allora è possibile ridirigere lo stream N sullo stream M mediante l'operatore

N >& M

Dopo ciò, quello che scrivo su N viene scritto su M. lo stream N mantiene il valore del suo file descriptor ma, nella tabella dei file aperti del processo, il suo puntatore alla tabella di sistema viene sostituito da una copia del puntatore dello stream. Da quel moentno esisteranno due file descriptor N ed M di valore diverso ma che puntano allo stesso file aperto.



cat out.txt 2>1

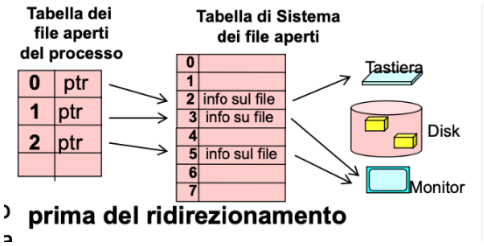
qui 1 è visto come il nome di un file si ridirige lo stderr di cat sul file 1 non sullo stdout di cat

cat out.txt 2>&1

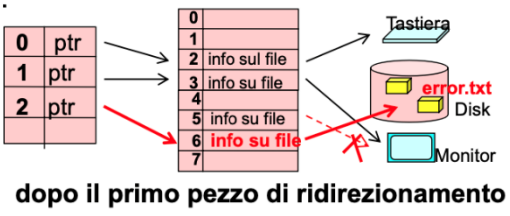
qui 1 è visto come file descriptor di un file aperto si ridirige lo stderr di cat sullo stdout di cat.

È importante l'ordine con cui compongo gli operatori di ridirezionamento di file descriptor. Sono eseguiti da sinistra verso destra.

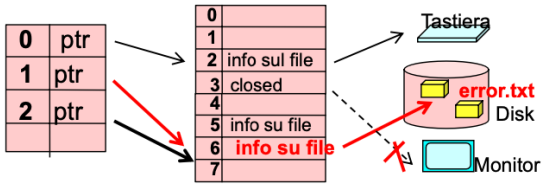
ls pippo.txt 2>error.txt 1>&2



2>error.txt → apre il file error.txt e mette le info sul file aperto nella nuova posizione 6 della tabella di sistema. Poi copia nello spazio occupato dal puntatore del file descriptor 2 l'indirizzo della riga nuova della tabella di sistema. Poi copia nello spazio occupato dal puntatore del file descriptor 2 l'indirizzo della riga nuova della tabella di sistema. Quindi ora fd 2 punta al file error.txt mentre fd 1 scrive ancora su video.



1>&2 → copia il puntatore del file descriptor 2 nello spazio occupato dal puntatore del file descriptor 1. Quindi ora sia fd1 che fd2 puntano alla stessa riga della tabella di sistema ed entrambi scrivono sul file error.txt



dopo il secondo pezzo di ridirezionamento

Voglio ridirigere il solo stderr del comando ls, usandolo come stdin del comando grep e buttando via lo stdout di ls stesso.

ls nonesiste.txt 2>&1 > /dev/null | grep such

1) ridireziona stderr di ls su stdout di ls,
2) ridireziona stdout di ls su /dev/null facendo buttare via l'output
3) passa il nuovo stdout di ls come stdin di grep
4) quindi il vecchio stderr di ls finisce nello stdin di grep

ls nonesiste.txt > /dev/null 2>&1 | grep such

1) ridireziona stdout di ls su /dev/null
2) ridireziona stderr di ls su stdout di ls che ora punta a /dev/null
3) quindi entrambi stdout e stderr di ls finiscono nel device /dev/null
4) Al comando grep non arriva niente di niente.

Operatore |&

ls pippo.txt |& grep such

Esiste un operatore apposito che rappresenta una scorciatoia dell'operatore composito 2>&1 | che fa la stessa cosa ed è |&.

program1 |& program2

con l'operatore composito |& i due programmi specificati partono assieme e l'output del primo programma sono ridirezionati insieme nell'input del second programma.

C'è una notevole differenza nelle tempistiche di esecuzione dei programmi in base a se sono separati da ; o |. Con il separatore ";" io faccio eseguire i diversi programmi uno dopo l'altro, cioè prima che parta il secondo programma deve finire il primo e così via. Inoltre, l'output di un programma non viene ridirezionato nell'input del programma successivo. Invece, con la | i programmi specificati partono assieme e l'output di un programma viene ridirezionato nell'input del programma successivo.

Auto-Ridirezionamento

Una bash può ridirigere un proprio stream aperto verso un nuovo file descriptor. Da quel momento la bash vedrà il nuovo stream mediante il vecchio file descriptor. Gli autodirezionamenti sono effettuati con i consueti operatori applicati ai file descriptor ma usati come argomenti della funzione *exec*.

Ridirezionamento per blocchi di comandi

Per i costrutti linguistici bash che eseguono blocchi di comandi (for, while, if then else) è possibile applicare un ridirezionamento univo per tutti i comandi del blocco di comandi, compresi i comandi eseguiti nella condizione.

È possibile applicare un ridirezionamento anche per flussi di input.

Ridirezionamento dell'input passato a riga di comando → **Here Documents <<**

La word che segue il metacarattere **"<<"** viene utilizzata come delimitatore finale dell'input da passare al comando a sinistra del metacarattere. Viene ridirezionato nell'input tutto quello che compare tra la word esplicitata dopo il metacarattere << e fino a dove quella word compare ancora all'inizio di una riga. NON vengono espansi eventuali metacaratteri presenti nella word.

La sua utilità sta nel poter passare input come se provenisse da un file ma senza dover scrivere un file.

Esempio:

while read A B C ; do echo $B ; done ***<<FINE***

uno due tre quattro

alfa beta gamma

gatto cane

FINE

echo ciao

produce in output:

due

beta

cane

ciao

Raggruppamento di **comandi**

Ci sono casi in cui abbiamo necessità che alcuni comandi vengano eseguiti in una subshell, i quali utilizzano gli stessi stdin, stdout, stderr. Se si racchiude una sequenza di comandi tra parentesi tonde, allora in esecuzione viene creata una subshell per eseguire quella sequenza di comandi.

Il fatto di eseguire dei comandi in una shell figlia fa in modo che queste ereditano stdin, stdout, stderr dalla shell corrente e che i processi lanciati dalla shell figlia usano stdin, stdout, stderr della shell corrente e i comandi eseguiti fanno analogamente.

Il risultato restituito dalla subshell è il risultato dell'ultimo comando eseguito nella subshell, cioè l'ultimo comando eseguito tra quello nelle parentesi tonde. Questo mi permette di trattare/ridirigere input e output di tutti i comandi dentro le parentesi tonde come se fossero un solo comando.

Esempio concatenazione stdout

(cat out.txt ; cat pippo.txt) | grep stringa

Il comando grep legge, come se provenissero da tastiera, le righe di entrambi i file concatenati.

Esempio concatenazione stdout e stderr

(cat out.txt ; cat pippo.txt) |& grep stringa

Esempio concatenazione stdin

cat out.txt | (read RIGA1 ; usa RIGA1 ; read RIGA2 ; usa RIGA2)

I due comandi read leggono uno la prima l'altro la seconda riga prodotte da cat.

GNU Coreutils

I comandi genericamente lavorano con i file di testo che lavorano con delle righe e mettendo sul stdout una loro versione modificata. I principali comandi sono head, tail, sed, cut, cat, grep, tee. Tutti questi comandi tipicamente possono avere come argomento un nome di file da cui leggere l'input altrimenti possono leggere input da stdin se non c'è l'argomento nomefile o il "-" alla fine.

Esempio:

(head -n 2 dati.txt ; tail -n 3 dati.txt) | grep AL | sed 's/AL/CUF/g' | cut -b 4- > output.txt

head -n 2 →chiedo le prime due righe del file dati.txt in output

tail -n 3 → chiedo le ultime tre righe del file dati.txt in output

metto questi due comandi dentro le parentesi tonde in modo che l'output venga concatenato.

grep AL→ seleziona tra tutte le righe scelte solo quelle con le lettere AL

sed 's/AL/CUF/g' → s significa sostituisci quello che c'è tra il primo / e il secondo / con quello che c'è dopo, g significa che deve sostituire ogni occorrenza.

cut -b 4- → elimina i primi 3 caratteri, mettendo in output solo dal 4 in poi. -b sta per byte, ovvero tratta le righe che hai letto come byte quindi carattere per carattere. Può essere utilizzata anche una virgola per far stampare due intervalli, ad esempio cut -b -6,8-9 nomefile.txt.

Il comando **tail**

Fa vedere le ultime righe di un file che io passo il nome come argomento.

tail -n 3 nomefile

l'opzione -n NUM→ indica quante righe farmi vedere, NUM è il numero di righe da far vedere.

Se non dico il nome del file allora lui aspetta e legge dallo stdin finché non l'utente preme Ctrl +D. può leggere le cose anche tramite ridirezionamento

cat nomefile | tail -n 3

in questo caso prende input dal ridirezionamento della pipe.

tail -f nomefile.txt

l'opzione **-f** →fa in modo che non termini il comando dopo che ha letto le ultime righe, rimane in attesa che qualcuno aggiunga delle nuove righe nel file per stamparle.

Il comando **tee**

Fa una duplicazione dell'output, stampa a video nello stdout e poi salva l'output nel file che gli viene passato come parametro

cat nomefile | tee nuovofile.txt

il contenuto di nomefile andrà stampato in stdout da tee che ne salva anche il contenuto in nuovofile.txt

Il comando **sed** - stream editor

Può sostituite la prima occorrenza di una sequenza di caratteri con un'altra

sed 's/togli/metti//' nomefile

Può rimuovere il primo tra i caratteri che trova in ciascuna riga del file

sed 's/a//' nomefile

Può rimuovere il carattere in prima posizione di ogni linea che si riceve dallo stdin. ^ significa inizio linea, "." Significa carattere qualunque.

sed 's/^.//'

Può rimuovere l'ultimo carattere di ogni linea ricevuta dallo stdin "$" significa fine linea

sed 's.$//'

Posso eseguire più rimozioni concatenate con ;

sed 's/.//;s/.$//'

Può rimuovere i primi tre caratteri ad inizio linea

sed 's/…//'

Rimuovere i primi quattro caratteri ad inizio linea

sed -r's/.{4}//'

opzione **-i /--in-place** → vuol dire che il comando se che scrivo viene applicato proprio al file che do come parametro, senza stampare a video niente.

Il comando **head**

Fa vedere le prime righe di un file di testo o dell'output di un comando.

head [opzioni] nomefile

il comando **cut**

viene utilizzato per estrarre specifiche porzioni di testo da file o input standard

cut [opzioni] nome file

Consente di estrarre:

- Caratteri speciali: tramite posizione
- Campi specifici: tramite delimitatori
- Intervalli di dati

opzione **-c** → estrae specifici caratteri tramite la posizione.

cut -c 1-5 nomefile

opzione **-b** → estrae specifici byte da ogni riga di file.

Processi Identifier di Shell Bash → **$$ e $BASHPID**

ogni processo è identificato da un identificatore numerico univoco, il PID, anche le shell in esecuzione ne posseggono uno.

$$ →variabile usata nel caso in cui occorra conoscere il PID di una shell in esecuzione. Se raggruppo dei comandi facendoli eseguire in una subshell (), allora dentro le ( ) la variabile $$ mi fa vedere il PID della bash più esterna e non della subshell.

$BASHPID → contiene il PID della bash corrente, anche se questa è creata con ( ).

Terminale di controllo

Processo → è un insieme di thread di esecuzione operanti all'interno di un contesto, il quale coprendo uno spazio di indirizzamento in memoria ed una tabella dei descrittori di file aperti di quel processo.

Gruppo di processi → un processo può lanciare l'esecuzione di altri processi: questi altri processi appartengono allo stesso gruppo di processi del processo padre, a meno che non si svolgano azioni che modificano il gruppo di appartenenza

Terminale di controllo → Quando si accende a una macchina Unix lo si fa attraverso un terminale che può essere i) fisico: tastiera e monitor, ii) remoto: tramite ssh

A questo terminale di controllo viene legata la bash interattiva che è il processo principale dal quale nascono tutti gli altri processi figli. La caratteristica dei processi a partire dalla bash è che il sistema operativo tiene traccia del terminale di controllo al quale è legata. Il sistema operativo monitora il funzionamento di tutti i terminali di controllo e se si accorge che è morto allora ammazza tutti i processi figli che gli appartengono. Questa operazione di terminazione viene realizzata mandando ad ogni processo un segnale di terminazione detto ***SIGHUP (***"signal hang up").

nohup→ comando che serve per lanciare un programma che voglio che sia sganciato dal terminale.

Processi in foreground→ processi che controllano il terminale fino a quando sono morti. Sono i processi che noi vediamo in esecuzione direttamente sul terminale

Processi in background→ sono processi che vengono lanciati e la shell continua ad avere il controllo del terminale, ovvero leggere da tastiera e mangiare stdout e stderr. Questi processi lasciano alla shell la possibilità di interagire ma anche i processi in background possono usare stdin, stdout, stderr. Nasce, quindi, un problema legato al fatto che sia la shell che il processo in background posso scrivere, leggere e mandare errori sullo standard, ciò significa che il mio processo lanciato in background può essere che rubi dei caratteri destinati alla shell. Quando il mio processo in background è in esecuzione e chiuso la shell di controllo, il processo viene concluso. I processi in background vengono chiamati jobs. Il jobs control permette di portare i processi da back ground a foreground e viceversa

Elenco di comandi/operatori utilizzabili con i processi:

& →lancia un processo in background

^Z → sospende un processo in foreground

^C→ termina un processo in foreground

bg →riprende l'esecuzione in background di un processo sospeso

jobs → comando che produce una lista numerata dei processi in background o sospesi nella shell corrente

fg %n →porta in foreground un processo sospeso

kill → elimina il processo specificato dal proprio identificatore pid oppure specificato dal numero del job

disown –[ar] jobs → sgancia il job dalla shell interattiva che lo ha lanciato. L'opzione -r fa in modo che tutti i job in running vengano sganciati. L'opzione -a sgancia tutti i job in running e sospesi

Quando lancio un processo direttamente in back ground allora il PID del processo viene messo nella variabile $! Della shell che lanciato quel processo. La shell mantiene quel valore fino a che la stessa shell non lancia in background un altro processo.

Attesa terminazione di un processo figlio di shell – **wait**

Supponiamo che una shell bash lanci in esecuzione un processo e lo metta in background, poi la shell può fare qualcosa mentre il programma è in esecuzione, e infine la shell vuole aspettare che il programma lanciato termini, per fare ciò si usa il comando wait.

Al comando passo come argomento il *PID del processo* di cui attendere la terminazione. Ma posso passargli i) un elenco di PID e in tal caso aspetta la terminazione di tutti processi indicati e restituisce come risultato il risultato dell'ultimo processo nella lista di PID, ii) un elenco di jobs, in questo caso aspetta la terminazione di tutti i jobs, iii) nessun argomento, quindi, aspetta la terminazione di tutti i processi figli della shell in cui il comando wait è stato lanciato e restituisce exit status 0.

Il comando wait può essere invocato SOLO dalla shell che ha lanciato il processo di cui vogliamo attendere la terminazione. Se l'attesa per la terminazione viene invocata da una shell diversa, il comando wait termina subito e restituisce 127. Il comando wait come exit status restituisce l'exit status restituito dal processo figlio specificato dal PID.

Processi zombie

Quando un processo figlio termina molte delle sue strutture dati vengono rilasciate ma non tutte, in particolare quando accade il PID del figlio viene usato ancora per identificare il processo morto, e quindi quel PID non può essere riutilizzato da un altro processo. Viene rilasciato soltanto quando il padre lancia il comando wait. Finché il padre non esegue quel comando il processo figlio è morto ma rimane ancora nel mio sistema per quello si chiama zombie. Se il processo padre non fa ma la wait, i figli rimangono sempre come zombie, ma quando il padre termina le esecuzioni i figli vengono adottati dal processo init ed è quel processo che ogni tanto fa la wait per i processi adottati.

Se nessuno mai facesse le wait dopo un po' non si potrebbero fare più processi perché non ci sarebbero PID per identificarli.

Il comando **find**

opzione -maxdepth

opzione -mindepth

opzione -type d → fa vedere solo delle directory

opzione -type f → fa vedere solo dei file

opzione -exec '{ }' → per ciascuno dei file che trovi esegui il comando che segue. '{ }' significa il nome di ciascun file che viene trovato dai comandi precedenti, in questo caso dalla find. Alla fine devo inserire \; per far capire al comando find che lì termina.