# Bash

## 1. Interfaccia utente a linea di comando

L'interfaccia utente a linea di comando permette di impartire ordini al OS sotto forma di sequenze di caratteri alfa-numerici digitati sulla tastiera.

L'**interprete dei comandi** (o anche **Shell di comandi**) è un programma eseguibile che si occupa di un singolo terminale. Gli ordini che un terminale può ricevere possono essere di due tipi:

* **Comandi**: sono ordini la cui implementazione è contenuta nel file eseguibile della Shell stessa (ad esempio il comando **cd**).
* **Eseguibili**: sono ordini la cui implementazione è contenuta fuori della Shell, cioè in altri file.

Le Shell cambiano in base al OS:

* **Unix**, **Linux**: bash;
* **Mac**: bash;
* **Windows**: Shell DOS.

## 2. Librerie e System calls

Le system calls forniscono ai processi i servizi offerti dal OS. Il modo in cui sono realizzate e vengono chiamate in base alla CPU e al OS. Normalmente i programmi applicativi invocano funzioni contenute in librerie di sistema ed utilizzate dai compilatori per generare eseguibili. L’insieme di tali funzioni di libreria rappresentano le **API** (**Application** **Programming** **Interface**) cioè l’interfaccia che il sistema operativo offre ai programmi di alto livello.

**Cambiando OS** oppure cambiando CPU, i nomi delle funzioni delle API rimangono gli stessi ma **cambia il modo di invocare le system calls**, quindi cambia l’implementazione delle funzioni delle API per adattarle alle system calls e all’hardware.

Alcune API diffuse sono:

* Win32 API per Windows;
* **POSIX API** per sistemi POSIX-based (tutte le versioni di **UNIX**, **Linux**, **Mac OS X**);
* Java API per la Java Virtual Machine.

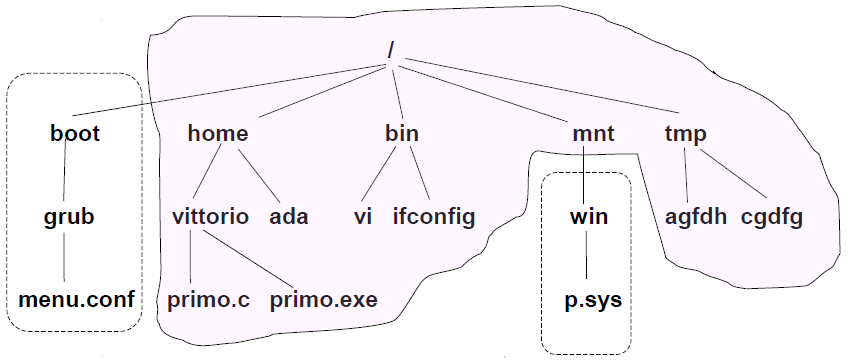
In Linux esiste un comando che permette di vedere quali system calls sono usate da un programma in esecuzione: **strace** *comando*.

## 3. File system

Il **file** **system** è l’organizzazione del disco rigido che permette di contenere i file e le loro informazioni. Lo spazio di ciascun disco rigido è suddiviso in una o più parti, dette **partizioni**. Le partizioni contengono:

* File
* Directory

In Windows le partizioni del disco sono logicamente separate e ciascuna è indicata da una lettera (C, D, …). In Linux/Mac invece le partizioni sono viste come collegate tra loro. Esiste una partizione principale il cui nome è “**/**” (slash).



Esempio di strutturazione in directories e files delle partizioni di un file system Linux.

Per ciascun utente esiste una directory in cui sono contenuti i files personali di quell’utente, chiamata **home dell’utente**.

Quando si accede al terminale a riga di comando, esso stabilisce la posizione logica attuale dell’utente all’interno del file system, collocandolo inizialmente nella propria home directory. La directory in cui l’utente si trova logicamente si chiama **directory corrente**. Per sapere in quale percorso ci si trova si può utilizzare il comando **pwd**.

Per spostarsi logicamente in un percorso specifico si utilizza il comando **cd**.

## 4. Permessi per utenti e gruppi

Nei sistemi Linux esistono le astrazioni di utente (**user**) e gruppo di utenti (**group**). Un **user** è caratterizzato da una stringa chiamata **username** e da un identificatore **userID**. Un **group** è caratterizzato da una stringa chiamata **groupname** e da un identificatore **groupID**.

Ogni file ha un proprietario e un gruppo del proprietario. Il proprietario/creatore può cambiare il proprietario del file con il comando **chown** *nuovoproprietario nomefile*.

Ciascun file mantiene diversi diritti di lettura, scrittura ed esecuzione assegnati al proprietario, al gruppo del proprietario e a tutti gli altri:

* Lettura: valore **4**, simbolo **r**;
* Scrittura: valore **2**, simbolo **w**;
* Esecuzione: valore **1**, simbolo **x**.

Metodo di assegnazione contemporanea di permessi mediante formato numerico:

**chmod** **ugo** *nomefile*

Dove al posti di “**u**” ci va il valore dei permessi per l’**user proprietario**, al posto di “**g**” ci va il valore dei permessi per **gruppo**, al posto di “**o**” ci va il valore dei permessi per **others**.

Es: **chmod** **764** ./*file.txt*

* 7 = lettura, scrittura ed esecuzione per user proprietario;
* 6 = lettura e scrittura per gruppo;
* 4 = lettura per others.

Utilizzare il comando **ls -alh** per vedere tutti i permessi di tutti i file contenuti nel percorso corrente.

## 5. Espansioni

La Shell legge ciascuna riga di comando e la interpreta eseguendo alcune operazioni. Alcune di queste sostituiscono (**espandono**) alcune parti della riga di comando con altre, interpretando caratteri speciali. Le espansioni principali sono:

* **Variable expansion** ${var}
* **History expansion** !123
* **Parameter expansion** $1 $?
* **Brace expansion** a{1,2,3}b
* **Tilde expansion** ~/directory
* **Pathname expansion** \* ? […]
* **Arithmetic expansion** $(( ))
* **Command substitution** ` ` $( )
* **Word splitting**
* **Quote removal**

### Variable expansion

La shell dei comandi permette di usare delle variabili, oggetti dotati di un **nome** e di un **valore**. Sintassi di una variabile: **&{***variabile***}**.

Il comando **echo** permette di visualizzare a video la sequenza di caratteri scritti subito dopo il comando e fino al primo carattere di andata a capo.

Es:

**echo** *ciao*

ciao

Ovviamente funziona anche con le variabili.

Es:

NUM=5

**echo** ${NUM}

5

Esiste una variabile d’ambiente, detta **PATH**. Contiene una sequenza di percorsi assoluti nel file system di alcune directory in cui sono contenuti gli eseguibili. I diversi percorsi sono separati dai “**:**” (due punti).

Una **subshell** è una shell (figlia) creata da un’altra shell (padre). Una subshell viene creata in caso di:

* Esecuzione di **comandi raggruppati**;
* Esecuzione di **script**;
* Esecuzione di **processo in background**.

È inoltre possibile eseguire uno script senza creare la subshell in cui lo script dovrebbe essere eseguito, tramite il comando **source** *nomescript*. In questo modo le variabili modificate dallo script sono proprio quelle della shell, quindi quest’ultima vede modificate le proprie variabili.

Ogni shell supporta due tipi di variabili:

* **Variabili locali**: non vengono trasmesse da una shell alle subshell da essa create. Utilizzate per computazioni locali all’interno di uno script;
* **Variabili d’ambiente**: vengono trasmesse dalla shell alle subshell. Viene creata una copia della variabile per la subshell, quindi se viene modificata nella subshell, la variabile originale nella shell padre non cambia.

Per visualizzare l’elenco delle variabili d’ambiente si utilizza il comando **env**.

Per trasformare una variabile locale, già dichiarata, in una variabile d’ambiente, si utilizza il comando **export** *nomevariabile*.

C’è differenza tra una variabile che esiste ma è vuota ed una variabile che non esiste. È possibile eliminare una variabile esistente col comando **unset** *nomevariabile*.

Il comando built-in **set** svolge diversi compiti:

* Lanciato **senza parametri**, visualizza tutte le variabili della shell, sia quelle locali che quelle d’ambiente. Inoltre, visualizza le funzioni implementate nella shell.
* Lanciato **con parametri**:
  + **set -a**: le variabili create o modificate vengono immediatamente fatte diventare variabili d’ambiente e vengono ereditate dalle eventuali shell figlie;
  + **set +a**: le variabili create sono variabili locali. È il modo di default.

### History expansion

La funzionalità history memorizza i comandi lanciati dalla shell e permette di visualizzarli ed eventualmente rilanciarli. Il comando **history** visualizza un elenco numerato dei comandi precedentemente lanciati dalla shell. Se lancio il comando **!***numero* eseguo il comando che nella history è indicato con *numero*.

Il comando built-in **set** svolge diversi compiti anche nel caso della history:

* **set +o history**: disabilita la memorizzazione di ulteriori comandi eseguiti nel file di history;
* **set -o history**: abilita la memorizzazione dei comandi eseguiti mettendoli nel file di history.

Tramite il comando **man** *comando* (es: **man** set) è possibile leggere tutte le possibili opzioni di comportamento della shell che possono essere attivate/disattivate da *comando* (set).

### Parameter expansion

Gli **argomenti** o **parametri** a riga di comando di un 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.

La shell bash si comporta in maniera diversa a seconda di quali argomenti a riga di comando le vengono passati nel momento in cui viene lanciata l’esecuzione:

* **Shell non interattiva**: shell figlia che esegue script, lanciata con argomenti: **-c** *percorsoscript*;
* **Shell interattiva non di login**: quella che si vede all’inizio nella finestra di terminale, lanciata senza argomenti;
* **Shell interattiva di login**: è come la shell non di login, ma inizia chiedendo user e password, lanciata con argomenti: **-l** oppure **–login**.

La bash permette inoltre che in una stessa riga di comando possano essere scritti più comandi. Il carattere “**;**” (punto e virgola) è il separatore tra i comandi, mentre il carattere **“ ”** (doppi apici) delimita l’inizio e la fine di ogni argomento.

È possibile disabilitare l’interpretazione del separatore “**;**” facendolo precedere da un ulteriore carattere speciale, detto carattere di escape “**\**” (backslash).

Esistono variabili d’ambiente che contengono gli argomenti passati allo script (NB: **I parametri non possono essere modificati**):

* **$#**: 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

Es: “$\*” à “$1 $2 $3” quotati tutti gli argomenti assieme

“$@” à “$1” “$2” “$3” quotato singolarmente ogni argomento

### Brace expansion

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

Es:

echo a{1,2,3}b

a1b a2b a3b

È consentito l’annidamento, ovvero inserire degli ordini di brace expansion all’interno di altri ordini di brace expansion.

Es:

echo a{1,{b..d}}e

a1e abe ace ade

### Tilde expansion

Cinque casi essenziali:

* cd ~ cambia la directory corrente in /home/andrea
* echo ~/ stampa /home/andrea
* echo ~/documenti stampa /home/andrea/documenti
* echo ~gamer stampa /home/gamer
* echo ~gamer/video stampa /home/gamer/video

### Pathname expansion (wildcards)

Sono tre metacaratteri che vengono inseriti dall’utente nei comandi digitati e la shell interpreta cercando di sostituirli con una sequenza di caratteri per ottenere i nomi di files nel file system.

* **\*** à viene sostituito da una qualunque sequenza di caratteri, anche vuota;
* **?** à viene sostituito da esattamente un singolo carattere;
* **[***elenco***]** à viene sostituito da un solo carattere tra quelli specificati in *elenco*.
  + **[**abk**]** può essere sostituito da un solo carattere tra i tre.
  + **[**1-7**]** può essere sostituito da un solo carattere tra 1 2 3 4 5 6 oppure 7.
  + **[**c-f**]** può essere sostituito da un solo carattere tra c d e oppure f.
  + **[**[:digit:]**]** può essere sostituito da un solo carattere numerico (una cifra).
  + **[**[:upper:]**]** può essere sostituito da un solo carattere maiuscolo.
  + **[**[:lower:]**]** può essere sostituito da un solo carattere minuscolo.

### Arithmetic expansion

È possibile valutare una stringa come se fosse un’espressione costituita da operazioni aritmetiche tra soli numeri interi. L’operatore **(( ))** racchiude tutta una riga di comando, che deve essere un’espressione (più un eventuale assegnamento). L’operatore **$(( ))** invece può racchiudere anche solo una parte di una riga di comando, che deve essere un’espressione (più un eventuale assegnamento).

Es:

**((**NUM=${NUM}+3**))**

4

Es:

NOMEFILE=pippo**$((** ${NUM}+3 **))**

pippo4

Piccola parentesi:

{

Accedere al valore di una variabile il cui nome è il valore di una seconda variabile è possibile nella bash a partire dalla versione 2. Si sfrutta l’operatore **${!***nomevariabile***}**.

Es:

varA=pippo

nomevarA=varA

echo ${!nomevarA}

pippo

}

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. Questo valore si chiama **exit status**. Un risultato **0** **indica** **tutto bene**, **altrimenti** c’è un **errore**. Il comando **exit** restituisce il risultato in uno script bash. La variabile d’ambiente **$?** contiene il risultato numerico restituito ogni volta che termina un comando o programma.

**$?** conterrà 0 se:

* Una valutazione aritmetica fornisce un risultato logico TRUE;
* Una valutazione aritmetica fornisce un risultato intero diverso da 0.

**$?** conterrà un valore diverso da 0 se:

* Durante una valutazione aritmetica è accaduto un errore;
* Una valutazione aritmetica fornisce un risultato logico FALSE;
* Una valutazione aritmetica fornisce un risultato intero uguale a 0.

Una **lista di comandi** è un elenco di comandi da lanciare in esecuzione in successione. Ciascun elemento dell’elenco deve essere separato dal successivo da un punto e virgola. Il risultato exit status da una lista di comandi è l’**exit status** restituito dall’**ultimo comando** che è stato lanciato in lista.

## 6. 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 diverso da 0 per indicarne la **falsità**. Ciascuna espressione condizionale si scrive mettendo le condizioni da valutare tra doppie parentesi quadre **[[ ]]**. **Non possono essere comandi**. Sono disabilitate le interpretazioni di word splitting, brace expansion e pathname expansion.

Esistono tre operatori per le espressioni condizionali al cui interno posso specificare gli operatori per Condizioni di file, Confronto tra stringhe, Confronto aritmetico:

* **[[** *condizione* **]]** à Permette di comporre tra loro più condizioni, utilizzando gli stessi operatori logici usati in C ( ! , && , || ). Inoltre, all’interno, posso usare parentesi tonde per raggruppare espressioni e modificare la precedenza egli operatori e posso andare a capo, proseguendo l’espressione.
* **[** *condizione* **]** e **test** *condizione* àPermettono di comporre tra loro più condizioni, utilizzando però gli operatori logici: **!** (not), **-a** (and), **-o** (or). Inoltre, non posso usare parentesi tonde per raggruppare espressioni e modificare la precedenza degli operatori. Infine, non posso andare a capo se non usando **\** a fine linea.

Alcuni operatori per verificare condizioni su file:

* **-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 the effective group id.
* **-L** *file* True if *file* exists and is a symbolic link (deprecated, see -h).
* *file1* **-nt** *file2* True if *file1* is newer than *file2*, or if *file1* exists and *file2* does not.
* *file1* **-ot** *file2* True if *file1* is older than *file2*, or if *file2* exists and *file1* does not.

Alcuni operatori per verificare condizioni su stringhe, aritmetiche e altre varie:

* **-o** *optionname* True if shell option *optionname* is enabled.

Operatori su stringhe

* **-z** string True if the length of *string* is zero.
* **-n** string True if the length 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.

Operatori aritmetici su stringhe

* *arg1* **OP** *arg2* Where **OP** is one of:
  + **-eq** equal
  + **-ne** not equal
  + **-lt** less than
  + **-le** less or equal
  + **-gt** greater than
  + **-ge** greater or equal

## 7. Costrutti condizionali

### If

**if** **[[** *condizione* **]]** **;** **then**

*istruzione1*

**elif**

*istruzione2*

**else**

*istruzione3*

**fi**

### If-case

**case** *variabile* **in**

*valore1***)**

*istruzione1*

*valore2***)**

*istruzione2*

*…*

\*)

*istruzioneN*

**esac**

### for

**for** *i* **in {***startVal..endVal***}**

**do**

*istruzione*

**done**

Oppure

**#!/bin/bash**

**for ((** *i = startVal* **;** *i* <= *endVal* **;** *i++* **)) ; do**

*istruzione*

**done**

### while

**while [[** *condizioneTrue* **]] ; do**

*istruzione*

**done**

### until

**until [[** *condizioneFalse* **]] ; do**

*istruzione*

**do**

“**||**” 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).

## 8. Stream I/O

### File Descriptor

Il **file descriptor** è un’astrazione per permettere l’accesso ai file. Ogni file descriptor è rappresentato da un integer. Ogni processo ha la propria **file descriptor table** che contiene (indirettamente) le informazioni sui file attualmente utilizzati (aperti) dal processo stesso. In particolare, la file descriptor table contiene un file descriptor per ciascun file usato dal processo.

### Stream I/O predefiniti dei processi

Quando un programma entra in esecuzione, l’ambiente del sistema operativo si incarica di aprire tre flussi di dati standard:

* Standard input (**stdin**): serve per l’input normale (default tastiera). Valore rappresentativo **0**;
* Standard output (**stdout**): serve per l’output normale (default schermo video). Valore rappresentativo **1**;
* Standard error (**stderr**): serve per l’output che serve a comunicare messaggi di errore all’utente (default schermo video). Valore rappresentativo **2**;

Una shell figlia B 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.

#### Command substitution

Catturare, in una variabile, l’output di un programma chiamato da uno script:

*variabile*=`./*eseguibile.exe*`

L’apice giusto da usare è “ ` “ (backtick).

In alternativa si può utilizzare **$(** *./eseguibile.exe* **)** .

#### Read

Uno script può leggere dallo standard input delle sequenze di caratteri usando un comando chiamato **read** *variabile*. Il comando riceve la sequenza di caratteri digitate da tastiera fino alla pressione del tasto INVIO e mette i caratteri ricevuti in *variabile*. La read restituisce un risultato che indica se la lettura è andata a buon fine: **0** (se non si arriva alla fine e viene letto qualcosa) oppure **>0** (se si arriva a fine file).

Può capitare che leggendo da file, nell’ultima riga non ci sia l’andata a capo **\n** alla fine. Ciò può causare risultati inaspettati. Perciò, quando si fa una lettura con la read, se la read dice di essere arrivata a fine file occorre comunque controllare se in *variabile* c’è qualcosa dentro o meno.

Esempio:

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

Posso anche passare più variabili come argomenti a read:

**read** varA varB varC

Poi digito da tastiera:

prima seconda terza

Le variabili assumono il valore:

* varA=”prima”
* varB=”seconda”
* varC=”terza”

Per leggere correttamente anche gli spazi bianchi, occorre dire alla bash che gli spazi bianchi e le tabulazioni NON SONO delimitatori delle parole. Occorre a tal scopo settare preventivamente la variabile IFS come VUOTA.

IFS=""; read RIGA < file\_con\_spazio\_iniziale\_nella\_riga.txt echo \"${RIGA}\"

causa l'output corretto seguente:

" K 23F G2"

L’opzione **-N** della read permette di specificare il numero di caratteri che devono essere letti. Attenzione. Se durante la lettura viene raggiunta la fine riga, allora la read termina anche se non ho letto tutti i caratteri richiesti.

### Stream di I/O non predefiniti

È possibile, inoltre, aprire file su disco per fare I/O anche se non vengono passati mediante stdin e stdout. Ecco come:

|  |  |
| --- | --- |
| Modo apertura | Sistema sceglie un fd libero e lo inserisce in *variabile* |
| Solo lettura | **exec** {*variabile*}**<** *percorsofile* |
| Scrittura | **exec** {*variabile*}**>** *percorsofile* |
| Aggiunta in coda | **exec** {*variabile*}**>>** *percorsofile* |
| Lettura e scrittura | **exec** {*variabile*}**<>** *percorsofile* |

Nel comando read posso specificare l’opzione **-u** (seguita dal fd del file aperto) per indicare al comando read da quale file aperto deve essere effettuata la lettura.

Esempio di apertura di file in LETTURA:

**exec** **{FD}**< /home/andrea/mioinput.txt

**while** read **-u ${FD}** StringaLetta ; **do**

echo “ho letto: $StringaLetta}”

**done**

Esempio di apertura di file in SCRITTURA:

**exec** **{FD}**> /home/andrea/miooutput.txt

**for** name **in** pippo pippa pippi ; **do**

echo “inserisco ${name}” **1>&${FD}**

**done**

Dove trovo i file descriptors aperti da una bash?

Nella directory /proc/ esiste una sub-directory per ciascun processo, in esecuzione, del processo stesso. Il comando: **ls /proc/$$** visualizza il contenuto della directory corrispondente alla shell corrente.

Nella sub-directory propria di ciascun processo, esiste un’ulteriore sub-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 può essere effettuata tramite il seguente comando: **exec ${FD}>&-** (dove **FD** contiene il file descriptor del file da chiudere).

### Ridirezionamenti di Stream di I/O

Modalità:

1. **< : ricevere input da file;**
2. **> : mandare stdout verso file eliminandone il vecchio contenuto;**
3. **>> : mandare stdout verso file aggiungendo il nuovo contenuto al vecchio;**
4. **| : ridirigere output di un programma nell’input di un altro programma.**

#### 1) Ricevere input da file ( < )

È possibile dare input a un programma non solo da tastiera, ma anche da file. Al momento dell’ordine di esecuzione del programma, si ridireziona il contenuto di un file sullo stdin del programma.

programma < input\_file

Il programma vedrà il contenuto di input\_file come se venisse digitato da tastiera.

#### 2) Mandare stdout verso file eliminandone il vecchio contenuto ( > )

Analogamente, è possibile ridirezionare lo stdout di un programma su un file, su cui sarà scritto l’output del programma invece che su video.

programma > output\_file

output\_file conterrà l’output del programma, sovrascritto al vecchio contenuto del file.

#### 3) Mandare stdout verso file aggiungendo il nuovo contenuto al vecchio ( >> )

Il ridirezionamento dello stdout può essere fatto anche senza eliminare il vecchio contenuto del file di output, ma semplicemente aggiungendo l’output al contenuto del file.

programma >> output\_file

output\_file conterrà l’output del programma, in coda al suo vecchio contenuto.

#### 4) Ridirigere output di un programma nell’input di un altro programma - PIPE ( | )

è possibile far eseguire contemporaneamente due processi, mandando lo stdout del primo nel stdin del secondo.

programma1 | programma2

Se una bash ha due stream (due file aperti) entrambi di output (o di input) e ciascuno stream è identificato da un fd di valore **N** e **M** rispettivamente, allora è possibile ridirigere lo stream **N** sullo stream **M**:

**N**>&**M**

Dopo di questo, ciò che scrivo su **N** viene scritto anche su **M**.

### Raggruppamento di comandi

Se si **racchiude** una **sequenza** **di comandi tra parentesi tonde**, allora in esecuzione viene creata una **subshell** per eseguire quella sequenza di comandi. Il risultato restituito dalla subshell è il risultato dell’ultimo comando racchiuso tra parentesi. Tutti i comandi condividono gli stessi stdin/stdout/stderr, utilizzandoli in sequenza. Ciò permette di trattare/**ridirigere input e output di tutti i comandi dentro le parentesi tonde come se fossero u solo comando**.

Esempio (concatenazione stdout):

**(** cat file1.txt ; cat file2.txt **)** | **grep** stringa

Il comando **grep** riceve in input, come se provenissero da tastiera, le righe di entrambi i file di testo.

In particolare, grep è un comando per la ricerca di stringhe.

### Alcuni comandi utili per editing di file di testo

* Prendere le prime 2 righe di un file: **head -n 2 file.txt**
* Prendere le ultime 3 righe di un file: **tail -n 3 file.txt**
* eliminare i primi 3 caratteri in una riga: **cut** **-b 4-**
* Comando **sed** (editor non interattivo di file di testo):

Opzioni per sed:

|  |  |  |
| --- | --- | --- |
| Operatore | Nome | Effetto |
| [indirizzo]/p | print | Visualizza [l'indirizzo specificato] |
| [indirizzo]/d | delete | Cancella [l'indirizzo specificato] |
| s/modello1/modello2/ | substitute | Sostituisce in ogni riga la prima occorrenza della stringa modello1 con la stringa modello2 |
| [indirizzo]/s/modello1/modello2/ | substitute | Sostituisce, in tutte le righe specificate in *indirizzo*, la prima occorrenza della stringa modello1 con la stringa modello2 |
| [indirizzo]/y/modello1/modello2/ | transform | sostituisce tutti i caratteri della stringa modello1 con i corrispondenti caratteri della stringa modello2, in tutte le righe specificate da *indirizzo* (equivalente di **tr**) |
| g | global | Agisce su *tutte* le verifiche d'occorrenza di ogni riga di input controllata |

Esempio:

**sed** ‘s/*seq1*/*seq2*/g’

Sostituisce ogni occorenza della sequenza di caratteri *seq1*, con la sequenza *seq2*.

## 9. Processi, Sessioni e Gruppi di processi

Un singolo comando può far partire più processi (Es: la pipe “ | “ ). L’insieme di questi processi viene chiamato **gruppo di processi** (*process group* o *job*).

Una **sessione** invece è un insieme più grande di processi, che comprende di solito tutti i processi fatti partire da un unico login. Il primo di questi viene chiamato **session leader**. Quando un processo viene creato, appartiene alla stessa sessione a cui appartiene il padre; tramite però il comando **setsid** il processo può divenire session leader di una nuova sessione. Questo muove il processo in un altro gruppo.

Normalmente si ha un solo session leader per login, e questo è la shell. Essa controlla il terminale attuale, e deve anche gestire i vari job che possono voler utilizzare contemporaneamente il terminale, assegnando a uno solo per volta il controllo dello stesso. Il processo che ha l’uso del terminale in un dato momento si dice in **foreground**, mentre gli altri sono in **background**.

Quando il processo che controlla il terminale (nel caso precedente è la shell) viene terminato manda a tutti gli appartenenti alla sessione un segnale (**SIGHUP**) che, se non è ignorato, causa la morte dei processi. Nel caso che il segnale sia ignorato, il processo non muore, ma diventa ``orfano'', cioè non appartiene più alla sessione, che non esiste, e diventa session leader.

Comandi:

* **&** à lancia un processo direttamente in background (Es: **prova &**).
* **Ctrl+z** à sospende un processo in foreground.
* **Ctrl+c** à termina un processo in foreground.
* **bg** à riprende l’esecuzione in background di un processo sospeso.
* **jobs** à produce una lista numerata dei processi in background o sospesi nella shell corrente.
* **fg %n** àporta in foreground un processo sospeso.
* **kill** à elimina il processo specificato dal proprio identiatore pid oppure specificato dal numero del job (Es: **kill 6152** oppure **kill %2**, 6152 = pid mentre %2 = numero indice nella lista visualizzata dal comando jobs).
* **disown –[ar] jobs** à sgancia il job dalla shell interattiva che lo ha lanciato.
* **ps -ax** oppure **top** à mostra la lista completa dei processi in esecuzione nel sistema.

### kill, trap e wait

#### kill

Un segnale è una interruzione software, provocata in un processo destinazione, per notificare un evento asincrono.

Il comando **kill** invia segnali ai processi. Il tipo di segnale viene specificato con un simbolo oppure con il valore numerico. I processi di destinazione vengono specificati mediante il loro PID.

Sintassi nel caso di **invio di segnali**:

**kill** [*opzioni*] [--] *arg1* [*arg2* …]

I parametri *arg* identificano i processi a cui inviare i segnali, che possono essere specificati nei due modi che seguono:

* dei PID;
* degli identificativi di job della shell, secondo la seguente sintassi:
  + **%%** o **%+** à indica il job corrente;
  + **%-** à indica il job precedente;
  + **%***n*à indica il job con identificativo *n*.
  + **%***stringa* à indica il job la cui riga di comando inizia con *stringa*.
  + **%?***stringa* à indica il job la cui riga di comando contiene *stringa*.

Il doppio trattino [--] (facoltativo) indica che i parametri successivi non sono da considerarsi opzioni.

Il comportamento predefinito è quello di inviare SIGTERM. “**-s** *segnale*” invia il *segnale* specificato, invece di SIGTERM.

Sintassi nel caso di **informazioni sui segnali**:

**kill** -l [*valore*]

Il parametro (facoltativo) *valore* può indicare:

* Il numero di un segnale, nel qual caso viene scritto sullo stdout il nome del segnale (senza il prefisso “SIG”);
* L’exit status di un comando che è terminato a causa del ricevimento di un segnale, come ad esempio ottenuto dalla variabile speciale “$?” della shell. Nel qual caso viene scritto sullo stdout il nome del segnale che ha causato la terminazione del comando (senza il prefisso “SIG”).

Segnali per kill più utilizzati:

|  |  |  |
| --- | --- | --- |
| Signal name | Signal value | Effect |
| SIGHUP | 1 | Hangup |
| SIGINT | 2 | Interrupt from keyboard |
| SIGQUIT | 3 | Quit |
| SIGABRT | 6 | Cancel |
| SIGKILL | 9 | Kill signal |
| SIGTERM | 15 | Termination signal - allow an orderly shutdown |
| SIGSTOP | 17,19,23 | Stop the process |

#### trap

Il comando **trap** definisce l’azione da svolgere al ricevimento di un elenco di segnali, esplicitando il nome di una funzione oppure una stringa di codice bash da eseguire.

Sintassi:

**trap** [*action*] [*signal*]

#### wait

Si supponga 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 questo bisogna utilizzare il comando built-in **wait** a cui passo come argomento il process identifier del processo di cui **attendere** **la terminazione**. Il comando wait termina quando termina il processo di cui è stato passato il PID. L’exit status restituito dal comando è lo stesso restituito dal processo figlio specificato.

Il comando waitpuò prendere diversi argomenti a riga di comando:

* Un **elenco di PID**: in tal caso wait aspetta la terminazione di tutti i processi indicati e restituisce come il risultato dell’ultimo processo nella lista PID, indipendentemente dall’ordine in cui terminano;
* Un **elenco di jobs**: in tal caso wait aspetta la terminazione di tutti i jobs indicati;
* **Nessun argomento**: in tal caso wait aspetta la terminazione di tutti i processi figli della shell in cui il comando wait è stato lanciato e restituisce exit status 0 non essendo indicato uno specifico figlio di cui restituire l’exit status.

### Processi: Zombie, Orfani e init

Un processo (padre) genera un processo figlio e lo esegue in background, il quale prima o poi terminerà, restituendo un valore intero. Il processo padre vuole sapere qual è il valore restituito dal figlio per sapere se tutto è andato bene o meno. A questo scopo, il processo padre esegue il comando **wait** *PIDfiglio* che attende la terminazione del processo figlio avente PIDfiglio e restituisce il risultato di tale processo.

Se il processo figlio termina prima del processo padre, il sistema operativo rilascia le risorse occupate dal processo figlio, ma mantiene nelle sue tabelle una struttura **pcb** (**process control block**) con una descrizione del processo terminato. Questa struttura conserva inoltre il PID e il risultato. In questo momento il processo figlio è uno **zombie** poiché è terminato ma la sua struttura descrittiva è ancora presente. La struttura pcb viene eliminata solo quando il padre invoca il comando wait.

Se invece il processo padre termina senza fare la wait per il figlio, il processo figlio diventa **orfano**. Quando un processo orfano termina, questo viene adottato dal processo **init**, quello con PID 0 che ha originato tutti i processi al boot. Gli orfani diventano così figli di init, il quale ogni tanto chiama la wait su di loro, e così fa rilasciare i pcb degli orfani.

### killall

Il comando **killall** viene usato, solitamente, per **terminare l’esecuzione di tutti i processi che hanno lo stesso nome**; quel nome viene passato come argomento a questo comando. Per default il segnale inviato è SIGTERM, ma può essere specificato anche uno segnale diverso.

Esempio:

**killall** -r **‘**programma[[:upper:]]\***’**

Gli apici semplici **‘ ‘** impediscono che sia la bash a interpretare i metacaratteri. L’interpretazione viene così fatta dalla killall con i nomi dei processi in memoria e non con quelli della directory corrente.

## 10. Approfondimento parameter expansion

### Estrazione di sottostringhe da variabili

* **Caso 1**: ${VAR%%pattern} à rimuovo il **più lungo suffisso**

VAR=”[13] qualcosa con [ o ] fine”

echo ${VAR%%**]\***}

[13~~] qualcosa con [ o ] fine~~

[13

* **Caso 2**: ${VAR%pattern} à rimuovo il **più corto suffisso**

VAR=”[13] qualcosa con [ o ] fine”

echo ${VAR%**]\***}

[13] qualcosa con [ o ~~] fine~~

[13] qualcosa con [ o

* **Caso 3**: ${VAR##pattern} à rimuovo il **più lungo prefisso**

VAR=”[13] qualcosa con [ o ] fine”

echo ${VAR##**\*[**}

~~[13] qualcosa con [~~ o ] fine

o ] fine

* **Caso 4**: ${VAR#pattern} à rimuovo il **più corto prefisso**

VAR=”[13] qualcosa con [ o ] fine”

echo ${VAR#**\*[**}

~~[~~13] qualcosa con [ o ] fine

13] qualcosa con [ o ] fine

### Sottostringhe da variabili

* **Caso 1**: ${VAR**/**pattern**/**string} à cerco in VAR la sequenza pattern e la sostituisco con string

VAR=”alfabetagamma”

echo “${VAR**/**g\*a**/**delta}

alfabeta~~gamma~~delta

alfabetadelta

* **Caso 2**: ${VAR**:**offset} à stampo VAR con un offset (elimino i primi offset caratteri)

VAR=”alfabetagamma”

echo “${VAR**:**4}

~~alfa~~betagamma

betagamma

* **Caso 3**: ${VAR**:**offset**:**length} àstampo VAR con un offset e una lunghezza length

VAR=”alfabetagamma”

echo “${VAR**:**4**:**5}

~~alfa~~betagamma

betag

### Ultime varianti aggiornate

* **Caso 1**: ${VAR**//**pattern**/**string} à cerco in VAR TUTTE le sequenze pattern e le sostituisco con string;
* **Caso 1**: ${VAR**/#**pattern**/**string} à cerco in VAR la sequenza pattern e la sostituisco con string SOLO se si trova all’INIZIO di VAR;
* **Caso 1**: ${VAR**/%**pattern**/**string} à cerco in VAR la sequenza pattern e la sostituisco con string SOLO se si trova alla FINE di VAR.

### Operatori vari su variabili

${**!***VarNamePrefix***\***}

Restituisce un elenco con tutti i nomi delle variabili il cui nome inizia con *VarNamePrefix*.