

Introduzione alla programmazione con SHELL

La shell è l'interprete dei comandi. Esistono vari tipi di shell (csh, rsh, tcsh, ecc.), ma la più usata in ambiente Linux è bash anche perché è compatibile con la Bourne Shell presente in tutti i sistemi Unix.

Spesso è utile creare file che contengono intere sequenze di comandi (script) che la shell può eseguire semplicemente digitando il nome dello script (dopo averlo reso eseguibile abilitando il diritto di esecuzione mediante il comando chmod).

bash offre un linguaggio di programmazione ad alto livello simile a linguaggi come il C. Noi tratteremo esclusivamente le caratteristiche di bash presenti anche nella Bourne Shell.

Redirezione dell' I/O:

Di default le applicazioni Unix acquisiscono l'input da tastiera e inviano l'output su video. Spesso è utile acquisire l'input da file e/o inviare l'output su un file di destinazione.

Esempi:

```
$ ls /usr/doc > listadoc.txt
$ cat listadoc.txt
```

Il comando `ls` visualizza il contenuto della directory su schermo, ma tramite il simbolo `>` l'output di `ls` è rediretto su un file (nel nostro caso `listadoc.txt`).

Se il file non esiste viene creato, mentre se esiste già viene riscritto.

I caratteri di redirezione `>>` consentono invece di aggiungere l'output in coda al file di destinazione. Se il file non esiste viene comunque creato.

I comandi quando segnalano un errore inviano il loro output allo standard error output. Analogamente allo standard output, lo standard error output può essere rediretto tramite i caratteri `2>` o `2>>`

```
$cc pippo.c 2> errori.compilazione
$cc pluto.c 2>> errori.compilazione
```

Se si vogliono redirigere standard output e standard error output contemporaneamente sullo stesso file si usano i caratteri `&>`

Spesso è anche utile poter redirigere lo standard input usando il carattere `<`

```
$wc < /etc/passwd
    15      23     788
```

Il comando `wc` legge da standard input un file restituendo il numero di linee (15), parole (23) e caratteri (788) del file.

Tramite carattere di redirezione `<` il file `/etc/passwd` viene inviato a `wc` come se fosse stato digitato da tastiera.

Talvolta è invece più comodo aggiungere i dati di input di un programma (che sarebbero letti da standard input) direttamente nello script file. Per questo si usano i caratteri di redirezione `<<`.

Esempio:

```
cat << FINE > testo.txt
```

Il comando attende dallo standard input linee di testo che reindirizza sul file `testo.txt`. Il comando termina quando si digita la stringa `FINE`.

Pipes:

Può risultare utile reindirizzare lo standard output di un comando nello standard input di un altro comando per evitare la creazione di file temporanei.

Esempio:

Contare il numero di file nella directory `/usr/doc`:

```
$ls /usr/doc | wc -l
231
```

Il comando `ls` visualizza i nomi dei file contenuti in `/usr/doc`, uno per linea.

Il comando `wc -l` conta il numero di linee di un file.

Tramite l'uso dell'operatore `|` (operatore di piping) i due comandi vengono connessi e si ottiene su video il numero di files contenute nella directory `/usr/doc`.

Ovviamente è possibile eseguire una sequenza di comandi connessi da pipes, es.:

```
$sort testo.txt | uniq | wc -l
```

`sort` ordina un testo in ordine alfabetico, `uniq` elimina le righe che risultano uguali alla precedente, pertanto la sequenza di comandi produce su video il numero di righe differenti che compaiono nel file `testo.txt`.

Come creare uno script file:

Per creare uno script file è sufficiente creare un file contenente il programma della shell tenendo conto delle seguenti note:

1. La prima linea deve contenere `#! <nome dell'interprete dello script>` (nel nostro caso `/bin/sh` o `/bin/bash`) seguita da una linea vuota.
2. I commenti possono essere inseriti con il carattere `#`
3. Il file deve essere reso eseguibile

Esempio:

```
#!/bin/sh

# Commento: Ecco uno script banale
cat pippo.txt # Visualizza pippo.txt
ls /usr/bin   # Visualizza il contenuto della directory /usr/bin
```

Le variabili d'ambiente:

La shell dispone di un proprio ambiente nel quale sono definite alcune variabili che consentono di impostare il funzionamento dell'interprete dei comandi e che possono essere utilizzate dai programmi che vengono eseguiti.

Ecco la lista delle variabili più importanti:

PATH	<p>Contiene una lista di percorsi del file system (separati da :) in cui la shell cerca il file eseguibile corrispondente ad un comando inserito dall'utente esempio:</p> <p>Se PATH è uguale a <code>/bin:/usr/bin:/usr/local/bin</code> allora tutti i comandi digitati verranno cercati, in sequenza, solo nelle directory <code>/bin</code>, <code>/usr/bin</code> e <code>/usr/local/bin</code></p>
HOME	Contiene il percorso della home directory dell'utente (es: <code>/home/davide</code>)
PS1	Contiene la stringa utilizzata come prompt dell'utente (\$ per default).
PS2	Contiene la stringa utilizzata come prompt secondario, per indicare la continuazione della linea precedente (> per default).

Il contenuto di una variabile può essere cambiato dall'utente tramite assegnazione:
variabile = "*valore*" (le virgolette servono nel caso in cui *valore* contenga degli spazi).

Ad esempio digitare `PS1="Cosa vuoi fare?"` per veder cambiare il prompt.

Per accedere al contenuto di una variabile si usa il nome della variabile preceduto dal carattere \$ es: `$HOME`. \$ può essere visto come un operatore che restituisce il contenuto della variabile.

Esempi che usano il comando `echo` che visualizza una stringa sul video.

```
$echo Ciao a tutti
Ciao a tutti
$echo $PS1
$
$echo $PS2
>
$PS1="Cosa vuoi fare?"
Cosa vuoi fare?echo $PS1
Cosa vuoi fare?
```

Per visualizzare il contenuto di tutte le variabili presenti nella shell attuale il comando è `set`.
 Per creare una nuova variabile è sufficiente digitarne il nome seguito da `=` e da un valore di inizializzazione (es: `numlinee=10`).

A volte è utile generare nomi di variabili utilizzando il contenuto di altre variabili o es:

```
$nome=Gian
$echo $nomemarco
$echo ${nome}marco
Gianmarco
$
```

Si può notare che il comando `echo $nomemarco` restituisce il contenuto della variabile `nomemarco` che non esiste e quindi produce una linea vuota.

Per stampare Gianmarco occorre racchiudere tra parentesi graffe il nome della variabile per separarlo dalla stringa successiva. Quindi il comando corretto è `echo ${nome}marco`.

Argomenti negli shell script

Come per le funzioni nei linguaggi di programmazione, a volte è necessario passare dei parametri ad uno script.

La shell predefinisce le variabili 0, 1, 2, 3, ... per contenere gli argomenti passati sulla riga di comando.

La variabile 0 contiene il nome dello script, mentre \$1, \$2, \$3, ... fanno riferimento alla serie di argomenti passati sulla riga di comando, separati da spazi.

Esempio:

Lo script *parametri*

```
echo 0=$0
echo 1=$1
echo 2=$2
```

visualizza i parametri passati sulla linea di comando:

```
$parametri uno due
0=parametri
1=uno
2=due
$
```

Altre variabili che fanno riferimento agli argomenti passati sulla linea di comando sono

\$# Numero dei parametri passati allo script

\$* Lista dei parametri (\$1 , \$2 , \$3 , ...) passati allo script separati da uno spazio

Talvolta è necessario far riferimento alla propria home directory.

La shell mantiene questa informazione nella variabile HOME. È comunque far riferimento alla home directory di qualsiasi utente mediante la sintassi ~username.

Esempio:

```
$ls ~Rossi                    #Visualizza il contenuto della directory home dell'utente
Rossi
$ls ~                        #Visualizza il contenuto della vostra directory home
#                            (equivalente a ls $HOME)
```

Altri modi di assegnazione di valori alle variabili:

Talvolta è utile gestire un input interattivo da parte dell'utente. Per ottenerlo si usano il comando *read* e la variabile predefinita *REPLY*.

Esempio:

```
$read
ciao, come stai?
$echo $REPLY
ciao, come stai?
$
```

L'assegnazione `linea=$REPLY` memorizzerà nella variabile `linea` la stringa inserita da tastiera.

La sintassi del comando *read* prevede anche una lista di variabili, in questo caso nelle variabili dalla prima alla penultima saranno memorizzate le "parole" inserite da tastiera; l'ultima variabile invece conterrà il resto della stringa che non è stata memorizzata nelle altre variabili.

Esempio:

```
$read
ciao
$echo $REPLY
ciao
$read uno due resto_della_linea
Il kernel di Linux viene aggiornato frequentemente
$echo $uno
Il
$echo $due
kernel
$echo $resto_della_linea
di Linux viene aggiornato frequentemente
$
```

Come si può notare il separatore predefinito (IFS, Internal Field Separator) è lo spazio. All'ultima variabile (resto_della_linea) è assegnata la parte rimanente della linea digitata dall'utente. Le variabili non inizializzate rimangono, ovviamente, vuote.

La variabile predefinita IFS può essere modificata dall'utente per sostituire un altro carattere come separatore di campi nella shell.

Assegnazione dell'output di un comando ad una variabile:

Talvolta è comodo assegnare l'output di un comando ad una variabile.

Per fare questo si usa il carattere apice inverso ` `.

Esempio:

```
$ date
Fri Jul 28 20:51:27 2000
$ oggi=`date`
$ echo $oggi
Fri Jul 28 20:51:41 2000
$
```

Il comando date produce una stringa che contiene la data e l'ora, la stringa viene assegnata alla variabile oggi.

N.B.: Per ottenere lo stesso effetto potete usare, se preferite, la sintassi \$(...) in modo equivalente: oggi=\$(date)

Esaminare lo stato di uscita dei programmi:

I programmi Unix restituiscono un valore di uscita quando terminano. Questo valore - che è memorizzato nella variabile ? - normalmente indica se l'elaborazione è stata completata correttamente. La convenzione utilizzata prevede che il processo restituisce il valore 0 quando non ci sono stati errori.

Nel caso di una cascata di comandi in pipe il valore restituito è quello relativo all'ultimo comando (es: In cat pippo.txt | grep 'Luca' viene ritornato il valore restituito dal comando grep).

Esempio:

```
$cd /khghg
bash: cd: /khghg: No such file or directory
$echo $?
1
$echo $?
0
```

Poiché la directory /khghg non esiste, il comando `cd` restituisce 1.

Il secondo comando `echo $?` restituisce 0 perché il comando precedente è andato a buon fine.

Se si desidera restituire un valore di uscita in uno script, occorre usare il comando `exit` numero (es: `exit 0` se lo script ha funzionato correttamente, `exit 1` se ci sono stati problemi).

Controllo del flusso:

Come i linguaggi di programmazione ad alto livello, la shell dispone di comandi in grado di controllare il flusso di esecuzione.

Il comando `if` ha la sintassi:

```
if comando_di_condizione
then
    comandi da eseguire se comando_di_condizione restituisce 0 (vero)
else
    comandi da eseguire se comando_di_condizione restituisce
    un valore diverso da 0 (falso)
fi
```

Ovviamente il comando `else` è opzionale.

Attenzione!! `if`, `then`, `else` e `fi` sono interpretati come comandi e quindi è errata la sequenza:

```
if comandi_di_condizione then comando_per_vero
else comando_per_falso fi
```

Se si vuole che `if` e `then` stiano sulla stessa riga occorre utilizzare un carattere ";" (che è l'operatore di sequenza) per separarli.

È bene ricordare che, al contrario del linguaggio C, nella shell 0 equivale a vero.

Comando di condizione:

Spesso come comando di condizione è necessario inserire un'espressione.

Per fare ciò è utile il comando `test` espressione.

Se il risultato di espressione è vero `test` ritorna 0 altrimenti ritorna 1.

Ecco quali condizioni può verificare `test` :

Condizioni sui file:

<code>-e nomefile</code>	vera se il file esiste
<code>-f nomefile</code>	vera se il file è di tipo normale
<code>-d nomefile</code>	vera se il file è una directory
<code>-r nomefile</code>	vera se il file è leggibile dall'utente che esegue lo script
<code>-w nomefile</code>	vera se il file è scrivibile dall'utente che esegue lo script
<code>-x nomefile</code>	vera se il file è eseguibile per l'utente che esegue lo script

Condizioni sulle stringhe:

<code>-z stringa</code>	vera se la lunghezza della stringa è zero
<code>-n stringa</code>	vera se la lunghezza della stringa <u>non</u> è zero
<code>str1 = str2</code>	vera se str1 è uguale a str2
<code>str1 != str2</code>	vera se str1 è diversa da str2

Condizioni sulle espressioni numeriche:

num1 -eq num2	vera se num1 è uguale a num2 (equal)
num1 -ne num2	vera se num1 è diverso a num2 (not equal)
num1 -lt num2	vera se num1 è minore a num2 (less then)
num1 -gt num2	vera se num1 è maggiore a num2 (greater then)
num1 -le num2	vera se num1 è minore o uguale a num2 (less or equal)
num1 -ge num2	vera se num1 è maggiore o uguale a num2 (greater or equal)

Il comando test ammette poi gli operatori logici AND,OR,NOT

espr1 -a espr2	vera se espr1 e espr2 sono entrambe vere (AND)
espr1 -o espr2	falsa se espr1 e espr2 sono entrambe false (OR)
!espressione	vera se espressione è falsa (NOT)

Variabili ed espressioni numeriche:

Le variabili di shell contengono stringhe di caratteri. È possibile far interpretare alla shell variabili che contengono sequenze di cifre come variabili numeriche ed eseguire su tali variabili semplici calcoli.

Per fare interpretare alla shell il contenuto di una variabile come numero e per effettuare operazioni aritmetiche si usa la sintassi `$(espressione)`.

```
$a=10
$b=2
$c=$((a/b))
$echo $c
5
$c=$((a/b+1*a-b))
$echo $c
13
$
```

Selezione: Per evitare l'uso di una serie di if annidati la shell dispone del costrutto case (equivalente a switch del linguaggio C) che consente la scelta multipla:

```
case stringa in
    str_1) alternativa 1
        ;;
    str_2) alternativa 2
        ;;
    str_3) alternativa 3
        ;;
    .....
    .....
    *) comandi di default # eseguiti se le condizioni precedenti non sono
                          # verificate
        ;;
esac
```

le varie stringhe di selezione delle alternative `:str_1, str_2, str_3, ...` possono anche essere costituite da espressioni regolari.

Cicli:

La shell dispone di tre comandi per realizzare cicli: `while`, `until`, e `for` con la seguente sintassi:

```
while:                                while condizione
                                     do
                                     istruzioni
                                     done

until:                                until condizione
                                     do
                                     istruzioni
                                     done

for:                                  for variabile in lista di parole
                                     do
                                     istruzioni
                                     done
```

Le istruzioni tra i comandi `do` e `done` vengono eseguite solo se il risultato della condizione é zero (vero).

Il ciclo `for` viene eseguito associando ad ogni ciclo a variabile una dopo l'altra le parole contenute nella lista di parole.

Esempio:

```
for i in 2 1 -1 a
do
    # Osservare: i , $i
    echo Il valore di i ora e\' $i
done
```

produce l'output:

```
Il valore di i ora e' 2
Il valore di i ora e' 1
Il valore di i ora e' -1
Il valore di i ora e' a
```

Notare anche `\'` : `\` è il metacarattere che modifica il significato usuale del carattere successivo (es. `\n` modifica il carattere `n` facendolo interpretare come newline). In questo caso evita che la shell interpreti il carattere `'`.

Il comando `for` è utile, ad esempio, per elaborare in sequenza i parametri `$1`, `$2`, `$3`,... oppure una sequenza di nomi di file.

Un ciclo `for` simile a quello del linguaggio C si realizza mediante il ciclo `while` :

```
i=0
while test $i -lt 10
do
    echo i vale $i
    i=$((i + 1))
done
```


Esercizio: Realizzare uno script che copi nel direttorio /home/salva i file specificati nella linea di comando se differiscono dai file già presenti in /home/salva.

(Il comando `cmp -s` restituisce vero se due file sono uguali senza emettere output)

Prima soluzione:

```
#!/bin/sh

for i in $*
do
    if test -e "/home/salva/$i" # Se il file esiste
    then
        cmp -s $i /home/salva/$i
        if test $? -ne 0; then cp $i /home/salva; echo Aggiornamento di $i; fi
    else
        cp $i /home/salva          # Se il file non esiste lo copia direttamente
    fi
done
```

Seconda soluzione:

Sapendo che se uno dei due file da confrontare non esiste `cmp` restituisce un codice diverso da zero (falso) ecco una soluzione più breve è:

```
#!/bin/sh

for i in $*
do
    cmp -s $i /home/salva/$i
    if test $? -ne 0; then cp $i /home/salva; fi
done
```

Esercizio:

Scrivere uno script in grado di stampare un triangolo di asterischi al centro dello schermo. Lo script riceve come parametro il numero di asterischi della prima linea

```
#!/bin/sh

inizio=$(( 80 / 2 - $1/2 )) # per terminali a 80 colonne

num_asterischi=$1
while test $num_asterischi -gt 0
do
    i=$inizio
    while test $i -gt 0
    do
        echo -n " "
        # L'opzione -n fa sì che echo non vada acapo
        i=$((i - 1))
    done

    i=0
    while test $i -lt $num_asterischi
    do
        echo -n "*"
        i=$((i + 1))
    done
```

```

echo ""                # Passa alla linea successiva
inizio=${inizio + 1}
num_asterischi=${num_asterischi - 1}
done

```

Come si può notare usare la shell per certe operazioni è scomodo quindi per script di questo tipo si consiglia l'uso di awk.

Ecco l'output prodotto dallo script:

```

$triangolo 5
          *****
          ****
          ***
          **
          *
$

```

Esercizio: Scrivere uno script che dati due parametri disegni un rettangolo e successivamente un trapezio di asterischi.

Esercizio:

Implementare uno script in grado di stampare un triangolo di numeri con il seguente formato:

```

1
2 3
4 5 6
7 8 9 10
.....
.....

```

Il numero a cui lo script deve fermarsi è passato come parametro (si assuma che sia compatibile con la fine della riga da stampare).

```

#!/bin/sh

if test $# -ne 1  # Se il numero di parametri e' diverso da 1
then
    echo Sintassi: $0 \"numero finale\"
    exit 1          # Esce ritornando codice 1
fi

fine=$1
i=1
acapo=1
while true        # True è un comando che restituisce sempre 0
do
    j=0
    while test $j -lt $acapo
    do
        echo -n \"$i \"
        if test $i -eq $fine; then echo \"\"; exit 0; fi;
    # exit 0 = esce restituendo codice 0
        i=$((i + 1))
        j=$((j + 1))
    done
done

```

```

echo ""
acapo=$((acapo + 1))
done

```

Per le condizioni dei cicli che sono operazioni aritmetiche o logiche è più comodo utilizzare al posto del comando test direttamente il risultato delle espressioni.

Esempio:

```

#!/bin/sh

i=0
while [ $i -lt 10 ]    # Al posto di test $i -lt 10 ;
do
    echo i vale $i
    i=$((i + 1))
done

```

Notare che `$(...)` interpreta il contenuto di una variabile come un numero, mentre il carattere `$` è omesso nel caso in cui l'espressione rappresenta una condizione.

Si assuma ora che lo script richieda il parametro di ingresso e che cicli all'infinito.

L'unico modo di interromperlo è usare la combinazione di tasti Ctrl-C.

Si scriva uno script simile al precedente ma che esca solo dopo aver stampato una riga completa anche se l'utente lo interrompe con Ctrl-C.

```

#!/bin/sh

i=1
acapo=1
while true
do
    j=0
    while test $j -lt $acapo
    do
        linea="$linea $i"    # Notare come si ottiene la concatenazione di stringhe
        i=$((i + 1))
        j=$((j + 1))
    done

    echo $linea
    linea=""
    acapo=$((acapo + 1))
done

```

Job in Background:

Un comando è interpretato ed eseguito dalla shell, quando il comando è completato (o è stato interrotto) la shell restituisce il controllo all'utente proponendo sul video il prompt per il comando successivo.

Talvolta un comando necessita di un lungo tempo di esecuzione e potrebbe essere utile disporre immediatamente del prompt per eseguire nuovi comandi senza attendere la terminazione del precedente.

Per ottenere immediatamente il prompt usa il carattere `&` alla fine della linea di comando.

```
$ls -R / > risLS.txt &
[1] 412
$
```

Mentre il comando `ls -R` traversa e memorizza sul file `risLS.txt` tutto l'albero del filesystem (operazione che può durare a lungo) è possibile inserire nuovi comandi. Si dice che l'esecuzione del comando è stato inviata in background.

I numeri che vengono prodotti prima del prompt sono rispettivamente il numero del job in background [1] e il PID del relativo processo (412).

Controllo dei job:

Programmi connessi attraverso pipe otterranno un unico numero di job (sono considerati un unico comando).

Il numero di job permette di controllare i job.

Per ottenere la lista dei job presenti nel sistema si usa il comando `jobs` senza argomenti

```
$jobs
[1] Running find / -name "*.txt" -print &> pr.txt &
[2]- Running ls -R / &> pr2.txt &
[3]+ Exit 1 ls -R mio* &> pr3.txt
$
```

Se si vuole inviare un processo in esecuzione in background occorre prima sospenderlo mediante la combinazione di tasti `Ctrl-z`:

```
$cat >testo.txt
<<<<<Ctrl-z>>>>>
[1]+ Suspended cat >testo.txt
$
```

Ora il comando `cat >testo.txt` risulta sospeso ed ha assegnato un numero di job [1].

Usando il comando `fg` si può farlo tornare in esecuzione (in foreground), se invece si usa il comando `bg` si può farlo tornare in esecuzione, ma in background:

Se usato senza parametri `bg` si riferisce al job di default contrassegnato dal "+" subito dopo il numero identificativo del job (nel nostro caso [1]).

Nel caso in cui si vogliano inviare in background altri job diversi da quello di default si usa la sintassi `bg %numero` dove numero è il numero identificativo del job (es: `bg %3` per mandare in background il job numero [3]).

Con la stessa sintassi di `bg`, il comando `fg` riporta un job da background in foreground.

Per interrompere un job in background si usa il comando `kill %numero`.

Esercizio: Realizzare uno script "elimina" che accetta come parametri i nomi di alcuni processi in esecuzione e li interrompe.

Il problema da risolvere è quello di ricavare il PID dei processi da interrompere.

Per fare questo si può usare il comando `ps`.

```
#!/bin/sh

for i in $*
do
    kpid=$(ps x|awk "/[0-9]*:[0-9]*\ $i/ { printf(\"%s \",\ $1) }")
```

```

    for j in $kpid
    do
        kill $j
    done

done

```

Chi non conoscesse ancora l'uso di `awk` sappia che in questo caso si occupa di trovare, nella lista generata da `ps`, il nome del processo *i*-esimo e di assegnare a `kpid` il PID estratto dal primo campo della linea corrispondente al nome del processo trovato

Si ricorda che `$(...)` corrisponde a ``...``

Rifare l'esercizio utilizzando i comandi `grep`, `cut` e `paste`.

Esercizio: Scrivere due script di nome `sincro1` e `sincro2` con le seguenti caratteristiche

- `sincro1`: Riceve in input un testo che termina con la parola "fine"
- `sincro2`: Attende che `sincro1` abbia finito di ricevere l'input e stampa il testo numerandone le linee

```

#!/bin/sh

#          sincro1

echo -n > testo.txt                                # Crea il file vuoto

until test "$linea" = "fine"                        # digitare fine per terminare
do
    echo -n '=>'
    read linea
    echo $linea >> testo.txt
done

ln -fs $0 link.tmp                                # Forza la creazione di un link
#                                                  simbolico allo script file script1

#!/bin/sh

#          sincro2

until test -e link.tmp # Aspetta che il file link.tmp sia stato creato

do
    sleep 5                                          # attende 5 secondi
done

echo                                                # va a capo
cat -n testo.txt                                    # cat -n numera le linee (si poteva usare anche awk)

rm link.tmp                                          # sincro2 ha finito di elaborare testo.txt

```

Per impedire che `sincro2` elabori l'output di `sincro1` prima che questi abbia finito con l'input del testo (che viene copiato nel file `testo.txt`) si è usata l'istruzione `ln` per creare un link simbolico nella directory corrente.

Lo script `sincro2` esamina ad intervalli regolari se il file `link.tmp` esiste nella directory per capire se il lavoro dello script `sincro1` è terminato.

Ovviamente `sincro2` deve essere avviato in background e quindi i due script potranno essere eseguiti con il comando: `sincro2 & sincro1`

Segnali:

Il comando `kill` è in grado di inviare segnali ai processi.

```
kill -sig pid
```

`-sig` indica il tipo di segnale da inviare al processo con PID `pid`. Se `-sig` viene omissso allora viene inviato il segnale di terminazione 15 (`TERM`)

Esamineremo solo il caso in cui `pid` è un numero maggiore di 0 e corrisponde ad un processo appartenente all'utente.

Quando un processo riceve un segnale si può comportare in tre modi differenti:

- 1) Lo ignora
- 2) Esegue l'azione di default del segnale (terminazione)
- 3) Esegue del codice di gestione del segnale

NB: Il segnale 9 (`KILL`) non può essere ignorato e neanche gestito e provoca quindi la terminazione immediata del processo

Uno script può reagire ad un segnale eseguendo un opportuno codice di gestione (caso 3), attraverso il comando `trap` che ha la seguente sintassi:

```
trap "comandi" lista_dei_segnali
```

Se al processo che esegue lo script arriva un segnale in `lista_dei_segnali` allora vengono eseguiti i comandi tra virgolette.

Per ignorare un segnale si può usare la sintassi `trap "" lista_dei_segnali`.

Ecco ora alcuni tra i segnali più usati:

<u>Signal</u>	<u>Valore</u>	<u>Note</u>
hangup	1	Usato per uccidere i processi durante il logout
interrupt	2	Generato tramite Ctrl-c
quit	3	Generato tramite Ctrl-\
kill	9	Terminazione sicura del processo
terminate	15	Segnale di default del comando <code>kill</code>
USR1	10	Segnale di utente 1
USR2	12	Segnale di utente 2

Esempio: Il comando `trap` è utile per interrompere in modo corretto uno script, ad esempio eliminando i file temporanei eventualmente creati e che sarebbero stati rimossi solo alla conclusione naturale dello script.

Rivediamo lo script `sincro2` modificato. Se si digita `Ctrl-c` per interromperne l'esecuzione, il file `link.tmp` non viene rimosso dal file system.

Una versione che gestisce il segnale di interruzione (INT) dato esplicitamente o come `Ctrl-c` è:

```
#!/bin/sh

#          sincro2

trap "rm link.tmp; exit 0" 2 # Se viene premuto Ctrl-c prima di uscire elimina
                             # il file temporaneo

until test -e link.tmp      # Aspetta che il file link.tmp sia stato creato
do
    sleep 5                 # attende 5 secondi
done

echo                        # va a capo
cat -n testo.txt           # cat -n numera le linee (si poteva usare anche awk)

rm link.tmp
```

Riprendiamo l'esercizio:

Implementare uno script in grado di stampare un triangolo di numeri con il seguente formato:

```
1
2 3
4 5 6
7 8 9 10
.....
.....
```

Se l'utente usa la combinazione di tasti `Ctrl-c` per terminare lo script la linea correntemente in stampa deve essere comunque terminata

```
#!/bin/sh

fine=1          # fine=falso
trap "fine=0" 2 # Se si preme Ctrl-c (segnale 2) a fine è assegnato vero

i=1
acapo=1

while true
do
    j=0
    while [ $j -lt $acapo ]
    do
        echo -n "$i "
        i=$((i + 1))
        j=$((j + 1))
    done
done
```

```
done

echo ""
if test "$fine" -eq 0; then exit 0;fi
acapo=$((acapo + 1)
done
```

I necessari approfondimenti possono essere reperiti nel testo di riferimento del corso "Linux A-Z".

Molto utili anche gli Appunti Linux di Daniele Giacomini (un manuale di Linux completo e scaricabile gratuitamente da Internet (viene aggiornato molto spesso)).

Per gli appassionati di WINDOWS

Collegandosi al sito

<http://sourceware.cygnus.com/cygwin>

è possibile scaricare gratuitamente i classici programmi GNU (tra cui **gawk**) ed un ambiente operativo (shell) UNIX.

Collegandosi al sito <http://www.vmware.com/>

è possibile scaricare un applicativo che consente di ottenere un sistema Linux che esegue all'interno di una finestra Windows, o viceversa.