Sistemi Operativi

Laboratorio – linea 2

6

Lezione 6:

Unix power tools parte I

Sistemi Operativi

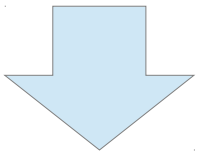
Laboratorio – linea 2

6

ls | sort

PIPE (1/2)

```
1  int main(void){
2          int fd[2], nbytes; pid_t childpid;
3          char string[] = "Hello, world!\n";
4          char readbuffer[80];
5
6          pipe(fd);
7          if(fork() == 0){
8                  /* Child process closes up input side of pipe */
9                  close(fd[0]);
10                 write(fd[1], string, (strlen(string)+1));
11                 exit(0);
12         } else {
13                 /* Parent process closes up output side of pipe */
14                 close(fd[1]);
15                 nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
16                 printf("Received string: %s", readbuffer);
17         }
18         return(0);}
```



Sistemi Operativi

Laboratorio – linea 2

6

PIPE (2/2)

```
1        if(fork() == 0)
2        {
3                /* Close up standard input of the child */
4                close(0);
5
6                /* Duplicate the input side of pipe to stdin */
7                dup(fd[0]);
8                execlp("sort", "sort", NULL);
9        }
```

UN VERO LINGUAGGIO DI PROGRAMMAZIONE

La shell è un vero (Turing-completo) linguaggio di programmazione (interpretato).

- **Variabili** (create al primo assegnamento, uso con $, export in un'altra shell)
  x=”ciao” **;** y=2 ; /bin/echo “$x $y $x”

- **Istruzioni condizionali** (valore di ritorno 0 indica true)
  if /bin/ls/piripacchio **;** then /bin/echo ciao**;** else /bin/echo buonasera; fi

- **Iterazione su insiemi**
  for i in a b c d e; do /bin/echo $i; done

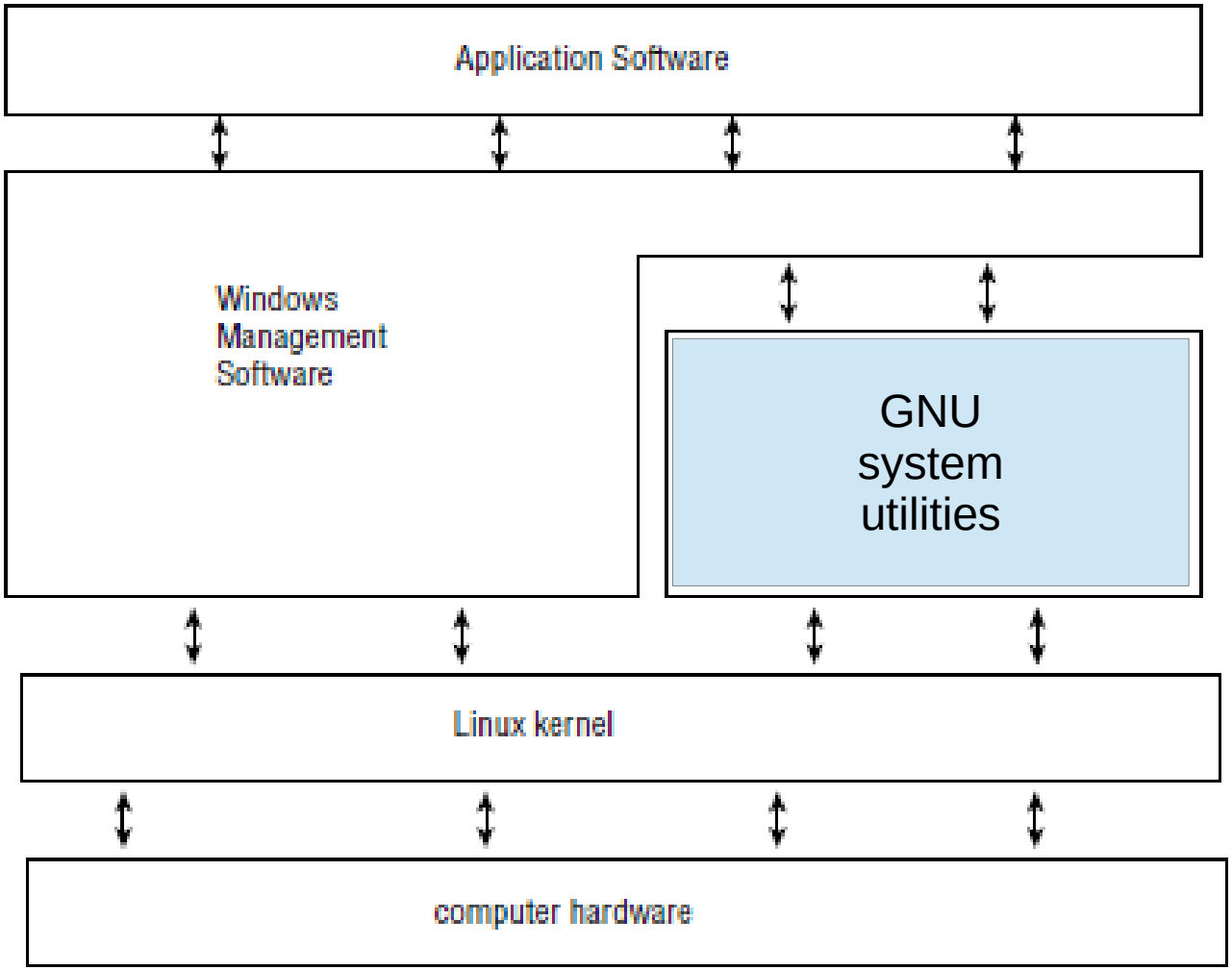
- **Cicli**

```
/usr/bin/touch piripacchio
while /bin/ls piripacchio; do
 /usr/bin/sleep 2
 /usr/bin/echo ciao
done & ( /usr/bin/sleep 10 ; /bin/rm piripacchio)
```

NB: l'interprete è la shell stessa:
/bin/bash

LINUX

The Linux system

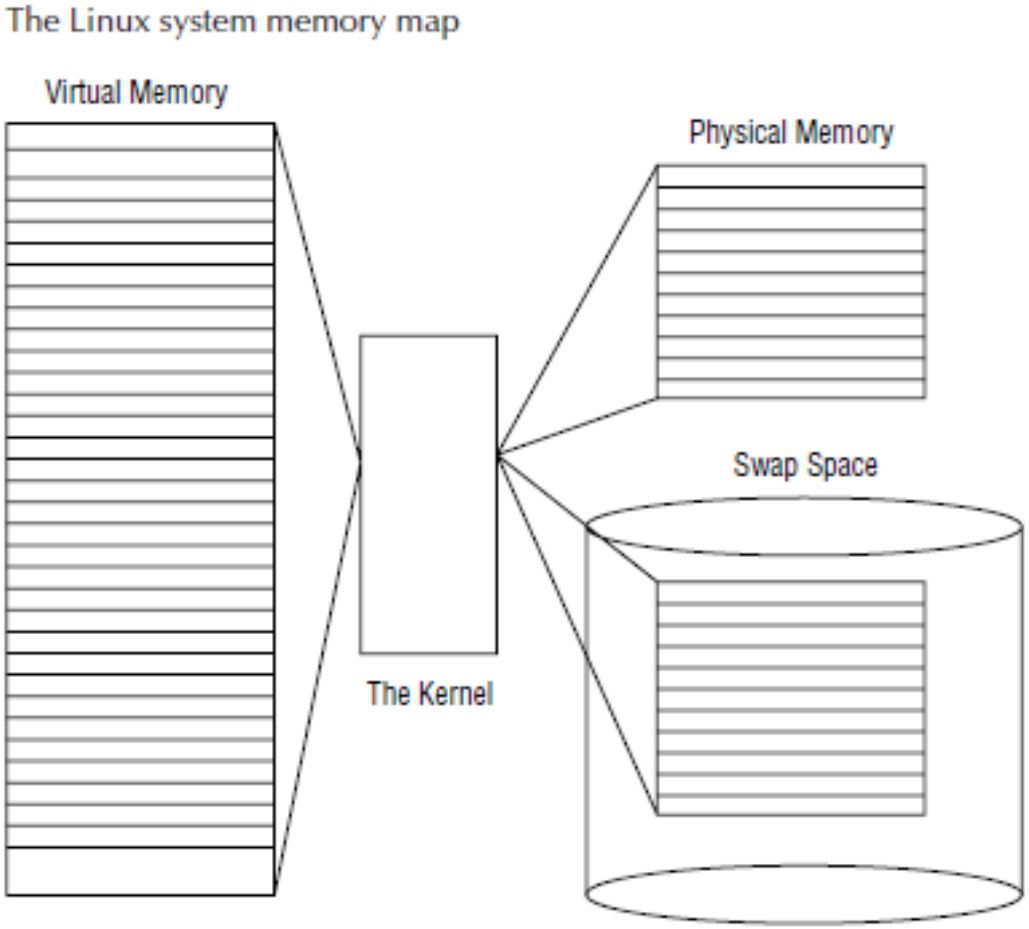


LINUX

Il cuore del sistema operativo è il kernel. Esso si occupa di gestire tutte le risorse hardware presenti nel calcolatore , allocandole in maniera da ottimizzarne l'utilizzo, e tutto il software, eseguendo determinati programmi in caso di necessità.

Il kernel è il principale responsabile per:

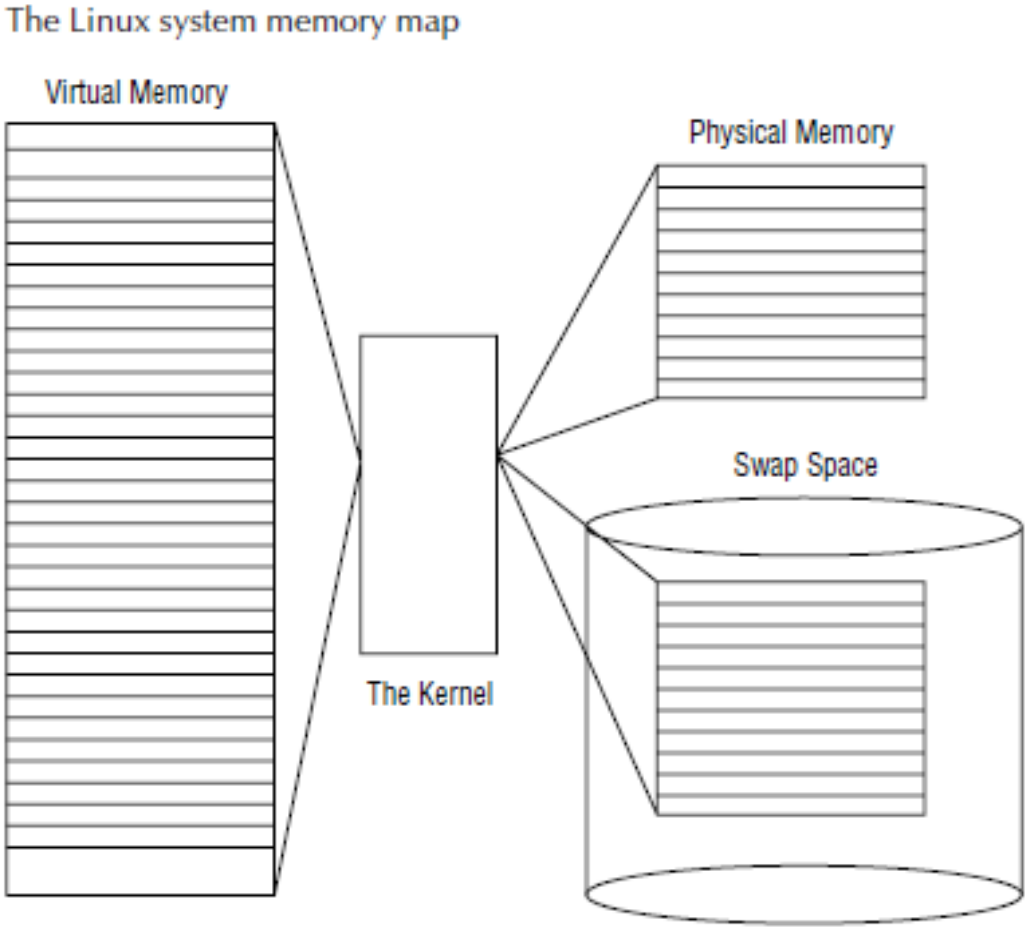
Gestione memoria
Gestione hardware
Gestione software
Gestione file system



LINUX

Il kernel si occupa di gestire la memoria fisica della macchina ma, oltre a questo, è anche in grado di creare (e gestire) una memoria “virtuale” … memoria che non esiste .

Per riuscirci utilizza uno spazio sul disco detto spazio di swap. Il kernel sposta dinamicamente locazioni di memoria dallo spazio di swap alla memoria fisica. In questo modo il kernel “simula” l'esistenza di una quantità di memoria superiore a quella fisicamente disponibile.



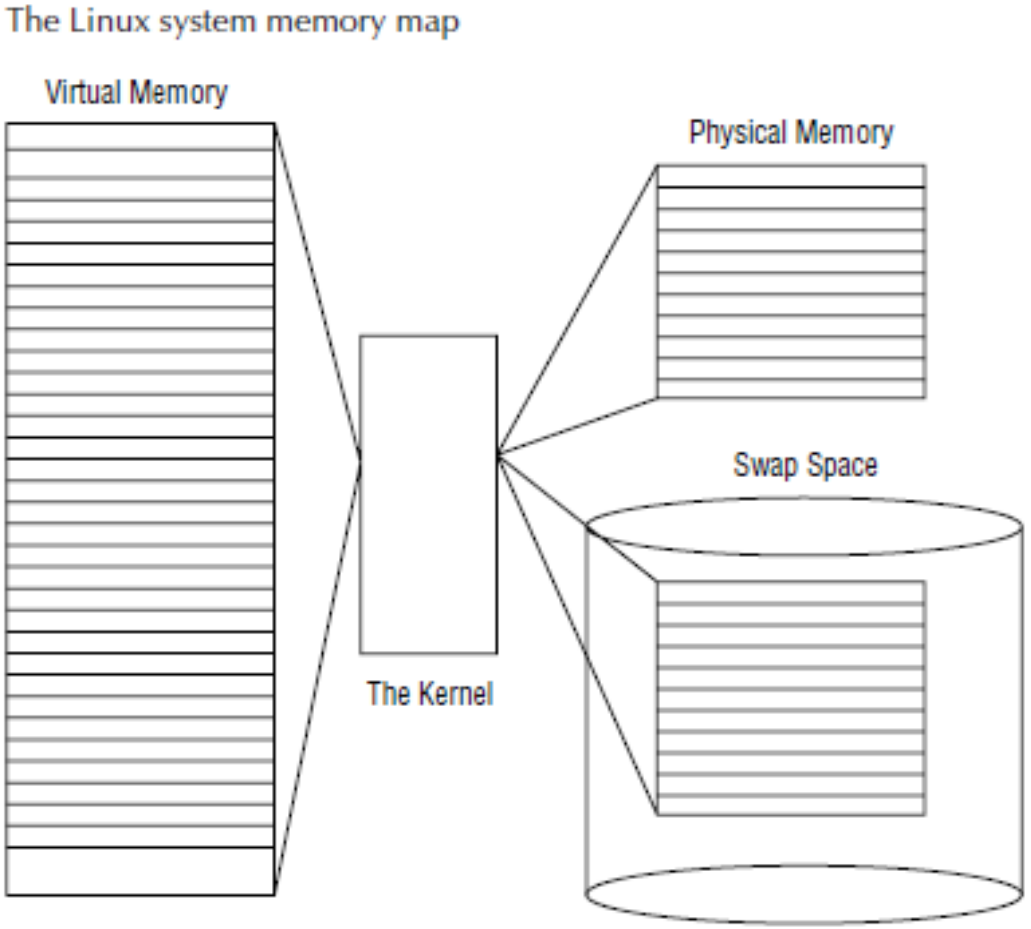
Il kernel si occupa di gestire la memoria fisica della macchina ma, oltre a questo, è anche in grado di creare (e gestire) una memoria “virtuale” … memoria che non esiste (fisicamente).

Per riuscirci utilizza uno spazio sul disco detto spazio di swap. Il kernel sposta dinamicamente locazioni di memoria dallo spazio di swap alla memoria fisica. In questo modo il kernel “simula” l'esistenza di una quantità di memoria superiore a quella fisicamente disponibile.

Le locazioni di memoria virtuale sono raggruppate in blocchi detti pagine. Il kernel può collocare le pagine di memoria sia nello swap che in memoria fisica. Il kernel mantiene una tabella che indica quali pagine di memoria sono in swap e quali in memoria fisica.

Informazioni sullo stato della memoria virtuale vengono registrate nel file **/proc/meminfo**

LINUX



$ cat nomefile stampa sullo schermo il contenuto di un file.

E' possibile utilizzare cat per indagare sullo stato della memoria (fisica e virtuale) del sistema semplicemente visualizzando il contenuto di /proc/meminfo !

Se, invece di voler scrivere l'output di un comando si STDOUT vogliamo scriverlo su un file utilizziamo la redirezione dell'output:

$ cat nomefile **>** nomenuovofile

```
# cat /proc/meminfo
MemTotal:       255392 kB
MemFree:          4336 kB
Buffers:          1236 kB
Cached:          48212 kB
SwapCached:       1028 kB
Active:         182932 kB
Inactive:        44388 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       255392 kB
LowFree:          4336 kB
SwapTotal:      524280 kB
SwapFree:       514528 kB
Dirty:              456 kB
Writeback:            0 kB
AnonPages:      176940 kB
Mapped:          40168 kB
Slab:            16080 kB
SReclaimable:     4048 kB
SUnreclaim:      12032 kB
PageTables:       4048 kB
NFS_Unstable:        0 kB
Bounce:              0 kB
CommitLimit:    651976 kB
Committed_AS:   442296 kB
VmallocTotal:   770040 kB
VmallocUsed:      3112 kB
VmallocChunk:   766764 kB
HugePages_Total:     0
```

Questo esempio serve a capire che la shell (unitamente alla conoscenza della posizione nel sistema dei file che contengono informazioni importanti) è uno strumento estremamente potente.

Di fatto la capacità di controllare appieno un sistema (in particolare un sistema della famiglia UNIX) dipende dalla nostra capacità di utilizzare la shell.

SHELL : BASI

ES1:

```
user@:~$  [ -f piripacchio ]
user@:~$  echo $?
1
user@:~$  touch piripacchio
user@:~$  [ -f piripacchio ]
user@:~$  echo $?
0
```

ES2:

```
user@:~$  test -f piripacchio
user@:~$  echo $?
0
user@:~$  rm piripacchio
user@:~$  test -f piripacchio
user@:~$  echo $?
1
```

Note:

---

touch nomefile : crea un file (vuoto), cambia timestamp
test -f nomefile : verifica l'esistenza di un file (TRUE vale 0 e FALSE vale 1)
[-f nomefile] : verifica l'esistenza di un file (se esiste ritorna 0)
$? : stampa il valore di ritorno dell'ultimo comando eseguito nella shell
man nomecomando : stampa il manuale contenente le Informazioni (ad es. parametri) su un determinato Comando

---

[**-f** /etc/passwd] && echo "**File** exists" || echo "**File** does not exists"

[**-d** /var/log] && echo "**Directory** exists" || echo "**Directory** does not exists"

VARIABILI DI AMBIENTE

La shell BASH utilizza le variabili di ambiente per tenere traccia di informazioni riguardanti la sessione corrente e l'ambiente di lavoro.

Queste variabili permettono anche all'utente di immagazzinare in **memoria** informazioni accessibili da qualsiasi programma che viene eseguito all'interno della shell. Alcune variabili di ambiente associate in modo particolare ad ogni specifico utente possono essere impostate durante l'avvio del sistema. Questo è un modo efficace di salvare informazioni inerenti a utenti, sistema, alla shell stessa in modo che siano facilmente accessibili.

Le variabili d'ambiente che vengono utilizzate a livello di sistema hanno nomi formati da lettere maiuscole (convenzione).

VARIABILI DI AMBIENTE

Le variabili d'ambiente definite a livello di sistema (variabili d'ambiente **GLOBALI**) possono essere visualizzate utilizzando il comando **printenv** :

```
$ printenv
HOSTNAME=testbox.localdomain
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=192.168.1.2 1358 22
OLDPWD=/home/rich/test/test1
SSH_TTY=/dev/pts/0
USER=rich
LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:
bd=40;33;01:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=00;32:
*.cmd=00;32:*.exe=00;32:*.com=00;32:*.btm=00;32:*.bat=00;32:
*.sh=00;32:*.csh=00;32:*.tar=00;31:*.tgz=00;31:*.arj=00;31:
*.taz=00;31:*.lzh=00;31:*.zip=00;31:*.z=00;31:*.Z=00;31:
*.gz=00;31:*.bz2=00;31:*.bz=00;31:*.tz=00;31:*.rpm=00;31:
*.cpio=00;31:*.jpg=00;35:*.gif=00;35:*.bmp=00;35:*.xbm=00;35:
*.xpm=00;35:*.png=00;35:*.tif=00;35:
MAIL=/var/spool/mail/rich
PATH=/usr/kerberos/bin:/usr/lib/ccache:/usr/local/bin:/bin:/usr/bin:
/home/rich/bin
INPUTRC=/etc/inputrc
PWD=/home/rich
LANG=en_US.UTF-8
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SHLVL=1
HOME=/home/rich
LOGNAME=rich
```

Variabili d'ambiente **globali**:

visibili dall'interno della shell e di qualsiasi processo (anche altre shell) eseguiti all'interno della shell originale

VARIABILI DI AMBIENTE

Le variabili d'ambiente valide solo all'interno di una determinata sessione shell vengono dette variabili d'ambiente **LOCALI**. Non esiste un comando dedicato per la visualizzazione delle sole variabili d'ambiente locali … è però possibile utilizzare il comando **set** senza argomenti:

ES3:

```
user@:~$  vloc1=”a”
user@:~$  set > prova1.txt
user@:~$  bash
user@:~$  vloc2=”b”
user@:~$  set > prova2.txt
user@:~$  exit
```

Apertura nuova sessione shell

Chiusura sessione shell

Esaminare (vi) il contenuto dei file prova1.txt e prova2.txt . Dove sono vloc1 e vloc2 ? Quali conclusioni potete trarne?

VARIABILI DI AMBIENTE

Ora vedremo come creare ed eliminare variabili d'ambiente locali e globali.

ES4:

```
user@:~$  var1=”test”
user@:~$  echo $var1
test
user@:~$  unset var1
user@:~$  echo $var1

user@:~$  var1=”globaltest”
user@:~$  export var1
user@:~$  printenv
user@:~$  unset var1
```

Creazione variabile (locale)

Distruzione variabile

Esporta variabile locale (diventa globale)

CARATTERI SPECIALI

\# commento

Esempi:

echo "Seguirà un commento." # Qui il commento

Commento posto dopo commando
Notate spazio prima del #

Commenti , apici , sostituzioni, conversioni:

```
echo "Il presente  # non inizia un commento."
echo 'Il presente  # non inizia un commento.'
echo Il presente  \# non inizia un commento.
echo Il presente   # inizia un commento.

echo ${PATH#*:}       # È una sostituzione di parametro, non un commento.
echo $(( 2#101011 ))  # È una conversione di base, non un commento.
```

NB: I caratteri ‘ \ ” evitano la reinterpretazione di # come inizio commento !

CARATTERI SPECIALI

; separatore comandi

Permette di utilizzare più comandi sulla stessa riga

Esempi:

echo ‘stringa1’ ; echo ‘stringa2’

```
echo ehilà; echo ciao
```

```
if [ -f "$nomefile" ]; then     #  Notate che "if" e "then" hanno bisogno del
                                #+ punto e virgola. Perché?
  echo "Il file $nomefile esiste."; cp $nomefile $nomefile.bak
else
  echo "$nomefile non trovato."; touch $nomefile
fi; echo "Verifica di file completata."
```

CARATTERI SPECIALI

;; delimitatore opzione case

```
case "$variabile" in
abc)  echo "\$variabile = abc" ;;
xyz)  echo "\$variabile = xyz" ;;
esac
```

CARATTERI SPECIALI

. **(punto)** È un builtin BASH. Equivale a **source**. Se volete caricare un file contenente, ad esempio, variabili, potete scrivere . nomefile . Si utilizza negli script. Il file da caricare deve essere presente in PWD (directory di lavoro corrente).

WARNING:

In linux/unix il punto nel nome dei file ha un significato special: se il nome del file inizia con . Allora il file è un file NASCOSTO . Si può visualizzare informazioni su di esso mediante ls –l**a (a è un parametro che dice a ls di mostrare tutti I file (all) … anche quelli nascosti).**

NB:
Il punto . si utilizza anche nelle ricerche di carattere. In questo caso il punto indica UN SOLO carattere!

“ **(quoting parziale, doppio apice)** Il doppio apice preserva il contenuto, stringa, evitando che caratteri speciali al suo interno vengano interpretati come tali. Anche apice singolo ‘ ha lo stesso effetto ma è più forte (più restrittivo) di apice doppio.

CARATTERI SPECIALI

, **(virgola)** Concatena una serie di operazioni aritmetiche. Vengono valutate tutte ma viene restituita solo l'ultima!

```
let "t2 = ((a = 9, 15 / 3))"  # Imposta "a" e "t2 = 15 / 3".
```

**** (escape, barra inversa) Effettua il quoting (preserva) caratteri singoli :
\ viene utilizzato spesso per il quoting di **"** e **'** in modo che essivengano interpretati
letetralmente

` **(apice inverso, sostituzione commando)** restituisce il risultato del commando posto tra apici inversi in modo da renderlo disponibile per l'assegnamento ad una variabile.

: **(commando NULL, due punti)** è l'equivalente nell'istruzione NOP, non far niente. E' da considerarsi un sinonimo del comandi di shell **true**.

```
while :
      do
         ISTRUZIONI
      done
```

ciclo infinito

```
if  condizione
 then :
 else
         ISTRUZIONI
 fi
```

non fare nulla

CARATTERI SPECIALI

: **(commando NULL, due punti)** è l'equivalente nell'istruzione NOP, non far niente. E' da considerarsi un sinonimo del comandi di shell **true**.

In combinazione con l'operatore di redirezione > è in grado di azzerare il contenuto. Ha lo stesso effetto di un cat /dev/null > nomefile .

```
: > nomefile.xxx  # ora nomefile.xxx è vuoto
```

! **(inversione o negazione)** Inverte il significato di un commando o di un test logico, ad esempio **!=** è "non uguale" (quindi "diverso").

$ **(sostituzione di variabile)** $nomevar accede al **contenuto** della variabile nomevar.

$$ **(PID dello script in esecuzione)**

CARATTERI SPECIALI

! **(inversione o negazione)** Inverte il significato di un commando o di un test logico, ad esempio **!=** è “non uguale” (quindi “diverso”).

$ **(sostituzione di variabile)** $nomevar accede al **contenuto** della variabile nomevar.

$$ **(PID dello script in esecuzione)**

***** **(jolly espansione MULTIcarattere)**
? **(jolly SINGOLO carattere)** cas? da match con cas**a** e cas**o**

( ) gruppo di comandi es. (a=ciao; echo $a) . Usate anche per inizializzare array: a = (var1 var2)

PERICOLO!!!
Le parentesi () danno origine ad una **subshell** … quindi le variabili LOCALI alla parent shell **NON SONO VISIBILI** nella «child» subshell.

Notare spazio

CARATTERI SPECIALI

{} (espansione multipla)

cat {file1,file2,file3} > nomenuovofile
cp nomefile.{txt,bkp} equivale a cp nomefile.txt nomefile.bkp

Viene utilizzato dal comando ***xargs*** per creare liste di argomenti

COSTRUTTI CONDIZIONALI

- Il costrutto **if/then** verifica se l'exit status di un elenco di comandi è 0 (perché 0 significa "successo" per convenzione UNIX) e se questo è il caso, esegue uno o più comandi.
- Esiste il comando specifico **[** (parentesi quadra aperta). È sinonimo di **test** ed è stato progettato come builtin per ragioni di efficienza. Questo comando considera i suoi argomenti come espressioni di confronto, o di verifica di file, e restituisce un exit status corrispondente al risultato del confronto (0 per vero, 1 per falso).
- Con la versione 2.02, Bash ha introdotto [[...]], *comando di verifica estesa*, che esegue confronti in un modo più familiare ai programmatori in altri linguaggi. Va notato che **[[** è una parola chiave, non un comando.

Bash vede `[[ $a -lt $b ]]` come un unico elemento che restituisce un exit status.

Anche i costrutti ((...)) e let ... restituiscono exit status 0 se le espressioni aritmetiche valutate sono espanse ad un valore diverso da zero. Questi costrutti di espansione aritmetica possono, quindi, essere usati per effettuare confronti aritmetici.

```
let "1<2" restituisce 0 (poiché "1<2" espande a "1")
(( 0 && 1 )) restituisce 1 (poiché "0 && 1" espande a "0")
```

COSTRUTTI CONDIZIONALI

Un costrutto **if** può verificare qualsiasi comando, non solamente le condizioni comprese tra le parentesi quadre.

```
if cmp a b &> /dev/null  # Sopprime l'output.
then echo "I file a e b sono identici."
else echo "I file a e b sono diversi."
fi

# L'utilissimo costrutto "if-grep":
# ---------------------------------
if grep -q Bash file
then echo "Il file contiene almeno un'occorrenza di Bash."
fi

parola=Linux
sequenza_lettere=inu
if echo "$parola" | grep -q "$sequenza_lettere"
# L'opzione "-q" di grep elimina l'output.
then
  echo "$sequenza_lettere trovata in $parola"
else
  echo "$sequenza_lettere non trovata in $parola"
fi
```

COSTRUTTI CONDIZIONALI

- Un costrutto **if/then** può contenere confronti e verifiche annidate.

```
if echo "Il prossimo *if* è parte del costrutto del primo *if*."

  if [[ $confronto = "intero" ]]
    then (( a < b ))
  else
    [[ $a < $b ]]
  fi

then
  echo '$a è inferiore a $b'
fi
```

COSTRUTTI CONDIZIONALI

```
if [ condizione-vera ]
then
    comando 1
    comando 2
    ...
else
    #  Opzionale (può anche essere omesso).
    #  Aggiunge un determinato blocco di codice che verrà eseguito se la
    #+ condizione di verifica è falsa.
    comando 3
    comando 4
    ...
fi
```

COSTRUTTI CONDIZIONALI

Nota: Quando *if* e *then* sono sulla stessa riga occorre mettere un punto e virgola dopo l'enunciato *if* per indicarne il termine. Sia *if* che *then* sono parole chiave. Le parole chiave (o i comandi) iniziano gli enunciati e prima che un nuovo enunciato possa incominciare, sulla stessa riga, è necessario che il precedente venga terminato.

```
if [ -x "$nome_file" ]; then
```

---

elif è la contrazione di else if. Lo scopo è quello di annidare un costrutto if/then in un altro.

```
if [ condizione1 ]
then
   comando1
   comando2
   comando3
elif [ condizione2 ]
# Uguale a else if
then
   comando4
   comando5
else
   comando-predefinito
fi
```

COSTRUTTI CONDIZIONALI

Il costrutto **`if test condizione-vera`** è l'esatto equivalente di **`if [ condizione-vera ]`**. In quest'ultimo costrutto, la parentesi quadra sinistra **[**, è un simbolo che invoca il comando **test**. La parentesi quadra destra di chiusura, **]**, non dovrebbe essere necessaria. Ciò nonostante, le più recenti versioni di Bash la richiedono.

Nota: Il comando **test** è un builtin Bash che verifica i tipi di file e confronta le stringhe. Di conseguenza, in uno script Bash, **test** *non* richiama l'eseguibile esterno `/usr/bin/test`, che fa parte del pacchetto *sh-utils*. In modo analogo, **[** non chiama `/usr/bin/[`, che è un link a `/usr/bin/test`.

```
bash$ type test
test is a shell builtin
bash$ type '['
[ is a shell builtin
bash$ type '[['
[[ is a shell keyword
bash$ type ']]'
]] is a shell keyword
bash$ type ']'
bash: type: ]: not found
```

COSTRUTTI CONDIZIONALI

Il costrutto [[ ]] è la versione Bash più versatile di [ ]. È il *comando di verifica esteso*, adottato da *ksh88*.

Nota: Non può aver luogo alcuna espansione di nome di file o divisione di parole tra [[e]], mentre sono consentite l'espansione di parametro e la sostituzione di comando.

```
file=/etc/passwd

if [[ -e $file ]]
then
  echo "Il file password esiste."
fi
```

Suggerimento: L'utilizzo del costrutto di verifica **[[...]]** al posto di **[...]** può evitare molti errori logici negli script. Per esempio, gli operatori &&, ||, < e > funzionano correttamente in una verifica [[ ]], mentre potrebbero dare degli errori con il costrutto [ ] .

CONFRONTO DI INTERI [...]

è uguale a

```
if [ "$a" -eq "$b" ]
```

è diverso (non uguale) da

```
if [ "$a" -ne "$b" ]
```

è maggiore di

```
if [ "$a" -gt "$b" ]
```

è maggiore di o uguale a

```
if [ "$a" -ge "$b" ]
```

è minore di

```
if [ "$a" -lt "$b" ]
```

è minore di o uguale a

```
if [ "$a" -le "$b" ]
```

CONFRONTO DI INTERI ((...))

è minore di (tra doppie parentesi)

```
(("$a" < "$b"))
```

è maggiore di o uguale a (tra doppie parentesi)

```
(("$a" >= "$b"))
```

è minore di o uguale a (tra doppie parentesi)

```
(("$a" <= "$b"))
```

è maggiore di (tra doppie parentesi)

```
(("$a" > "$b"))
```

CONFRONTO DI STRINGHE

è uguale a

```
if [ "$a" = "$b" ]
```

è diverso (non uguale) da

```
if [ "$a" != "$b" ]
```

All'interno del costrutto [[...]] questo operatore esegue la ricerca di corrispondenza.

è uguale a

```
if [ "$a" == "$b" ]
```

è inferiore a, in ordine alfabetico ASCII

```
if [[ "$a" < "$b" ]]
```

```
if [ "$a" \< "$b" ]
```

Si noti che "<" necessita dell'escaping nel costrutto `[ ]`.

CONFRONTO DI STRINGHE

è maggiore di, in ordine alfabetico ASCII

```
if [[ "$a" > "$b" ]]
```

```
if [ "$a" \> "$b" ]
```

Si noti che “>” necessita dell’escaping nel costrutto `[ ]`.

CONFRONTI COMPOSTI

-a

and logico

exp1 `-a` *exp2* restituisce vero se *entrambe* exp1 e exp2 sono vere.

-o

or logico

exp1 `-o` *exp2* restituisce vero se è vera o exp1 *o* exp2.

Sono simili agli operatori di confronto Bash **&&** e || utilizzati all'interno delle doppie parentesi quadre.

```
[[ condizione1 && condizione2 ]]
```

Gli operatori **-o** e **-a** vengono utilizzati con il comando **test** o all'interno delle parentesi quadre singole.

```
if [ "$exp1" -a "$exp2" ]
```

CICLI

cicli for

for `arg` **in** *`[lista]`*

È il costrutto di ciclo fondamentale. Differisce significativamente dal suo analogo del linguaggio C.

```
for arg in [lista]
do
  comando(i)...
done
```

Nota: Ad ogni passo del ciclo, *`arg`* assume il valore di ognuna delle successive variabili elencate in *`lista`*.

ES CICLI for

```
#!/bin/bash
# Elenco di pianeti.

for pianeta in Mercurio Venere Terra Marte Giove Saturno Urano Nettuno Plutone
do
  echo $pianeta  # Ogni pianeta su una riga diversa
done

echo

for pianeta in "Mercurio Venere Terra Marte Giove Saturno Urano Nettuno Plutone"
# Tutti i pianeti su un'unica riga.
# L'intera "lista" racchiusa tra apici doppi crea un'unica variabile.
do
  echo $pianeta
done

exit 0
```

Generate [lista] in un ciclo for con una sostituzione di comando

```
#!/bin/bash
#  for-loopcmd.sh: un ciclo for con [lista]
#+ prodotta dalla sostituzione di comando.

NUMERI="9 7 3 8 37.53"

for numero in `echo $NUMERI`  # for numero in 9 7 3 8 37.53
do
  echo -n "$numero "
done

echo
exit 0
```

CICLI

while

Questo costrutto verifica una condizione data all'inizio del ciclo che viene mantenuto in esecuzione finché quella condizione rimane vera (restituisce exit status 0). A differenza del ciclo for, il *ciclo while* viene usato in quelle situazioni in cui il numero delle iterazioni non è conosciuto in anticipo.

```
while [condizione]
do
  comando...
done
```

Come nel caso dei *cicli for*, collocare il *do* sulla stessa riga della condizione di verifica rende necessario l'uso del punto e virgola.

```
while [condizione]; do
```

ES ciclo while

```
#!/bin/bash

var0=0
LIMITE=10

while [ "$var0" -lt "$LIMITE" ]
do
  echo -n "$var0 "        # -n sopprime il ritorno a capo.
  #             ^          Lo spazio serve a separare i numeri visualizzati.
  var0=`expr $var0 + 1`    # var0=$(($var0+1))  anche questa forma va bene.
                           # var0=$((var0 + 1)) anche questa forma va bene.
                           # let "var0 += 1"    anche questa forma va bene.
done                       # Anche vari altri metodi funzionano.

echo

exit 0
```

Sistemi Operativi

Laboratorio – linea 2

6

ESERCIZI

1. Per ciascuno dei file `dog`, `cat`, `fish` controllare se esistono nella directory `bin` (hint: usare `/bin/ls` e nel caso scrivere ``Trovato'')
2. Consultare il manuale (programma `/usr/bin/man`) del programma `/bin/test` (per il manuale `man test`)
3. Riscrivere il primo esercizio facendo uso di `test`

Input e Output

In generale il paradigma UNIX permette alle applicazioni di fare I/O tramite:

Input

- Parametri al momento del lancio
- Variabili *d'ambiente*
- File (tutto ciò che può essere gestito con le syscall `open`, `read`, `write`, `close`)
  - Terminale (interfaccia testuale)
  - Device (per es. il mouse potrebbe essere `/dev/mouse`)
  - Rete (socket)

Output

- Valore di ritorno
- Variabili *d'ambiente*
- File (tutto ciò che può essere gestito con le syscall `open`, `read`, `write`, `close`)
  - Terminale (interfaccia testuale)
  - Device (per es. lo schermo in modalità grafica potrebbe essere `/dev/fb`)
  - Rete (socket)

REDIREZIONI

Ad ogni processo sono sempre associati tre file (già aperti)

- Standard input (Terminale, tastiera)
- Standard output (Terminale, video)
- Standard error (Terminale, video, usato per le segnalazione d'errore)

Possono essere *rediretti*

- /usr/bin/sort < lista Lo stdin è il file `lista`
- /bin/ls > lista Lo stdout è il file `lista`
- /bin/ls piripacchio 2> lista Lo stderr è il file `lista`
- (echo ciao & date ; ls piripacchio) 2> errori 1>output

PIPE

La pipe è un canale, analogo ad un file, bufferizzato in cui un processo scrive e un altro legge. Con la shell è possibile collegare due processi tramite una pipe anonima.
Lo stdout del primo diventa lo stdin del secondo

/bin/ls | sort

ls −lR / | sort | more

funzionalmente equivalente a

ls −lR >tmp1; sort <tmp1 >tmp2; more<tmp2; rm tmp∗

Molti programmi copiano lo stdin su stdout dopo averlo elaborato: sono detti filtri.

COMMAND SUBSTITUTION

Con una pipe è possibile "collegare" lo stdout di un programma con lo stdin di un altro.

Per usare l'output di un programma sulla riga di comando di un altro programma, occorre usare la command substitution

```
/bin/ls -l $(/usr/bin/which sort)
```

Sistemi Operativi

Laboratorio – linea 2

6

ESERCIZI

1. Verificare qual è il valore di ritorno di una *pipeline*, anche in caso che qualcuno dei "filtri" fallisca.
2. Scrivere una *pipeline* di comandi che identifichi il le informazioni sul processo `dropbear` (`ps`, `grep`)
3. Scrivere una *pipeline* di comandi che identifichi il solo processo con il PPID piú alto (`ps`, `sort`, `tail`)
4. Ottenere il numero totale dei file contenuti nelle directory `/usr/bin` e `/var` (`ls`, `wc`, `expr`)
5. Si immagini di avere un file contenente il sorgente di un programma scritto in un linguaggio di programmazione in cui i commenti occupino intere righe che iniziano con il carattere #. Scrivere una serie di comandi per ottenere il programma senza commenti. (`grep`)
6. Ottenere la somma delle occupazioni dei file delle directory `/usr/bin` e `/var` (`du`, `cut`)

ESERCIZI

2. Scrivere una *pipeline* di comandi che identifichi il le informazioni sul processo `dropbear` (`ps`, `grep`)

Apparentemente il problema potrebbe essere risolto dal solo utilizzo di ps e grep ma … quante righe otteniamo con questa pipeline?

```
ps aux | grep 'dropbear'
```

Dovreste ottenere 2 righe (la seconda cosa contiene? Perchè la otteniamo?). Per verificare usate:

```
ps aux | grep 'dropbear' | wc -l
```

Vogliamo solo la prima!

```
ps aux | grep 'dropbear' | head -n1
```

ESERCIZI

3. Scrivere una *pipeline* di comandi che identifichi il solo processo con il PPID piú alto (`ps`, `sort`, `tail`)

Ancora ps …

ps -eo uname,pid,ppid,args

all

formato output

Passiamo il tutto (via pipe) a sort -nk3

numerical sort

column 3

ordinamento numerico crescente sulla base del contenuto di Colonna 3 (PPID)

ps -eo uname,pid,ppid,args | sort -nk3

Vogliamo il processo con PPID più alto … quindi l'ultimo (dato l'utilizzo di sort). Passiamo il tutto (via pipe) a tail -n1

numero di righe prelevate dalla coda del file

ps -eo uname,pid,ppid,args | sort -nk3 | tail -n1

DOMANDA: Il processo che abbiamo ottenuto è l'unico ad avere questo PPID?

ESERCIZI

4. Ottenere il numero totale dei file contenuti nelle directory `/usr/bin` e `/var` (`ls`, `wc`, `expr`)

Partiamo da ls … ed eseguiamo alcuni test.

in alternativa potete provare ls -l /usr/bin | less

ls -l /usr/bin > prova1.txt ora apriamo il file con vi. Quante righe ci sono?

Dobbiamo trovare un modo per ottenere il numero di file in una directory. L'esercizio suggerisce l'utilizzo di ls e wc . Il comando wc può contare sia il numero di righe che il numero di parole. Secondo voi quale delle seguenti pipeline di comandi restituisce l'effettivo numero di file in /usr/bin ?

ls -l | wc -l
ls | wc -w

Motivate la risposta ...

ESERCIZI

4. Ottenere il numero totale dei file contenuti nelle directory /usr/bin e /var (ls, wc, expr)

Ora siamo in grado di ottenere il numero di file presenti in una data directory. Ma dobbiamo ancora sommare il numero di file presenti in /usr/bin e /var. Per ottenere questo risultato utilizziamo il comando expr e la command substitution

L'idea è quella di utilizzare la command substitution per sostituire I due argomenti da passare al comando expr per effettuare la somma:

expr ARG1 + ARG2

expr $(ls /usr/bin/ | wc -w) + 0 (verificate che il numero di file sia coerente con la slide prec.)

expr $(ls /usr/bin | wc -w) + $(ls /var | wc -w)

DOMANDE: Quale è il risultato?
Siete soddisfatti del risultato? (suggerimento... andate in usr/bin e in /var e indagate utilizzando ls -l)

ESERCIZI

4. Ottenere il numero totale dei file contenuti nelle directory /usr/bin e /var (ls, wc, expr)

Appare evidente che i “file” contenuti in /usr/bin e in /var non sono tutti regular file (alcuni sono link e altri sono directory...)

Proviamo di nuovo cercando di ottenere il numero dei soli regular file. Dovremo utilizzare un filtro (ad esempio grep).

expr $(ls **-l** /usr/bin | grep **^-** | wc **-l**) + $(ls **-l** /var | grep **^-** | wc **-l**)

DOMANDA: quanti sono I file regolari contenuti in /usr/bin e in /var ?
Se vogliamo la somma di tutti I file **filtrando** le **d**irectory possiamo procedere così:

expr $(ls **-l** /usr/bin | grep **-v ^d** | wc **-l**) + $(ls **-l** /var | grep **-v ^d** | wc **-l**)

DOMANDA2: Siamo sicuri che l'ultimo esempio sia giusto? (vi ricordate di ls -l /usr/bin | less ?)
In effetti contiene un errore. Sapreste trovarlo?

ESERCIZI

4. Ottenere il numero totale dei file contenuti nelle directory `/usr/bin` e `/var` (`ls`, `wc`, `expr`)

Un problema di questo approccio è che ls, con I parametri che abbiamo utilizzato fin qui, non esplora il contenuto delle sottodirectory (usatela redirezione per salvare l'output di ls -l /var e di ls -Rl /var per endervene conto). Purtroppo l'utilizzo diretto dell'opzione recursive (R) di ls produce righe vuote e righe di intestazione per le sottodirectory impedendo l'utilizzo diretto di in pipe con wc -l. Una possibile soluzione è l'utilizzo dell'opzione R di ls e di grep :

ls -**R**l /var | grep **^-** | wc -l

ESERCIZI

5. Si immagini di avere un file contenente il sorgente di un programma scritto in un linguaggio di programmazione in cui i commenti occupino intere righe che iniziano con il carattere #. Scrivere una serie di comandi per ottenere il programma senza commenti. (`grep`)

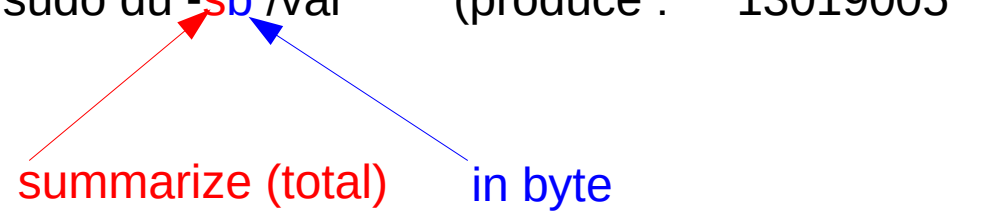
Si potrebbe risolve utilizzando un filtro (magari grep con l'opzione di esclusione -v):

grep **-v** '#'

ESERCIZI

6. Ottenere la somma delle occupazioni dei file delle directory /usr/bin e /var (du, cut)

du -sb /var (problemi di permessi)
sudo du -sb /var (produce : 13019005 /var/)



Dobbiamo estrarre solo il primo campo dalla riga ottenuta. Mandiamo tutto (via pipe) a cut :

sudo du -sb /var | cut -f1

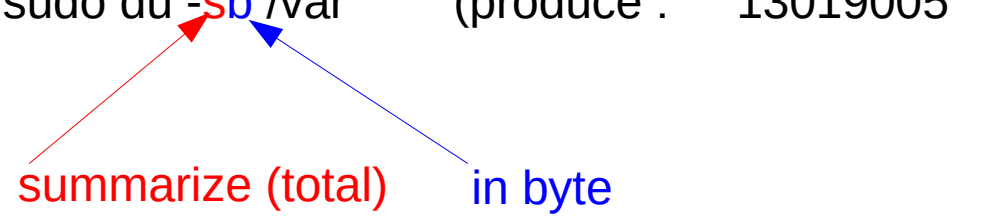
a questo punto possiamo procedere con expr (niente pipe, command substitution) …

expr $(sudo du -sb /var | cut -f1) + $(sudo du -sb /var | cut -f1)

ESERCIZI

6. Ottenere la somma delle occupazioni dei file delle directory /usr/bin e /var (du, cut)

du -sb /var (problemi di permessi)
sudo du -sb /var (produce : 13019005 /var/)



Dobbiamo estrarre solo il primo campo dalla riga ottenuta. Mandiamo tutto (via pipe) a cut :

sudo du -sb /var | cut -f1

a questo punto possiamo procedere con expr (niente pipe, command substitution) …

expr $(sudo du -sb /var | cut -f1) + $(sudo du -sb /var | cut -f1)

ESERCIZI

1. Verificare qual è il valore di ritorno di una *pipeline*, anche in caso che qualcuno dei "filtri" fallisca.

Questa domanda la affrontiamo per ultima … in effetti sembra così facile (e invece... :)

Intuitivamente il modo più semplice di verificare l'exit status (valore di uscita) di una pipeline sembrerebbe essere quello di usare un semplice:

echo $?

Ma cosa otteniamo in questo modo? Solo lo stato di uscita dell'ultimo comando della pipeline ine effetti. Rivediamo un esempio di qualche slide fa :

du -sb /var | cut -f1

che produce:

```
user@:~$ du -sb /var | cut -f1
du: cannot read directory '/var/cache/ldconfig': Permission denied
13011056
```

ESERCIZI

1. Verificare qual è il valore di ritorno di una *pipeline*, anche in caso che qualcuno dei "filtri" fallisca.

Ora … quello che dovrebbe metterci in allarme è il termine "qualcuno" … quale? E soprattutto … è possibile verificare l'exit status di ogni comando della pipeline?

Come abbiamo visto du -sb /var | cut -f1 non produce solo il risultato … ma anche un messaggio di errore! Eppure, se verifichiamo l' "exit status" della pipeline come ci eravamo proposti di fare otteniamo :

echo $?
0

...nessun errore. Il che ha senso poichè l'ultimo comando che abbiamo eseguito (cut) ha funzionato.

Quindi dove possiamo trovare traccia dell'errore avvenuto durante l'esecuzione di du -sb /var ?

Provate (dopo aver rieseguito la pipeline): echo ${PIPESTATUS[0]} ${PIPESTATUS[1]}

Ottenere lista di tutti i file *.h contenenti un numero di righe pari in una data directory

```
#!/usr/bin/bash

mycounter=0
lstarget='/usr/include/*.h'

for i in `ls $lstarget`; do
     nrows=`wc -l $i | cut -d' ' -f1`
     mytest=$(( $nrows % 2 ))
     if [ $mytest -eq 0 ] ; then
     echo $nrows $i
     mycounter=$(($mycounter+1))
     Fi
done

echo "file con numero di righe pari : "
echo $mycounter
```

Ottenere numero di tutti i file *.h contenenti un numero di righe pari in una data directory

ls -1 /usr/include/*.h | xargs wc -l | sed 's/\s\+/ /g' | cut -d' ' -f2 | xargs -I {} expr {} % 2 | grep ^0 | wc -l

questa è una i maiuscola!

Sistemi Operativi

Laboratorio – linea 2

6

TABELLA RIASSUNTIVA

`ls (1)`	list directory contents
`echo (1)`	display a line of text
`touch (1)`	change file timestamps
`sleep (1)`	delay for a specified amount of time
`rm (1)`	remove files or directories
`cat (1)`	concatenate files and print on the standard output
`man (1)`	an interface to the on-line reference manuals
`test (1)`	check file types and compare values
`sort (1)`	sort lines of text files
`date (1)`	print or set the system date and time
`less (1)`	file perusal filter for crt viewing
`which (1)`	locate a command
`ps (1)`	report a snapshot of the current processes.
`tail (1)`	output the last part of files
`wc (1)`	print the number of newlines, words, and bytes in files
`dc (1)`	An arbitrary precision calculator language
`grep (1)`	print lines matching a pattern
`cut (1)`	remove sections from each line of files
`du (1)`	print disk usage

SHELL E FILE SYSTEM

- Ogni processo (compresa la shell stessa) ha associata una *directory di lavoro* (working directory), che può essere cambiata col comando (interno alla shell) cd
- I programmi fondamentali per operare sul file system

ls (1)	list directory contents
cp (1)	copy files and directories
rm (1)	remove files or directories
mv (1)	move (rename) files
mkdir (1)	make directories
rmdir (1)	remove empty directories
df (1)	report file system disk space usage
du (1)	estimate file space usage
pwd (1)	print name of current/working directory

PERMESSI

Ad ogni file vengono associati dei *permessi*, che definiscono le azioni permesse sui dati del file

111 → 7
100 → 4

- **Read:** leggere il contenuto del file o directory
- **Write:** scrivere (cambiare) il file o directory
- **eXecute** eseguire le istruzioni contenute nel file o accedere alla directory

R	W	X	
1	1	0	6
1	0	1	5
1	0	0	4
1	1	1	7

I permessi possono essere diversi per 3 categorie di utenti del sistema:

- **User:** il "proprietario" del file
- **Group:** gli appartenenti al gruppo proprietario
- **All:** tutti gli altri

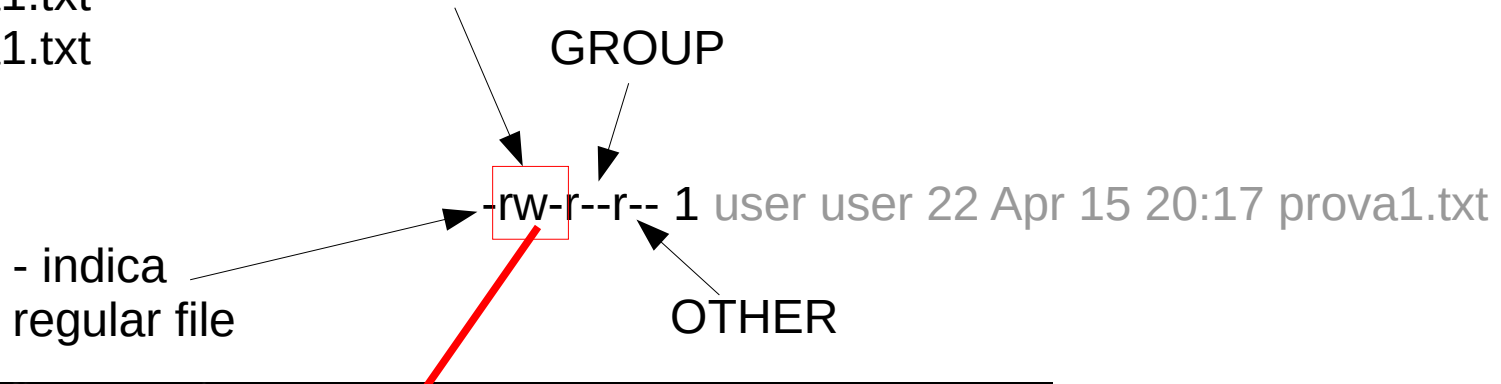
PERMESSI

gruppi di **3** bit , USER, GROUP, OTHER
USER: rw- READ(on) WRITE (ON) EXECUTE (OFF)

touch prova1.txt
echo “test riga1” prova1.txt
echo “test riga2” prova1.txt
cat prova1.txt

test riga1
test riga2

ls -lah




```
user@:~$ ls -lah
total 20K
drwxr-xr-x 3 user user  160 Apr 15 20:16 .
drwxr-xr-x 3 root root   60 Apr 15 12:36 ..
-rw-r--r-- 1 user user  220 Apr 15 12:36 .bash_logout
-rw-r--r-- 1 user user 1003 Apr 15 12:36 .bashrc
drwx------ 3 user user   60 Apr 15 12:36 .kde
-rw-r--r-- 1 user user  675 Apr 15 12:36 .profile
-rw-r--r-- 1 user user   22 Apr 15 20:17 prova1.txt
-rw-r--r-- 1 user user   19 Apr 15 12:36 .su-to-rootrc
user@:~$
```

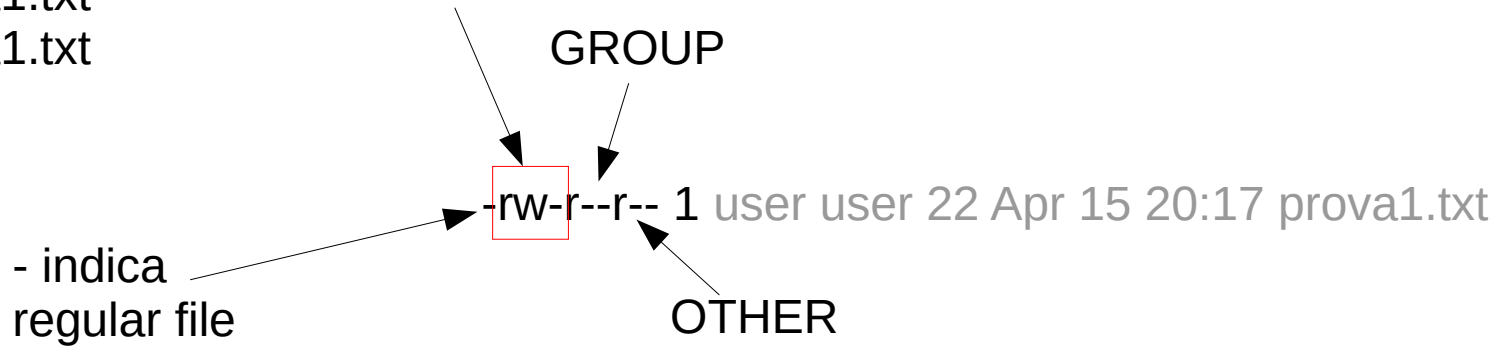
PERMESSI

touch prova1.txt
echo “test riga1” prova1.txt
echo “test riga2” prova1.txt
cat prova1.txt

test riga1
test riga2

ls -lah

gruppi di **3** bit , USER, GROUP, OTHER
USER: rw- READ(on) WRITE (ON) EXECUTE (OFF)



Vogliamo rendere il file scrivibile da tutti gli appartenenti al gruppo a cui appartiene il proprietario e non scrivibile e non leggibile da tutti gli utenti non appartenenti al gruppo dle proprietario.

-rw-rw---- le “triplette” di permessi sono quindi: 110 → 6 (USER)
110 → 6 (GROUP)
000 → 0 (OTHER)

Il comando da utilizzare è : chmod 660 prova1.txt (provate e verificate in risultato, ls -lah)

AGIRE SUI PERMESSI

- Cambiare il proprietario
  - chown utente[:gruppo] file
- Cambiare il gruppo
  - chgrp gruppo file
- Cambiare i permessi
  - chmod 755 file
  - chmod +x file
  - chmod a=rw file
  - chmod g−x file
- (per creare un utente: adduser)

Il bit SUID

Il proprietario di un processo in esecuzione è normalmente *diverso* dal proprietario del file contente un programma (e diverso ad ogni esecuzione)

- effective UID bit: il processo assume come proprietario il proprietario del file del programma
- SUID `root`
- chmod 4555 file
- chmod u+s file

http://linuxdidattica.org/docs/drd_mr/debianadv/node43.html

Il bit SUID

Ai normali permessi si aggiungono altre informazioni definibili globalmente come modalità dei permessi, nelle quali rientra la modalità detta ``suid''.

La modalità suid attribuisce al file in esecuzione i privilegi dell'utente cui appartiene; se però l'utente cui appartiene è l'amministratore di sistema, ecco che la modalità suid attribuisce al file in esecuzione i privilegi di root. Ciò sarà più chiaro con un semplice esempio.

Facciamo per prima cosa una copia dell'eseguibile cp nella nostra cartella personale:

```
$ cd

$ cp /bin/cp .          (non dimenticate il punto finale)
```

Assumiamo i privilegi di root:

```
$ su
```

e cambiamo I permessi del file cp:

```
# chmod u+s cp                  ( attribuiamo al file la modalità suid )
```

Il bit SUID

Infatti se ora facciamo elencare il file:

```
# ls -l cp

-rwsr-xr-x root root 51212 2014-04-12 17:48 cp

# exit        (rinunciamo ai privilegi di root)
```

```
$ cp prova1.txt provanotsuid.txt
$ ./cp prova1.txt provasuid.txt
$ ls -l

-rw-r-r- 1 user user ... provanotsuid.txt

-rw-r-r- 1 root user  ... provasuid.txt
```

Il file copiato provasuid.txt appartiene a root pur essendo stato creato da un utente normale. Le implicazioni sulla sicurezza del sistema derivanti dall'uso di file eseguibili di proprietà di root in modalità suid sono enormi e pericolose, perchè la funzionalità suid non consente un uso selettivo sulla base dell'utente che esegue il comando stesso: in altre parole, chiunque può usare il comando con i privilegi di root.