

# Interfaccia Utente a caratteri (testuale)

## Interprete dei comandi

## Shell scripting

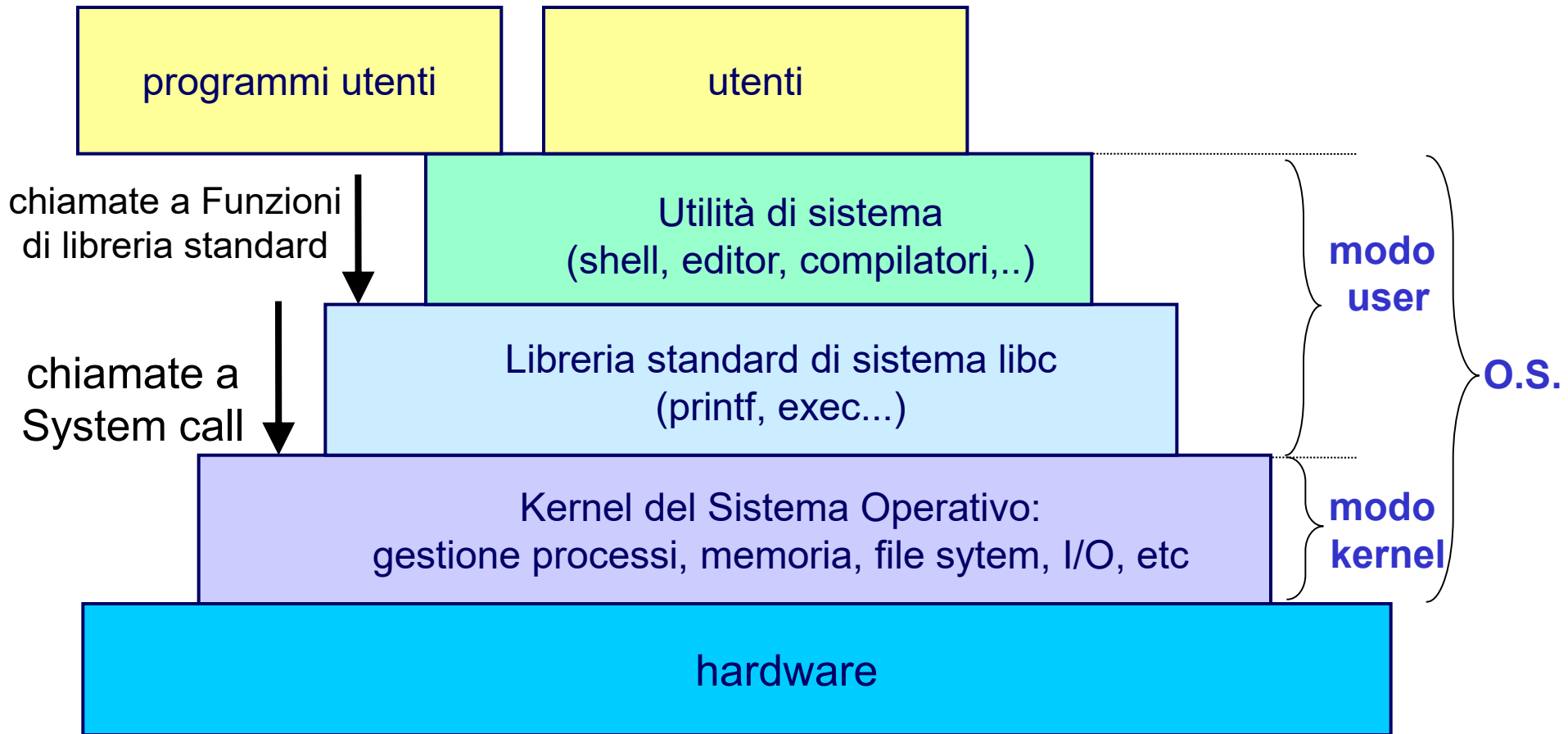
### NOTA BENE:

Questa guida contiene solo una introduzione minimale all'utilizzo dell'interprete di comandi bash, e serve per fornire agli studenti del corso di Sistemi Operativi le informazioni iniziali per utilizzare una semplice interfaccia a riga di comando con cui compilare programmi in linguaggio ANSI C mediante il compilatore gcc, eseguire e debuggare tali programmi, scrivere ed eseguire semplici script, capire le interazioni tra i programmi e l'utente e, in sostanza, identificare le funzionalita' messe a disposizione dell'utente dal sistema operativo.

Per tale motivo, alcuni concetti sono stati volutamente semplificati, allo scopo di rendere più semplice la comprensione, anche a discapito della realtà.

Vittorio Ghini

# Struttura del Sistema Operativo



# Requisito Hardware per il Sistema Operativo

## Modi di esecuzione della CPU

Instruction Set Architecture of IA-32  
Privilege Levels (Rings)

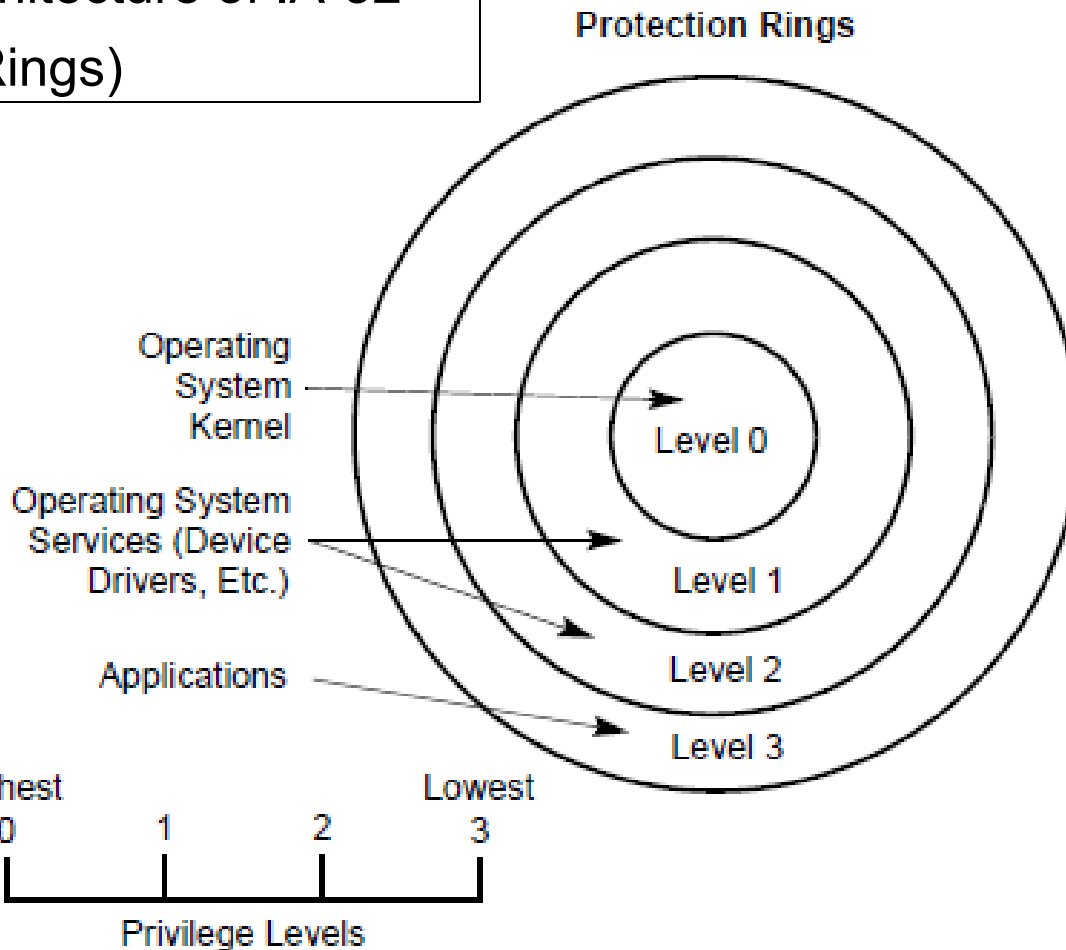


Figure 6-3. Protection Rings

# chiamata a system call mediante interrupt

```
/* file print.s Descr.: stampa a video di stringa Architettura: x86 Dialecto assembly AT&T (-masm=att)
Assemblaggio e Linking Modo 0, non usare gcc:
Assemblare con: as -o print.o --gstabs print.s (eventuale opzione --gstabs per poter usare gdb o ddd)
Linkare con: ld -o print.exe print.o
Eseguire con ./print.exe o (per il debug) con gdb print.exe o ddd print.exe */
```

## **.data**

```
miastringa: .string "MANNAG\n" # stringa da stampare compresa di a capo
```

## **.text**

```
.globl _start
```

### **\_start:**

```
    nop
```

/\* qui attivo due volte, mediante l'istruzione **int**, l'interruzione software identificata dal valore **0x80=8016** con la quale, in generale, si richiamano routine o servizi del sistema operativo (nel nostro caso Linux).

Il servizio richiamato e' identificato dal valore inserito nel registro eax prima di attivare la interruzione.

- Il valore 4 chiede il servizio di stampa di una stringa.

- Il valore 1 invece corrisponde alla routine di terminazione del programma e ritorno al sistema operativo.

```
*/
```

```
/* stampo stringa e vado a capo */
```

```
    movl $4, %eax                /* servizio da ottenere: stampa */
```

```
    movl $1, %ebx                /* sottoservizio da ottenere */
```

```
    movl $miastringa, %ecx       /* indirizzo di inizio stringa in registro ecx */
```

```
    movl $7, %edx                /* lunghezza della stringa da stampare in registro edx */
```

```
    int $0x80                    /* ordine di eseguire interrupt */
```

```
/* termino */
```

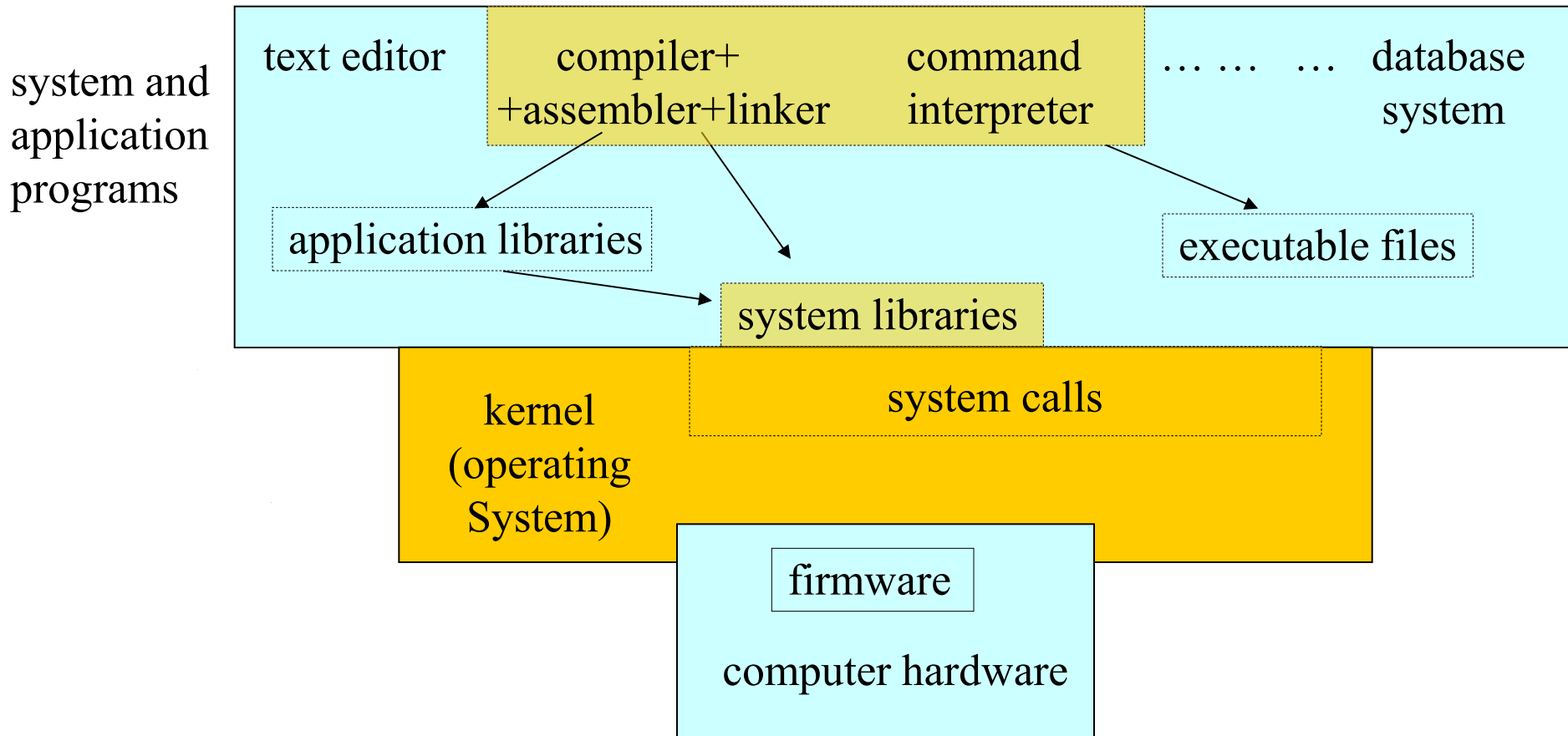
```
fine:
```

```
    movl $1, %eax                /* servizio da ottenere: terminazione processo */
```

```
    int $0x80                    /* ordine di eseguire interrupt */
```

# Librerie e Chiamate di sistema (1)

Ricordiamo l'organizzazione del computer, in particolare la relazione tra le system calls e le librerie di sistema.



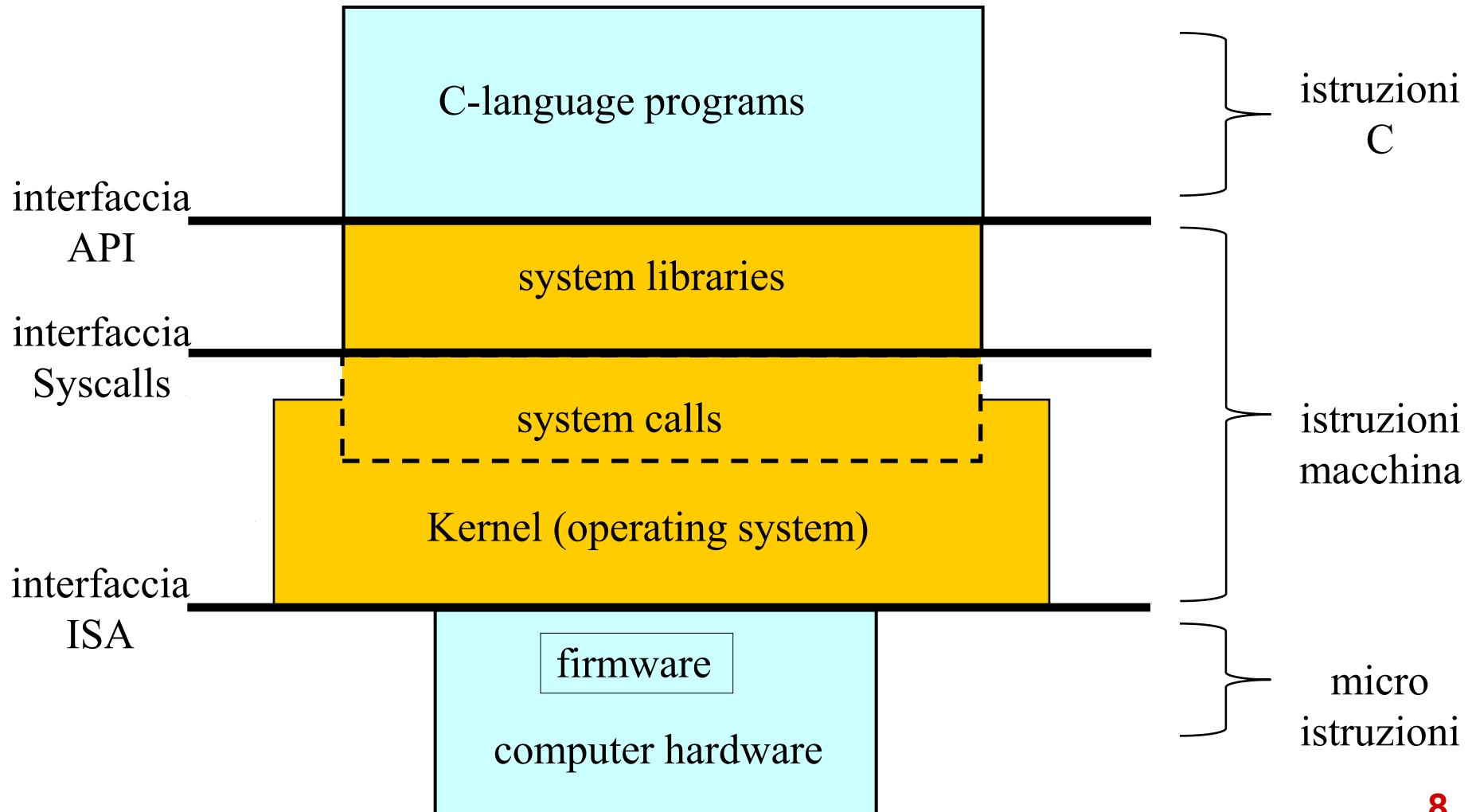
# Librerie e Chiamate di sistema (2)

- ✿ Le chiamate di sistema forniscono ai processi i servizi offerti dal SO:  
Controllo dei processi, Gestione dei file e dei permessi, Gestione dei dispositivi di I/O, Comunicazioni.
- ✿ Il modo in cui sono realizzate cambia al variare della CPU e del sistema operativo.
- ✿ Il modo di utilizzarle (chiamarle) cambia al variare della CPU e del sistema operativo.
- ✿ Sono utilizzate (invoke) direttamente utilizzando linguaggi di basso livello (assembly).
- ✿ Possono essere chiamate (invoke) indirettamente utilizzando linguaggi di alto livello (C o C++)
- ✿ Infatti, normalmente i programmi applicativi non invocano direttamente le system call, bensì invocano funzioni contenute in librerie messe a disposizione dal sistema operativo ed utilizzate dai compilatori per generare gli 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.
- ✿ Le librerie messe a disposizione dal sistema operativo sono dette **librerie di sistema**.

# Librerie e Chiamate di sistema (3)

- ✿ Per ciascun linguaggio importante, il s.o. fornisce API di sistema da usare.
  - ✿ **Cambiando il sistema operativo oppure cambiando il processore (CPU) su cui il sistema operativo si appoggia, i nomi delle funzioni delle API rimangono gli stessi ma cambia il modo di invocare le system call, quindi cambia l'implementazione delle funzioni delle API per adattarle alle system call e all'hardware.**
  - ✿ Perché è importante programmare usando linguaggi di alto livello (es C)  
**Un programma sorgente in linguaggio C che usa le API non deve essere modificato se cambio il sistema operativo o l'hardware (e questi mantengono le stesse API), poiché sul nuovo s.o. cambia l'implementazione delle API e quindi cambia l'eseguibile che viene generato su diversi s.o ed hardware.**
- Alcune API molto 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 (JVM).
- ✿ Possono esistere anche librerie messe a disposizione non dal sistema operativo bensì realizzate, ad esempio, da un utente.
- ✿ Solitamente tali librerie applicative sono implementate utilizzando le librerie di sistema.

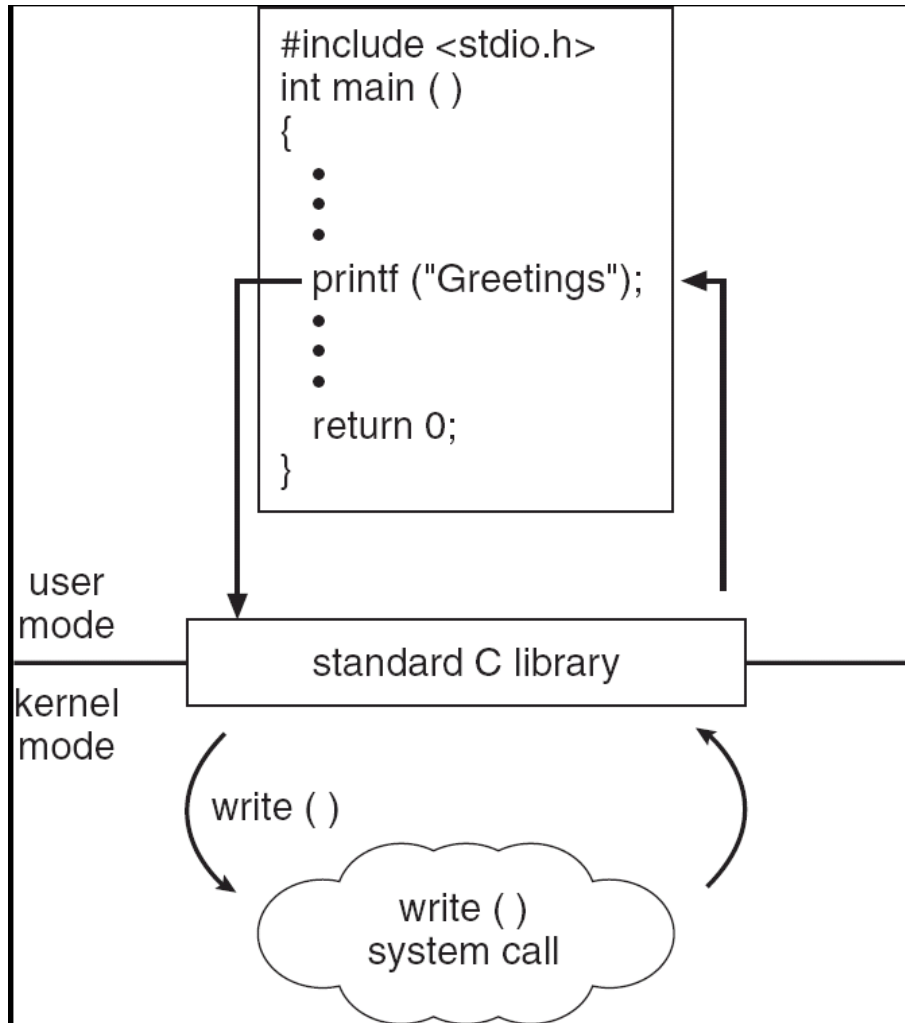
# Linguaggi utilizzati e interfacce di servizio





# Esempio con la libreria standard C

- ❁ Per Linux, la libreria standard del linguaggio C (il *run-time support system*) fornisce una parte dell'API



- ✗ Programma C che invoca la funzione di libreria per la stampa *printf()*
- ✗ La libreria C implementa la funzione e invoca la system call *write()* **nel modo richiesto dallo specifico s.o. e dalla specifica CPU.**
- ✗ La libreria riceve il valore restituito dalla chiamata al sistema e lo passa al programma utente

# Quali system call utilizza un eseguibile?

## strace

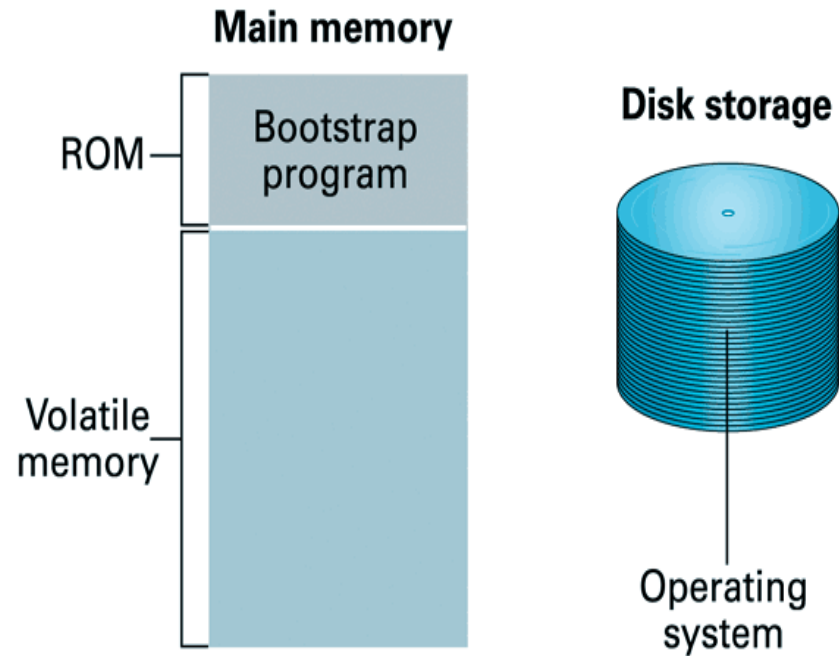
- ✱ In Linux esiste un comando che permette di vedere quali system call sono usate da un programma in esecuzione.
- ✱ **strace** esegue il comando specificato come argomento fino a che questo termina. Intercetta e visualizza le system calls che sono chiamate dal processo e i segnali che sono ricevuti dal processo.

Facciamo un esempio: lanciando il comando `ifconfig` visualizzo lo stato delle interfacce di rete. Vediamo quali syscall usa `ifconfig` eseguendo il comando:

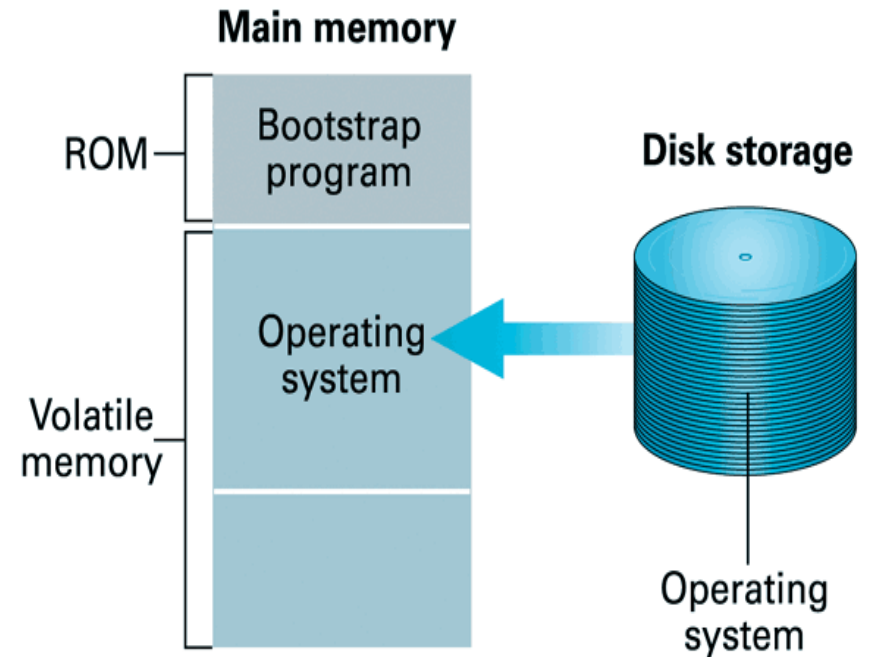
**strace** ifconfig

```
execve("/sbin/ifconfig", ["ifconfig"], [/* 52 vars */]) = 0
brk(NULL)                               = 0x12f3000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f37cb8ee000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=71665, ...}) = 0
mmap(NULL, 71665, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f37cb8dc000
close(3)
```

# L'avvio del sistema operativo - Bootstrap



**Step 1:** Machine starts by executing the bootstrap program already in memory. Operating system is stored in mass storage.



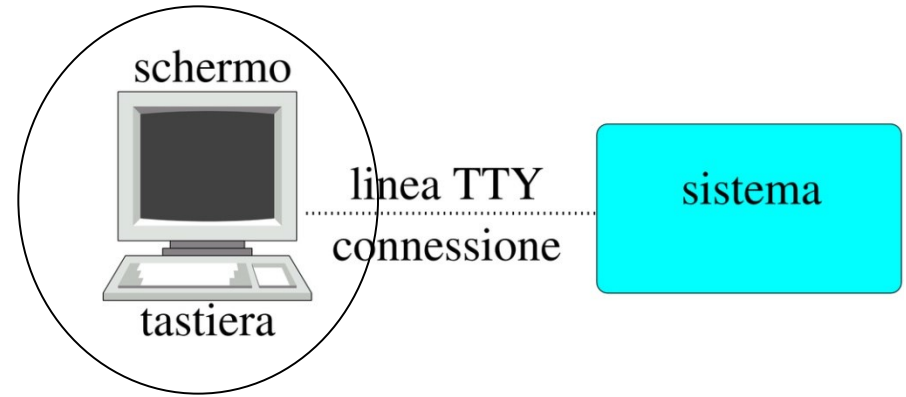
**Step 2:** Bootstrap program directs the transfer of the operating system into main memory and then transfers control to it.

# Servizi del Sistema Operativo

- ✱ **Gestione di Risorse:** allocazione, contabilizzazione, protezione e sicurezza (possessori di informazione devono essere garantiti da accessi indesiderati ai propri dati; processi concorrenti non devono interferire fra loro).
- ✱ Comunicazioni (intra e inter-computer)
- ✱ Rilevamento di errori che possono verificarsi nella CPU e nella memoria, nei dispositivi di I/O o durante l'esecuzione di programmi utente
- ✱ Gestione del file system — capacità dei programmi e dell'utente di leggere, scrivere e cancellare file e muoversi nella struttura delle directory
- ✱ Operazioni di I/O — il SO fornisce ai programmi utente i mezzi per effettuare l'I/O su file o periferica
- ✱ **Esecuzione di programmi** — capacità di caricare un programma in memoria ed eseguirlo, eventualmente rilevando e gestendo, situazioni di errore
- ✱ **Programmi di sistema**
- ✱ **Chiamate di sistema**
- ✱ **Interfaccia utente: a linea di comando** (*Command Line Interface*, CLI) o grafica (*Graphic User Interface*, GUI).

# Console = terminale = schermo + tastiera

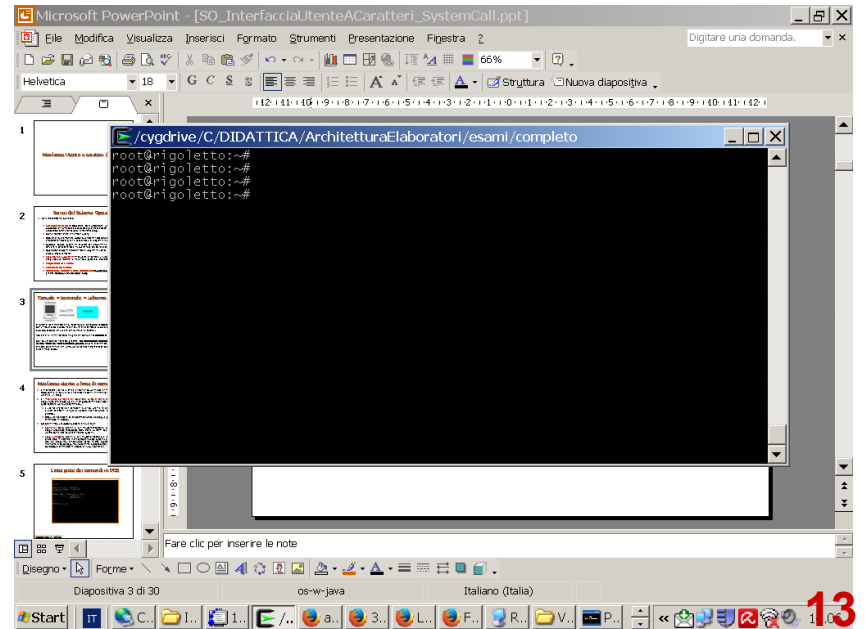
In origine i terminali (schermo+tastiera) erano dispositivi **separati** dal computer vero e proprio (detto mainframe) e comunicavano con questo mediante una linea seriale. L'output verso l'utente era di tipo solo testuale, l'input era fornito mediante tastiera.



Attualmente i terminali sono integrati nei computer (**schermo e tastiera del PC**).

Con l'avvento delle interfacce grafiche, **lo schermo del terminale viene emulato in una "finestra" dell'ambiente grafico.**

Si parla di terminale virtuale. Quando la finestra del terminale è in primo piano i caratteri digitati dalla tastiera vengono passati al terminale stesso.



# Interfaccia utente a linea di comando (CLI)

- ✱ L'interfaccia utente a linea di comando permette di impartire ordini al SO sotto forma di sequenze di caratteri alfa-numeriche digitati sulla tastiera (o presi da un file).
- ✱ L' **interprete dei comandi** (o anche **Shell di comandi**) è un programma eseguibile che si occupa di un singolo terminale. Attende i caratteri digitati sulla tastiera per quel terminale.
  - ✱ L'utente digita dei caratteri, quando preme il tasto di invio (return) questi caratteri vengono passati all'interprete dei comandi che li gestisce.
  - ✱ Ricevuti i caratteri, la shell li interpreta ed esegue gli ordini ricevuti, poi si rimette in attesa di nuovi ordini.
- ✱ La shell è programmabile perché al prompt è possibile passare uno script, ovvero una sequenza di ordini da eseguire.
- ✱ Gli ordini ricevuti possono essere di due tipi:
  - ✱ **Comandi** sono ordini la cui implementazione è contenuta nel file eseguibile della shell stessa (comandi built-in della shell). Non li troviamo perciò come file separati nel file system. Ad esempio il comando **cd**
  - ✱ **Eseguibili** sono ordini la cui implementazione è contenuta fuori della shell, cioè in altri file memorizzati nel file system. L'ordine è il nome del file da eseguire. L'interprete cerca il file specificato, lo carica in memoria e lo fa eseguire. Al termine dell'esecuzione la shell riprende il controllo e si rimette in attesa di nuovi comandi.

# L'interprete dei comandi in alcuni s.o.

- ✿ Unix, Linux:
  - ✿ Diverse Shell di comandi, tra le più utilizzate **bash** (bourne again shell).
- Windows:
  - Shell DOS
  - Emulazione di interfaccia testuale Linux: bash realizzata dalla piattaforma cygwin. Utile per far eseguire, in ambiente windows, applicazioni e servizi implementati per Linux.
  - In Windows 10 è stata aggiunta una bash (non completissima)
- Mac:
  - Diverse Shell di comandi, compresa **bash**

-----
- Nel corso di sistemi operativi utilizzeremo compilatori con sola interfaccia testuale, in particolare il GNU C Compiler (gcc).
- Gli ordini al compilatore, così come l'ordine di eseguire un programma, verranno impartiti utilizzando una shell bash.
  - In Linux e su Mac esiste già un terminale con shell bash.
  - In Windows è perciò necessario installare l'emulatore cygwin oppure installare una macchina virtuale Linux.

# Interfaccia utente GUI

- ✱ Interfaccia user-friendly che realizza la metafora della scrivania (**desktop**)
  - ✕ Interazione semplice via mouse
  - ✕ Le **icone** rappresentano file, directory, programmi, azioni, etc.
  - ✕ I diversi tasti del mouse, posizionato su oggetti differenti, provocano diversi tipi di azione (forniscono informazioni sull'oggetto in questione, eseguono funzioni tipiche dell'oggetto, aprono directory – *folder*, o *cartelle*, nel gergo GUI)



# Standard POSIX

## (Portable Operating System Interface)

E' una famiglia di standard sviluppato dall'IEEE

- **IEEE 1003.1 (POSIX.1)** Definizione delle system call (operating system interface) fruibili da linguaggio C
- IEEE 1003.2 (POSIX.2) Shell e utility
- IEEE 1003.7 (POSIX.7) System administration

Noi siamo interessati allo standard IEEE 1003.1 (POSIX.1)

- POSIX.1            1988        Original standard
- POSIX.1            1990        Revised text
- POSIX.1a           1993        Addendum
- POSIX.1b           1993        Real-time extensions
- POSIX.1c           1996        Thread extensions

**POSIX.1 include alcune primitive della C Standard Library**

**POSIX.1 include alcune primitive non appartenenti alla C Standard Library**

**Non tutte le primitive della C Standard Library appartengono a POSIX.1**

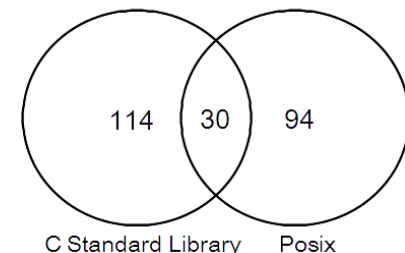
**Esempio:**

**open()**    e' POSIX.1

**fopen()**   e' Standard C e POSIX.1

**sin()**     e' Standard C

Numero di funzioni:



# Standard C conforme a POSIX

## Portabilità:

Un sorgente C Standard conforme a POSIX può essere eseguito, una volta ricompilato, su qualunque sistema POSIX dotato di un ambiente di programmazione C Standard.

**Per scrivere un programma conforme a POSIX, bisogna includere gli header files richiesti dalle varie primitive usate, scelti fra:**

- header files della C Standard Library (contenenti opportune modifiche POSIX)
- header files specifici POSIX

Per ottenere la lista completa degli header necessari ad utilizzare ciascuna primitiva, utilizzate **man nome\_primitiva**

## Inoltre:

Se vogliamo compilare un programma in modo tale che dipenda solo dallo standard POSIX nella sua versione originaria (IEEE Std 1003.1) , dobbiamo inserire prima degli **#include:**

```
#define _POSIX_SOURCE 1
```

```
/* significa IEEE Std 1003.1 */
```

# Standard C conforme a versioni specifiche di POSIX

## Portabilità:

Se vogliamo compilare un programma in modo tale che utilizzi solo funzioni aderenti allo standard POSIX, e in particolare ad una versione specifica dello standard POSIX, dobbiamo inserire un simbolo che identifica la particolare versione POSIX di nostro interesse.

Per la versione base di POSIX, prima degli **#include** dobbiamo inserire:

```
#define _POSIX_SOURCE 1 /* significa IEEE Std 1003.1 */
```

oppure, per aderire a specifiche versioni POSIX, dobbiamo inserire il simbolo

```
#define _POSIX_C_SOURCE VALORE
```

dove VALORE sarà uno dei seguenti

If ==1, like \_POSIX\_SOURCE; if >=2 add IEEE Std 1003.2;

if >=199309L, add IEEE Std 1003.1b-1993;

If >=199506L, add IEEE Std 1003.1c-1995;

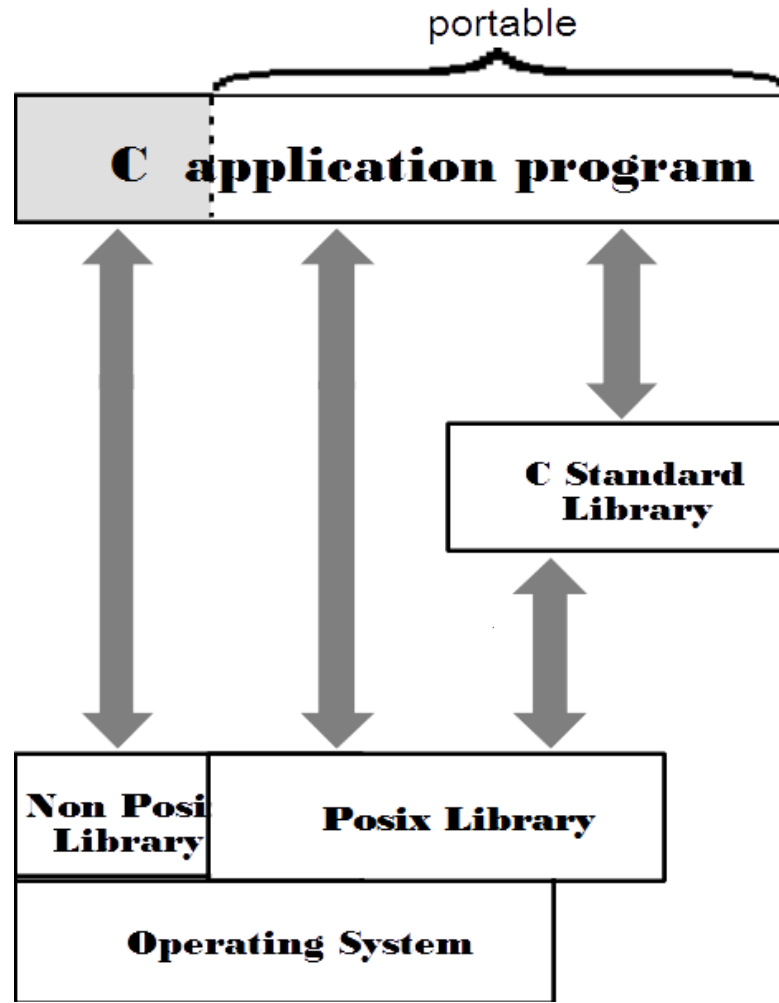
if >=200112L, all of IEEE 1003.1-2004

if >=200809L, all of IEEE 1003.1-2008

\_POSIX\_C\_SOURCE 1 equivale a \_POSIX\_SOURCE 1

( info tratte da /usr/include/features.h )

# Standard C conforme a POSIX



# Tipi Primitivi di POSIX

## Tipi di dato interi :

definiti negli header **stdint.h** e **inttypes.h** (quest'ultimo offre alcune macro)

**int8\_t**    **int16\_t**    **int32\_t**    **uint8\_t**    **uint16\_t**    **uint32\_t**

**intptr\_t**    **uintptr\_t** (questi 2 sono interi con dimensione uguale ai puntatori ad intero)

Lo header **sys/types.h** definisce i tipi dei **dati di sistema** primitivi (dipendenti dall'implementazione del sistema). Definiti per garantire portabilit . Sono definiti tramite typedef.

## Alcuni dei primitive system data types

<b>clock_t</b>	<b>counter of clock ticks</b>
<b>dev_t</b>	<b>device numbers</b>
<b>fd_set</b>	<b>file descriptor set</b>
<b>fpos_t</b>	<b>file position</b>
<b>gid_t</b>	<b>group id</b>
<b>ino_t</b>	<b>inode number</b>
<b>mode_t</b>	<b>file types, file creation mode</b>
<b>nlink_t</b>	<b>number of hard links</b>
<b>off_t</b>	<b>offset</b>
<b>pid_t</b>	<b>process id</b>
<b>size_t</b>	<b>dimensioni (unsigned)</b>
<b>ssize_t</b>	<b>count of bytes (signed) (read, write)</b>
<b>time_t</b>	<b>counter of seconds since the Epoch</b>
<b>uid_t</b>	<b>user id</b>

# Nozioni per l'uso del Terminale: File system (1)

Il filesystem è la 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

- dei contenitori di dati detti files
- dei contenitori di files, detti directories (folders o cartelle in ambienti grafici)..

In realtà ciascuna directory può a sua volta contenere dei files e anche delle altre directories formando una struttura gerarchica

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

Se un file si chiama pippo.c ed è contenuto in una directory che si chiama vittorio che a sua volta è contenuta in una directory che si chiama home che a sua volta è contenuta nella partizione chiamata C:, allora è possibile individuare univocamente il file pippo.c indicando il **percorso** mediante il quale, partendo dalla partizione C: si arriva al file pippo.c

**C:\home\vittorio\pippo.c** <- percorso per raggiungere pippo.c

Notare il carattere separatore \ (si chiama backslash) che indica dove inizia il nome di una directory o di un file.

# Nozioni per l'uso del Terminale: File system (2)

In **Linux/Mac** invece le partizioni sono viste come collegate tra loro.

Esiste una partizione principale il cui nome è / (slash).

Questa partizione, così come le altre partizioni, può contenere files e directories. Le directories possono contenere files e directories.

Se un file si chiama `primo.c` ed è contenuto in una directory che si chiama `vittorio` che a sua volta è contenuta in una directory che si chiama `home` che a sua volta è contenuta nella partizione principale, allora è possibile individuare univocamente il file `pippo.c` indicando il **percorso** mediante il quale, partendo dalla partizione / si arriva al file `primo.c`

**/home/vittorio/primo.c**      <- percorso per raggiungere `primo.c`

Notare il carattere separatore / (slash) che indica dove inizia il nome di una directory o di un file. Quando il separatore / è all'inizio del percorso invece indica l'inizio della partizione principale, inizio che viene detto **root**..

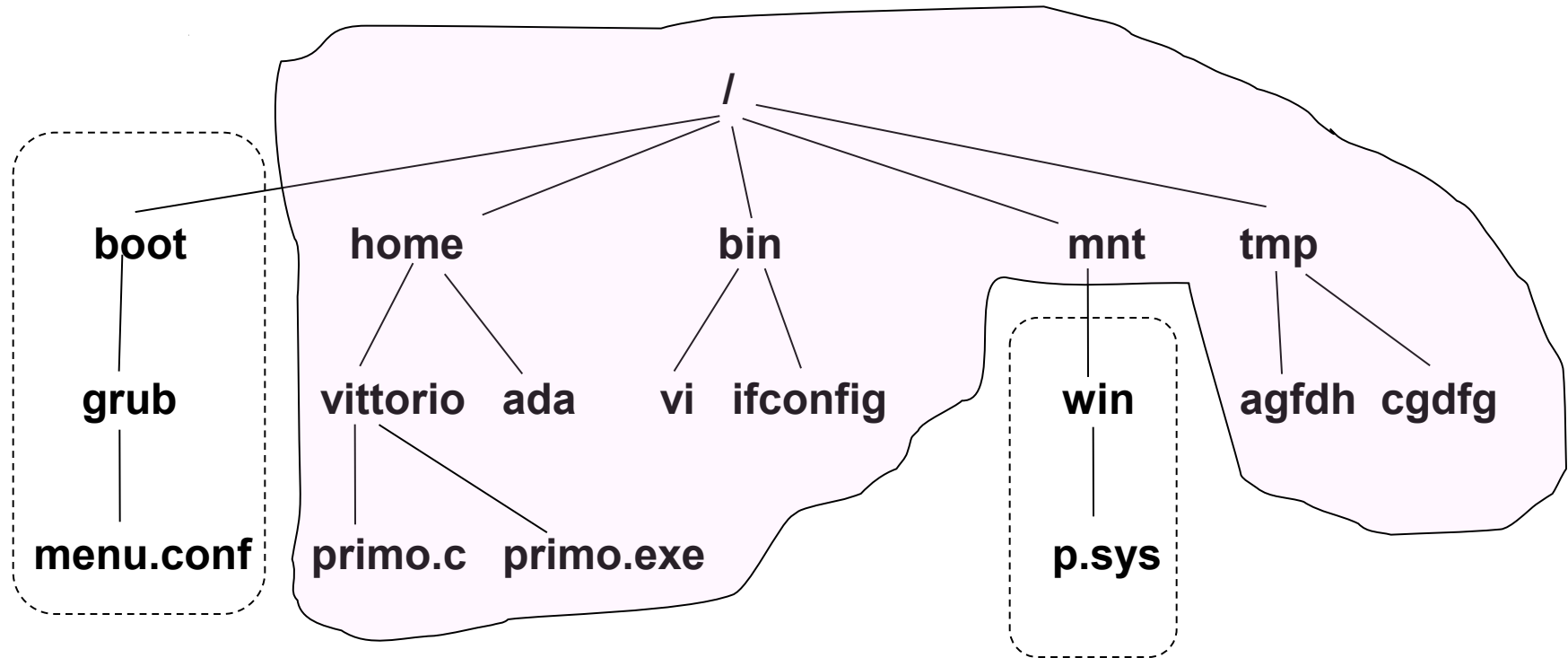
Le partizioni diverse da quella principale, si innestano (si collegano) logicamente in una qualche directory della partizione principale..

Ad esempio, una partizione chiamata `boot` può innestarsi direttamente nell'origine della partizione principale. Se questa partizione contiene una directory `grub` e quella directory contiene un file `menu.conf`, allora il percorso per identificare quel file `menu.conf` sarà:

**/boot/grub/menu.conf**      <- percorso per raggiungere il file.conf

Notare che non si capisce se un nome indica una directory o una partizione. **30**

# Nozioni per l'uso del Terminale: File system (3)



Esempio di strutturazione in directories e files delle partizioni di un filesystem Linux:

Nell'esempio, alla partizione principale `/` sono collegate (tecnicamente si dice **montate**) altre due partizioni (circondate dalle linee tratteggiate) di nome **boot** e **win**..

Notate che mentre la partizione `boot` è montata direttamente come fosse una directory contenuta nella directory di inizio (detta *root*) della partizione principale (`/`), la partizione `win` è montata in una sottodirectory della *root*.



# Nozioni per l'uso del Terminale: File system (4)

Nei sistemi **Linux/Mac** ciascun utente di un computer ha a disposizione dello spazio su disco per contenere i suoi files.

Per ciascun utente esiste, di solito, una directory in cui sono contenuti i files (intesi come directories e files) di quell'utente.

Quella directory viene detta **home dell'utente**.

Per esempio, nei pc dei laboratori del corso di laurea in Informatica di Bologna, la home dell'utente "rossi" si trova in **/home/students/rossi**

Per accedere ad un computer ciascun utente deve utilizzare farsi riconoscere mediante un nome utente (**account**) e autenticarsi mediante una **password**.

L'operazione iniziale con cui l'utente si autentica per accedere ad un pc si dice login.

L'autenticazione degli utenti permette che il sistema operativo protegga i files di un utente impedendo l'accesso da parte di altri utenti o di persone esterne.

Ciascun utente può vedere quali sono i permessi di accesso ai propri files ed eventualmente modificarli.

# Nozioni per l'uso del Terminale: File system (5)

Nel momento in cui si accede al terminale a riga di comando di un computer, il terminale **stabilisce la posizione logica attuale dell'utente all'interno del filesystem**, collocandolo inizialmente nella propria home directory.

Durante il lavoro l'utente può spostare la propria **posizione logica** in una diversa directory del filesystem.

NOTA BENE: spostarsi logicamente in una diversa directory VUOL DIRE VISITARE quella diversa directory e NON VUOL DIRE TRASFERIRE I PROPRI FILES IN QUELLA DIRECTORY

La directory in cui l'utente si trova logicamente in questo momento viene detta **directory corrente** e si dice che l'utente “**si trova nella**” directory corrente.

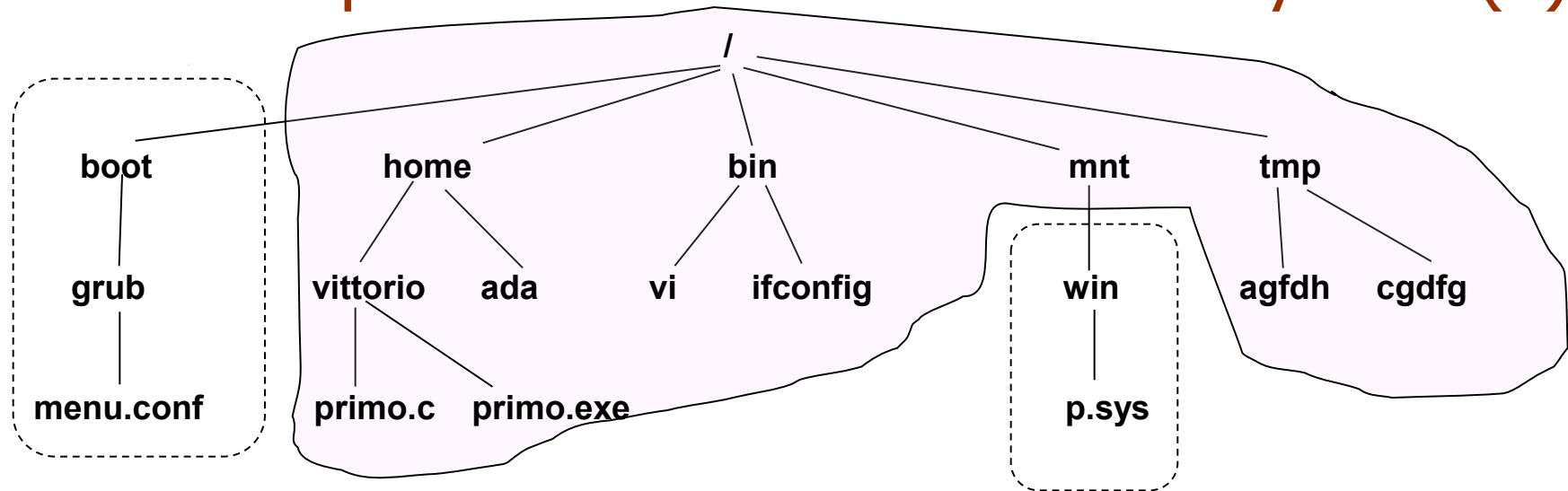
L'utente, per sapere in quale directory si trova logicamente in questo momento può eseguire il comando **pwd**, digitandolo da tastiera per farlo eseguire alla shell. Il comando **pwd** visualizza sullo schermo il percorso completo da / fino alla directory in cui l'utente si trova in quel momento.

```
# pwd  
/home/students/rossi/merda
```

Per spostarsi logicamente in una diversa directory, l'utente usa il comando **cd**

```
# cd /var/log
```

# Nozioni per l'uso del Terminale: File system (6)



Supponiamo di trovarci logicamente nella directory /home/vittorio

Per spostarmi logicamente nella directory /home posso usare cd in tre modi diversi

cd /home <- specifico percorso assoluto

cd ../ <- specifico percorso relativo cioè partendo dalla directory corrente

Si noti che il **simbolo ..** indica la directory superiore

Per spostarmi dalla directory /home/vittorio alla directory /mnt/win

cd /mnt/win <- specifico percorso assoluto

cd ../../mnt/win <- specifico percorso relativo

Per spostarmi dalla directory /boot alla directory /boot/grub

cd /boot/grub <- specifico il percorso assoluto

cd ./grub <- specifico percorso relativo (il **simbolo .** è la directory corrente)

cd grub <- specifico percorso relativo (più semplice)

# FHS – Filesystem Hierarchy Standard

Attualmente versione 3.0.

Specifiche disponibili presso Linux Foundation.

[http://refspecs.linuxfoundation.org/FHS\\_3.0/fhs-3.0.pdf](http://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf)

- Obiettivo: sapere dove siano posizionati files/directory e quale sia il loro utilizzo
- Vengono specificate un numero minimo di directory
- Per ciascuna Purpose, Requirements, Specific requirements

Directory richieste in /

bin	Essential command binaries
boot	Static files of the boot loader
dev	Device files
etc	Host-specific system configuration
lib	Essential shared libraries and kernel modules
media	Mount point for removeable media
mnt	Mount point for mounting a filesystem temporarily
opt	Add-on application software packages
sbin	Essential system binaries
srv	Data for services provided by this system
tmp	Temporary files
usr	Secondary hierarchy

# Nozioni per uso del Terminale: Metacaratteri

## Caratteri speciali

ne vedremo l'uso andando avanti

<b>&gt; &gt;&gt; &lt;</b>	<b>redirezione I/O</b>
<b> </b>	<b>pipe</b>
<b>* ? [...]</b>	<b>wildcards</b>
<b>`command`</b>	<b>command substitution</b>
<b>;</b>	<b>esecuzione sequenziale</b>
<b>   &amp;&amp;</b>	<b>esecuzione condizionale</b>
<b>(...)</b>	<b>raggruppamento comandi</b>
<b>&amp;</b>	<b>esecuzione in background</b>
<b>" " ' '</b>	<b>quoting</b>
<b>#</b>	<b>commento (tranne un caso speciale)</b>
<b>\$</b>	<b>espansione di variabile</b>
<b>\</b>	<b>carattere di escape *</b>
<b>&lt;&lt;</b>	<b>"here document"</b>
<b>&lt;&lt;&lt;</b>	<b>"here string"</b>

# Interpretazione dei comandi bash: Espansioni

La shell legge ciascuna riga di comando, e la interpreta eseguendo alcune operazioni. Alcune di queste servono a capire dove finisce un comando e dove inizia il successivo. Altre servono ad individuare le parole (word splitting) e a identificare le parole riservate del linguaggio e a identificare la presenza di costrutti linguistici for while if. Altre ancora sostituiscono (espandono) alcune parti della riga di comando con altre, interpretando caratteri speciali. Alcuni caratteri speciali però, disabilitano alcune espansioni. In generale, la shell comincia riconoscendo i caratteri speciali. Successivamente la shell opera alcune sostituzioni nella riga di comando, effettuando le cosiddette espansioni (expansion).

Le espansioni principali, elencate in ordine di effettuazione, sono:

- |   |   |
|---|---|
| • <b>history expansion</b>  | !123  |
| • <b>brace expansion</b>  | a{damn,czk,bubu}e   |
| • <b>tilde expansion</b>  | ~/nomedirectory   |
| • <b>parameter and variable expansion</b>                           | \$1 \$? \$! \${var}   |
| • <b>arithmetic expansion</b>                                       | \$(( ))   |
| • <b>command substitution</b> (effettuata da sinistra verso destra) | ` ` \$( )   |
| • <b>word splitting</b>   |   |
| • <b>pathname expansion</b>   | * ? [...]   |
| • <b>quote removal</b>  | rimuove unquoted \ ' " non generate dalle precedenti espansioni |

(In alcuni sistemi sono possibili process substitution).

Vedremo i dettagli di queste espansioni man mano andando avanti.

# Nozioni per uso del Terminale: comando echo

**anticipazione:** Il comando **echo** permette di visualizzare a video la sequenza dei caratteri scritti subito dopo la parola **echo** e **fino al primo carattere di andata a capolinea** (che è inserito digitando il tasto **<INVIO>** o **<RETURN>**).

Il comando

```
echo pippo pippa pippi
```

visualizza

```
pippo pippa pippi
```

Se ho bisogno di far stampare a video anche caratteri speciali come **punti e virgola**, **andate a capo (per visualizzare su più righe)**, e altri, devo inserire sia prima che dopo la stringa da stampare il separatore “ doppio apice (non sono due caratteri apici ma il doppio apice, quello sopra il tasto col simbolo 2).

Esempio: 

```
echo "pippo ; pippa pippi"
```

Esercizio: All'interno di una shell far stampare a video su due linee diverse la stringa

```
pappa
```

```
ppero
```

Impossibile se non si usano i doppi apici

Soluzione: digitare (notando i doppi apici)

```
echo "pappa<INVIO>
```

```
ppero" <INVIO>
```

# Nozioni per uso del Terminale:

## Variabili e Variable expansion (1)

La shell dei comandi permette di usare delle **Variabili** che sono dei simboli dotati di un **nome** (che identifica la variabile) e di un **valore** (**che può essere cambiato**).

Il nome è una sequenza di caratteri anche numerici, ad es: PATH, PS1, USER PPID.

Attenzione, maiuscole e minuscole sono considerate diverse. Ad es **PaTH !=PATH**

Il valore è anch'esso una sequenza di caratteri compresi caratteri numerici e simboli di punteggiatura, ad es: /usr:/usr/bin:/usr/sbin 17 i686-pc-cygwin

Si noti che anche se il valore è composto solo da cifre (caratteri numerici), si tratta sempre di una stringa di caratteri.

Le variabili di una shell possono essere stabilite e modificate sia dal sistema operativo sia dall'utente che usa quella shell.

Alcune variabili (dette d'ambiente) vengono impostate subito dal sistema operativo non appena viene iniziata l'esecuzione della shell (ad es la variabile PATH).

Altre variabili possono essere create ex-novo dall'utente in un qualunque momento dell'esecuzione della shell.

Le variabili possono essere usate quando si digitano degli ordini per la shell. La shell **riconosce i nomi delle variabili contenuti negli ordini digitati, e cambia il contenuto dell'ordine sostituendo al nome della variabile il valore della variabile.**

Affinchè la shell **distingua il nome di una variabile**, questa deve essere **preceduta** dalla coppia di caratteri **\${** e seguita dal carattere **}** Se, all'interno dell'ordine digitato, il nome della variabile è seguito da spazi (caratteri "bianchi") non c'è pericolo di confondere la variabile con il resto dell'ordine e si possono omettere le parentesi grane.



# Nozioni per uso del Terminale: Variabili (2)

## assegnamenti e **variable** (e parameter) **expansion**

Ricordando che Il comando echo permette di visualizzare a video la sequenza dei caratteri scritti subito dopo la parola echo e fino al primo carattere di andata a capolinea (che è inserito digitando il tasto <INVIO> o <RETURN>).

NUM=MERDA      definisco una variabile di nome NUM e valore MERDA

echo \${NUM}      stampo a video la variabile NUM, si vedrà MERDA

echo \${NUM}X      stampo a video la variabile NUM, seguita dal carattere X  
si vedrà MERDAX

echo \$NUM      stampo a video la variabile NUM si vedrà MERDA

echo \$NUMX      vorrei stampare a video la variabile NUM, ma non metto le parentesi  
graffe, così la shell non capisce dove finisce il nome della variabile e  
non sostituisce il valore al nome. Non viene visualizzato nulla

echo \$NUM X      come prima, ma ora c'è uno spazio tra NUM e il carattere V  
così la shell capisce che il nome della variabile finisce dove  
comincia lo spazio e sostituisce il valore al nome.  
Viene visualizzato   MERDA X

# Nozioni per uso del Terminale: Variabili (2bis)

## Nota Bene: Assegnazione di valore ad una variabile

Quando si assegna un valore ad una variabile, **NON SONO CONSENTITI SPAZI** NE' PRIMA NE' DOPO IL SIMBOLO **=**

**Assegnamento corretto senza spazi** prima o dopo il **=**  
VARIABLE=contenuto

**Assegnamento sbagliato a causa di spazio PRIMA del simbolo =**  
VARIABLE        =contenuto

In questo caso la shell cerca di eseguire il comando avente nome VARIABLE passandogli come argomento la stringa =contenuto  
Avrei lo stesso errore se lasciassi uno spazio ANCHE dopo l'uguale  
VARIABLE        =    contenuto

**Assegnamento sbagliato a causa di spazio DOPO il simbolo =**  
VARIABLE=        contenuto

In questo caso la shell cerca di eseguire il comando avente nome **contenuto** costruendo per tale comando un nuovo ambiente di esecuzione in cui colloca una variabile vuota di nome VARIABLE (vedere slide piu' avanti).

# Nozioni per uso del Terminale: Variabili (3)

Esiste una **variabile d'ambiente** particolare e importantissima, detta **PATH**. Viene impostata dal sistema operativo già all'inizio dell'esecuzione della shell. L'utente può cambiare il valore di questa variabile.

**La variabile PATH contiene una sequenza di percorsi assoluti nel filesystem di alcune directory in cui sono contenuti gli eseguibili.** I diversi percorsi sono separati dal carattere **:**

Esempio di valore di PATH     /bin:./sbin:./usr/bin:./usr/local/bin:./home/vittorio

Questa PATH definisce i percorsi che portano alle directory seguenti

- /bin
- /sbin
- /usr/bin
- /usr/local/bin
- /home/vittorio

Quando io ordino alla shell di eseguire un certo file binario, chiamandolo per nome, ma senza specificare il percorso completo (assoluto o relativo) per raggiungere quel file, allora **la shell cerca quel file binario all'interno delle directory specificate dai percorsi che formano la variabile PATH, nell'ordine con cui i percorsi sono contenuti nella variabile PATH**, cioè nell'esempio prima in /bin poi in /sbin etc.etc.

- Quando la shell trova il file eseguibile lo esegue.
- Se il file eseguibile non viene trovato nelle directory specificate, la shell visualizza un errore e non esegue il file.

# Usare il Terminale a linea di comando (bash)

Per **aprire il terminale a riga di comando** Linux-like in ambiente grafico:

In Windows cercare e cliccare l'icona di "cygwin bash shell" o "cygwin terminal".

In Linux e Mac cliccare sul menù "terminal" o "terminal emulator" o "console".

Si **aprirà una "finestra"** grafica e comparirà una piccola segnalazione **lampeggiante (cursore)** che indica che la shell è pronta ad accettare dei caratteri da tastiera.

Ad ogni istante, la shell opera stando in una posizione (directory) del filesystem denominata **directory corrente**. All'inizio dell'esecuzione della shell, la directory corrente è la home directory dell'utente che esegue la shell stessa.

L'utente può cambiare la directory corrente utilizzando il comando **cd**.

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

- **comandi** (piu' precisamente **comandi built-in**). Sono implementati e inclusi nella shell stessa e quindi non esistono come file separati. Sono forniti dal sistema operativo. Ad esempio cd, if, for.
- **file binari eseguibili**. Sono file che contengono codice macchina e che si trovano nel filesystem. Possono essere forniti dal sistema operativo o dagli utenti. Ad esempio, ls, vi, tar, gcc, primo.exe.
- **script**. Sono file di testo che contengono una sequenza di nomi di comandi, binari e altri script che verranno eseguiti uno dopo l'altro. Possono essere forniti dal sistema operativo o dagli utenti. Ad esempio esempio\_script.sh

Per essere eseguito, un file binario o uno script deve avere i **permessi di esecuzione**.

Un utente può impostare il permesso di esecuzione di un proprio file usando il comando

chmod come segue: **chmod u+x esempio\_script.sh**

# Utenti e Gruppi

Nei sistemi Unix/Linux esistono le astrazioni di utente (**user**) e gruppo di utenti (**group**).

- Un utente può corrispondere ad una persona umana oppure essere solo la rappresentazione di una entità usata per indicare chi è che esegue un servizio di sistema. Ad esempio lo user `mysql` che esegue il servizio del database `mysql`.
- Un utente (**user**) è caratterizzato da una stringa chiamata **username** che contiene il nome utente (`studente`, `vic`, `syslog`) e da un identificatore numerico chiamato **userID** entrambi univoci nel sistema.
- Un gruppo (**group**) è caratterizzato da una stringa chiamata **groupname** che contiene il nome del gruppo (`staff`, `admin`, ...) e da un identificatore numerico chiamato **groupID** entrambi univoci nel sistema. Ciascun utente appartiene ad uno o più gruppi.
- Un utente appartiene ad un gruppo primario, ma può far parte di diversi gruppi secondari (o supplementari).
- Differenza tra Gruppo Primario e Secondario:
  - **Gruppo Primario:** È il gruppo principale dell'utente. **Quando un utente crea un nuovo file (o directory), al file viene associato proprio il gruppo primario dell'utente.** Quindi il gruppo primario determina il gruppo dei file creati dall'utente (almeno all'inizio). In un momento successivo il gruppo del file può essere cambiato.
  - **Gruppo Secondario:** Sono gruppi aggiuntivi a cui un utente può appartenere. L'appartenenza a questi gruppi garantisce all'utente i permessi e i privilegi definiti per tali gruppi, consentendo ad esempio di accedere a file e risorse condivise.

# Utente Proprietario e Gruppi dei file

- **Ciascun file (e ciascuna directory) del filesystem appartiene ad un utente** (detto proprietario del file, owner) che normalmente è l'utente che ha creato quel file.
- **Ciascun file (e ciascuna directory) è associato ad un solo gruppo.**
- Inizialmente, il gruppo di un file è il gruppo primario dell'utente creatore del file (il proprietario del file).
- In un momento successivo, l'utente proprietario può cambiare il gruppo di un file (comando chgrp)
- L'utente proprietario di un file può anche cambiare il proprietario di un proprio file (comando chown). Lo stesso comando chown consente al proprietario di cambiare contemporaneamente il proprietario e il gruppo di un proprio file.
- **Un utente può tentare di accedere ad un file**, chiedendo di leggere o modificare il contenuto di un file oppure di eseguire un file eseguibile, **anche se non è il proprietario del file.**
- Viene indicato con **effective user** un utente quando cerca di accedere ad un file: il termine permette di distinguere tra chi vorrebbe usare il file e chi ne è il proprietario.
- **Il proprietario di un file stabilisce chi può accedere a quel suo file**, configurando i permessi di accesso a quel file. **Il proprietario stabilisce**, distinguendoli, **i permessi assegnati al proprietario del file** (sé stesso), agli utenti appartenenti allo stesso gruppo del file e, infine, **a tutti gli altri.**

# Permessi di file e directory

Ogni file ha un **proprietario** ed un **gruppo**.

Quando un utente crea un file, il s.o. assegna l'utente come proprietario del file.

Il proprietario/creatore poi può cambiare il proprietario del file con il comando:

***chown*** *nuovoproprietario nomefile*

Per ciascun file, il s.o. mantiene diversi diritti di lettura, scrittura ed esecuzione assegnati al proprietario, al gruppo e a tutti gli altri.

Lettura (valore **4**, simbolo **r**)

File: lettura

Directory: elenco file/directory nella cartella (non le proprietà di file e directory)

Scrittura (valore **2**, simbolo **w**)

File: modifica

Directory: creazione, eliminazione, cambio nome file

Esecuzione (valore **1**, simbolo **x**)

File: esecuzione

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

user			<u>group</u>			<u>others</u>		
R	W	X	R	W	X	R	W	X
4	2	1	4	2	1	4	2	1

Comandi per cambiare proprietario, gruppo e permessi: ***chown***, ***chgrp***, ***chmod***

Solo il proprietario del file può cambiare proprietario, gruppo e permessi del file.

# Permessi di file e directory

Se un utente (detto user effettivo) vuole accedere ad un file, si applicano i seguenti criteri di utilizzo basati sui permessi di quel file

- ♦ Se l'utente (effective user) che vuole accedere ad un file è il proprietario del file, si applicano le User permission.
- ♦ Altrimenti, se il group ID del file corrisponde ad uno dei gruppi a cui appartiene l'effective user, allora si applicano le Group permission.
- ♦ Altrimenti, si applicano le Other permission

user			<u>group</u>			<u>others</u>		
R	W	X	R	W	X	R	W	X
4	2	1	4	2	1	4	2	1

Solo il proprietario del file può cambiare proprietario, gruppo e permessi del proprio file.  
Comandi per cambiare proprietario, gruppo e permessi: *chown, chgrp, chmod*

Esempio di assegnazione contemporanea di permessi **mediante formato numerico**:

assegnazione contemporanea di permessi

per owner (lettura, scrittura e esecuzione: 7),

per group (lettura e scrittura: 6)

e per other (sola lettura: 4)

**chmod 764 ./miofile.txt**



# Visualizzazione Permessi di file e directory

Lanciamo il comando **`ls -al`** nella nostra home directory per vedere tutte le informazioni (opzione -l) e quindi anche i permessi, di tutti i file (opzione -a).

```
ls -alh /home/vic/
```

Otteniamo come output:

```
drwxr-xr-x      34 vic  vic   4096 set 25 10:55 .
drwxr-xr-x   3   root root   4096 dic  9 2015 ..
-rw-r--r--    1   vic  vic   3826 giu 16 16:21 .bashrc
-rw-rw-r--    1   vic  vic    158 dic 17 2015 main.c
-rwxrwxr-x    1   vic  vic   8608 dic 17 2015 main.exe
```

Guardiamo i permessi dell'eseguibile che permette di cambiare la propria password.

```
ls -alh /usr/bin/passwd
```

Otteniamo come output:

```
-rwsr-xr-x    1   root root   51K  lug 22 2015 /usr/bin/passwd
```

Notare la **s** **nella parte di permessi utente**, che sostituisce la x di esecuzione.

Dice che quando quell'eseguibile viene eseguito da qualcuno che puo' eseguirlo, il processo creato dall'eseguibile esegue con i permessi di chi lo ha lanciato, **ma anche con i permessi del proprietario dell'eseguibile** (root, nel nostro esempio). Serve per effettuare operazioni con i permessi dell'amministratore di sistema.

## Special Permissions

<u>setuid</u>	<u>setgid</u>	<u>sticky</u>
4	2	1

# Permessi speciali di file e directory

**setuid** - rappresentato da "s" (o da "S") nelle user permissions (settato s con chmod 4\*\*\* )

File: in esecuzione, il processo associato all'esecuzione del file ottiene anche i diritti dell'owner (l'effective uid diventa quello dell'owner del file).

Tipicamente root. Esempio del comando /usr/bin/passwd

Directory: ignorato

esempio di settaggio setuid per proprietario: `chmod u+s ./miofile`

altro esempio di settaggio numerico di setuid e altri permessi: il 4 all'inizio e' setuid

```
vic@vic:~$ chmod 4761 ./main.exe
```

```
vic@vic:~$ ls -alh main.exe
```

```
-rwsrw---x 1 vic vic 8,5K dic 17 2015 main.exe
```

**setgid** - rappresentato da "s" (o da "S") nelle group permissions (settato con chmod 2\*\*\*)

File: analogo al setuid ma per il gruppo (è l'effective gid che diventa quello del file)

Directory: implica che i nuovi file e subdirectory create all'interno della directory ereditino il gid della directory stessa (e non quello del gruppo principale dell'utente che lo ha creato). Esempio di una directory condivisa

**sticky bit** - rappresentato da "t" (o da "T") (settato con chmod 1\*\*\*)

File: ora ignorato

Directory: i file all'interno di una directory con sticky bit possono essere rinominati o cancellati solo dal proprietario del file, dal proprietario della directory...o da root, ovviamente!

# Esempi di cambio permessi speciali (setuid)

```
vic@vic:~$ ls -alh main.exe  
-rwxrwxr-x 1 vic vic 8,5K dic 17 2015 main.exe
```

```
vic@vic:~$ chmod u+s ./main.exe
```

```
vic@vic:~$ ls -alh main.exe  
-rwsrwxr-x 1 vic vic 8,5K dic 17 2015 main.exe
```

```
vic@vic:~$ chmod u-s ./main.exe
```

```
vic@vic:~$ ls -alh main.exe  
-rwxrwxr-x 1 vic vic 8,5K dic 17 2015 main.exe
```

NOTARE LE DIFFERENZE TRA s ed S

- s           setuid e permesso di esecuzione valore 4
- S           setuid SENZA permesso di esecuzione. Controsenso.  
INCONSISTENZA. Non posso eseguire il file.

```
vic@vic:~$ chmod 4666 ./main.exe                   NB: NON ho dato permessi esecuzione a owner  
vic@vic:~$ ls -alh main.exe  
-rwSrwx-rw- 1 vic vic 8,5K dic 17 2015 main.exe
```

NOTARE QUI SOTTO CHE NON POSSO ASSEGNARE S

La visualizzazione di S, è solo una conseguenza dell'inconsistenza tra x ed s

```
vic@vic:~$ chmod u+S ./main.exe                   PROVOCA ERRORE
```

```
chmod: invalid mode: 'u+S'
```

```
Try 'chmod --help' for more information.
```

ANALOGA DIFFERENZA SUSSISTE TRA "t" e "T" per quanto riguarda lo sticky bit

# Come eseguire da Terminale a linea di comando

## Come specificare il nome del comando o dell'eseguibile?

I **comandi** possono essere eseguiti semplicemente invocandone il nome. (es: `cd ../` )

I **file eseguibili (binari o script)** devono invece essere invocati specificandone

- o il **percorso assoluto** (a partire dalla root)  
ad es per eseguire primo.exe digito `/home/vittorio/primo.exe`
- oppure il **percorso relativo** (a partire dalla directory corrente)  
se la directory corrente è `/home/ada` allora digito `../vittorio/primo.exe`
- oppure il **solo nome, a condizione che quel file sia contenuto in una directory specificata nella variabile d'ambiente PATH**

Ad esempio, se la variabile PATH è `/bin:/usr/bin:/home/vittorio:/usr/local/bin` allora qualunque sia la directory corrente, per eseguire il file `/home/vittorio/primo.exe` basta digitare il solo nome del file `primo.exe`

Esercizio:

Se la directory corrente è `/home/vittorio` ma quella directory non si trova nella variabile PATH, allora come posso eseguire il file `primo.exe`, che si trova in quella directory, specificando il percorso relativo?

`./primo.exe`

# Subshell

Una subshell e' una shell (shell figlia) creata da un'altra shell (shell padre).

**IMPORTANTE:** Una subshell viene creata in caso di:

- Esecuzione di comandi **raggruppati** (vedi dopo).
- Esecuzione di **script**.
- Esecuzione di processo in **background** (vedi dopo).
- **NB:** l'esecuzione di un comando built-in avviene nella stessa shell padre

Ogni shell ha una propria directory corrente (ereditata dalla shell padre).

Ogni shell ha delle proprie variabili.

Ogni subshell eredita dalla shell padre una copia delle variabili d'ambiente (vedi dopo).

Ogni subshell non eredita le variabili locali della shell padre.

**Quando una shell deve eseguire uno script esegue queste operazioni:**

- Legge la prima riga dello script in cui è indicato quale interprete di comandi deve eseguire lo script  
es: `#!/bin/bash`
- **Crea una subshell**
- Il nome dello script viene passato come argomento (opzione -c) alla nuova subshell
- La nuova subshell esegue lo script.
- Alla fine dell'esecuzione dello script la subshell termina e restituisce il controllo alla shell padre, restituendo un valore intero che indica il risultato.

# Nozioni per uso del Terminale: Variabili (4)

Ogni shell supporta due tipi di variabili

## Variabili locali

Non “trasmesse” da una shell alle subshell da essa create

Utilizzate per computazioni locali all’interno di uno script

## Variabili di ambiente

“Trasmesse” dalla shell alle subshell.

Viene creata una copia della variabile per la subshell

Se la subshell modifica la sua copia della variabile,  
la variabile originale nella shell non cambia.

Solitamente utilizzate per la comunicazione fra parent e child shell

es: variabili \$HOME \$PATH \$USER %SHELL \$TERM

Per visualizzare l’**elenco delle variabili di ambiente**, utilizzare il comando **env** (ENVironment)

Quando dichiaro una variabile con la sintassi già vista dichiaro una variabile LOCALE.

**nomevariabile=ValoreVariabile**

Per trasformare una variabile locale già dichiarata in una variabile di ambiente, devo usare il comando **export** (notare che non uso il \$ )

**export nomevariabile**

Posso anche creare una variabile dichiarandola subito di ambiente

**export nomevariabile=ValoreVariabile**

# Nozioni per uso del Terminale: Variabili (5)

Esempio per esplicitare differenza tra var locali e var d'ambiente: ./var\_caller.sh

## # var\_caller.sh

```
echo "caller"  
# setto la var locale PIPPO  
# la var d'ambiente PATH esiste già
```

```
PIPPO=ALFA  
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"
```

```
echo "calling subshell"
```

## ./var\_called.sh

```
echo "ancora dentro caller"  
echo "variabili sono state modificate ?"
```

```
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"
```

## # var\_called.sh

```
echo "called"  
echo "le variabili sono state passate ? "
```

```
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"
```

```
echo "modifico variabili "
```

```
PIPPO="${PIPPO}:MODIFICATO"  
PATH="${PATH}:MODIFICATO"
```

```
echo "PATH = ${PATH}"  
echo "PIPPO = ${PIPPO}"
```

```
echo "termina called"
```

# Nozioni per uso del Terminale: Variabili (6)

Quando invoco uno script,  
viene eseguita una nuova shell  
al termine dell'esecuzione quella shell viene eliminata

Per ogni eseguibile invocato,  
viene collocata in memoria una copia dell'ambiente di esecuzione,  
l'eseguibile può modificare il proprio ambiente,  
al termine dell'esecuzione l'ambiente viene eliminato dalla memoria.



# Script execution senza creazione di subshell

Normalmente, se una shell lancia uno script, questo viene eseguito in una subshell

- con l'interprete di comandi indicato nella prima riga speciale dello script `#!/bin/bash`, se esiste quella riga,
- o con lo stesso interprete della shell chiamante, se quella prima riga non esiste.

Se lo script modifica le proprie variabili, la shell padre non si accorge delle modifiche.

**Però, una shell può eseguire uno script senza creare la subshell in cui lo script dovrebbe essere eseguito.**

Come ?

**source** nomescrit

Oppure

▪ nomescrit

In tal modo non viene considerata la prima riga speciale dello script.

**A cosa serve? A modificare le variabili della shell padre mediante uno script.**

Le variabili modificate dallo script sono proprio quelle della shell (padre, o meglio unica) , quindi la shell vede modificate le proprie variabili.

**source** e **▪** sono due comandi built-in ovvero implementati all'interno della shell

# Script execution senza creazione di subshell (2)

Attenzione che il comando `source` va usato solo se siamo sicuri che all'interno dello script da eseguire ci sono dei comandi che possono essere correttamente interpretati dalla shell `bash` chiamante.

Se invochiamo con `source` uno script che necessita di un interprete diverso provochiamo dei problemi, poiché **non verrà lanciato l'interprete corretto indicato nella prima riga dello script.**

Ad esempio, se lo script `perl myperl.pl` contiene le seguenti 3 righe:

```
#!/usr/bin/perl
```

```
@famiglia = ("padre", "madre", "figlio", "figlia");
```

```
for ($i=0; $i<=$#famiglia; $i++) { print "$famiglia[$i]\n"; }
```

ed io, all'interno di una `bash`, eseguo lo script con `source`

```
source ./myperl.pl
```

provoco degli errori perché non viene lanciato l'interprete `/usr/bin/perl` ed è la `bash` chiamante che cerca di interpretare i comandi `perl`

```
bash: ./myperl.pl: line 3: syntax error near unexpected token `('
```

```
bash: ./myperl.pl: line 3: `@famiglia = ("padre", "madre", "figlio", "figlia");'
```

# Nozioni per uso del Terminale: Variabili (7)

Esempio per vedere differenza tra var locali e var d'ambiente con source: ./var\_caller.sh

## # var\_caller2.sh

```
echo "caller"
# setto la var locale PIPPO
# la var d'ambiente PATH esiste già

PIPPO=ALFA
echo "PATH = ${PATH}"
echo "PIPPO = ${PIPPO}"

echo "calling subshell"

source ./var_called2.sh

echo "ancora dentro caller"
echo "variabili sono state modificate ?"

echo "PATH = ${PATH}"
echo "PIPPO = ${PIPPO}"
echo "NUOVA= ${NUOVA}"
```

## # var\_called2.sh

```
echo "called"
echo "le variabili sono state passate ? "

echo "PATH = ${PATH}"
echo "PIPPO = ${PIPPO}"

echo "modifico variabili "

PIPPO="${PIPPO}:MODIFICATO"
PATH="${PATH}:MODIFICATO"

echo "creo nuova variabile"
NUOVA=NUOVOCONTENUTO

echo "PATH = ${PATH}"
echo "PIPPO = ${PIPPO}"
echo "NUOVA= ${NUOVA}"

echo "termina called"

# che accade se DEcommento qui sotto?
# exit 13
```

# Nozioni per uso del Terminale: Variabili (8)

## Creazione di variabili d'ambiente da passare a subshell e Campo di visibilit  di variabili

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

Per esempio:

```
var="stringa" comando  
echo ${var}
```

```
# comando vede e puo' usare var  
#echo non vede la variabile var
```

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

Per esempio:

```
export var="contenutoiniziale"
```

```
var="stringa" comando  
echo ${var}
```

```
# comando vede var che vale "stringa"  
#echo vede la variabile che vale "contenutoiniziale"
```

# Nozioni per uso del Terminale: Variabili (9)

Esempio per esplicitare creazione di variabili da passare a subshell

Notare che tali variabili diventano automaticamente **di ambiente per la subshell**

```
# init_var_caller3.sh
```

```
#!/bin/bash
```

```
VAR="contenutoiniziale"
```

```
# non cambia se lascio VAR locale
```

```
# o se la imposto come variab. di ambiente
```

```
# export VAR
```

```
echo "VAR = ${VAR}"
```

```
VAR="stringa" ./init_var_called.sh
```

```
# non cambia se lo chiamo con source
```

```
# provarlo anche così
```

```
# VAR="stringa" source /init_var_called.sh
```

```
echo "VAR = ${VAR}"
```

```
# init_var_called.sh
```

```
#!/bin/bash
```

```
echo "la variabile e' stata passata ? "
```

```
echo "VAR = ${VAR}"
```

```
# Verifico se nuova VAR e' di ambiente
```

```
echo "env | grep VAR"
```

```
env | grep VAR
```

# Nozioni per uso del Terminale: Variabili (10)

## Variabili Vuote vs. Variabili non esistenti

C'e' differenza tra  
una variabile che esiste ma e' vuota  
ed una variabile che invece non esiste.

- Se una variabile non e' mai stata dichiarata allora non esiste.
- Se a una variabile e' stato assegnato come valore la stringa vuota "" allora esiste ma e' vuota.

Posso eliminare una variabile esistente (vuota o no) col comando **unset**

## unset nomevariabile

Dopo averla eliminata, quella variabile non esiste.

Attenzione, che una variabile vuota o non esistente puo' provocare errori sintattici a runtime negli script in cui si confronta il valore delle variabili.

[ \$var = "" ]      errore sintattico se \$var e' vuota oppure se non esiste  
e' come se ci fosse scritto      [ = "" ]

## Occorre sempre quotare il nome della variabile

[ "\$var" = "" ]      corretto sintatticamente se \$var e' vuota e anche se var non esiste

[ \$var = "" ]      SBAGLIATO se var vuota o non esiste, come se fosse scritto [ = "" ]

```
[ -n "$var" ]      # corretto , restituisce true se var non vuota, false se var vuota
```

```
[ -n $var ] # SBAGLIATO perche' restituisce vero se var non esiste
```

# Nozioni per uso del Terminale: funzionalità **history**

## **history expansion**

La funzionalità **history** memorizza i comandi lanciati dalla shell e permette di visualizzarli ed eventualmente rilanciarli. La storia dei comandi viene mantenuta anche dopo la chiusura della shell.

Il comando built-in **history** visualizza un elenco numerato dei comandi precedentemente lanciati dalla shell, con numero crescente dal comando più vecchio verso il più recente. Il numero assegnato ad un comando quindi non cambia quando lancio altri comandi. Il numero serve a rilanciare quel comando.

<i>più vecchio</i>	180	ls
	181	./sizeof_vettore.exe
	182	clear
	183	cat sizeof_pached.c
	184	gcc -ansi -o sizeof_packed.exe sizeof_packed.d
<i>più recente</i>	185	./sizeof_packed.exe

Se lancio il comando **!NUMERO** eseguo il comando che nella history è indicato col numero **NUMERO**. Se ad esempio lancio

**!181**

allora eseguo `./sizeof_vettore.exe`

Se lancio il comando **!stringa** allora eseguo il comando più recentemente lanciato (cioè quello col numero più grande) che nella history inizia con **stringa**. Se ad esempio lancio

**!c**

allora eseguo `cat sizeof_pached.c`

# Nozioni per uso del Terminale: Comando **set**

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

**set** lanciato senza nessun parametro visualizza tutte le variabili della shell, sia quelle locali che quelle d'ambiente. Inoltre visualizza le funzioni implementate nella shell (le vedremo piu' avanti (forse)).

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

Ad esempio

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

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

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

**set +a** causa il fatto che le variabili create sono variabili locali. E' il modo di default.

Vedere il comando **man set** per leggere tutte le possibili opzioni di comportamento della shell che possono essere attivate o disattivate col comando **set**.

Se il man non trova le info su set, installate il pacchetto manpages-posix-dev



# Nozioni per uso del Terminale: Comando **set**

## Creare variabili locali dopo **set -a**

Dopo avere eseguito in una bash il comando **set -a**  
le variabili create successivamente sono variabili di ambiente e non variabili locali.

Per configurare comunque una variabile affinché sia una variabile locale (non di ambiente) occorre utilizzare il comando **export** con il flag **-n**

Ad esempio, lanciando il seguente script `var_callerLocale.sh`, in cui la variabile `PLUTO` viene dichiarata locale mediante il comando `export -n PLUTO` ottengo che la variabile `PLUTO` non esista nell'ambiente di esecuzione dello script `var_calledLocal.sh` chiamato dal primo script. La variabile `PLUTO` non era di ambiente.

*var\_callerLocale.sh*

```
#!/bin/bash

set -a
PIPPO=pippo
PLUTO=pluto
echo "PIPPO = ${PIPPO}"
echo "PLUTO = ${PLUTO}"

export -n PLUTO
./var_calledLocal.sh
```

*var\_calledLocal.sh*

```
#!/bin/bash

echo dentro
var_calledLocal.sh

echo "PIPPO = ${PIPPO}"
echo "PLUTO = ${PLUTO}"
```

L'output  
dell'esecuzione  
è quello qui a destra:

```
PIPPO = pippo
PLUTO = pluto
PIPPO = pippo
PLUTO = 65
```

# Parametri a riga di comando passati al programma (1)

## Cosa sono 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.

Quando da una shell digito un comando da eseguire, gli argomenti devono perciò essere digitati assieme (e di seguito) al nome del programma per costituire l'ordine da dare alla shell. Ciascun argomento è separato dagli altri mediante uno o più spazi (carattere blank, la barra spaziatrice della tastiera).

Esempio: `chmod u+x /home/vittorio/primo.exe`

L'intera stringa di caratteri **`chmod u+x /home/vittorio/primo.exe`** si chiama **riga di comando**

Il primo pezzo **`chmod`** è il nome dell'eseguibile (o anche **argomento di indice zero**). Ciascuno degli altri pezzi è un argomento o parametro.

Nell'esempio, il pezzo `u+x` è l'**argomento di indice 1**.

Nell'esempio, il pezzo `/home/vittorio/primo.exe` è l'**argomento di indice 2**.

Nell'esempio, si dice che il numero degli argomenti passati è **2**.

Il programma, una volta cominciata la propria esecuzione, ha modo di conoscere il numero degli argomenti che gli sono stati passati e di ottenere questi argomenti ed utilizzarli.

# Avvio della shell bash

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

Distinguiamo 3 modi in cui può essere eseguita la bash:

## **shell non interattiva**

shell figlia che esegue script

lanciata con argomenti **-c** percorso\_script\_da\_eseguire

## **shell interattiva NON di login**

quella che vediamo all'inizio nella finestra di terminale

lanciata senza nessuno degli argomenti **-c -l --login**

## **shell interattiva di login**

è come la shell non di login, ma inizia chiedendo user e password

lanciata con argomenti **-l** oppure **--login**

# shell bash interattiva "di login" o "non di login"

La shell interattiva bash si comporta inizialmente in due modi differenti a seconda di come viene lanciata cioè a seconda di quali argomenti a riga di comando le vengono passati.

**Shell bash interattiva di login:** quando viene lanciata **con** una delle opzioni **-l** oppure **--login**.

La shell interattiva **di login**, nel momento in cui comincia la sua esecuzione, cerca di eseguire (se esistono) i seguenti files:

1. il file `/etc/profile`
2. poi uno solo (il primo che trova) tra i file `.bash_profile`, `.bash_login` e `.profile` collocati nella home directory dell'utente e che risulti essere disponibile;
3. poi il file `.bashrc` collocato nella *home directory* dell'utente;

Nel momento in cui termina, la shell di login esegue il file `.bash_logout` collocato nella home dell'utente (se il file esiste).

**Shell bash interattiva non di login:** quando viene lanciata **senza** alcuna delle due opzioni **-l** oppure **--login**.

La shell interattiva **non di login**, nel momento in cui comincia la sua esecuzione, cerca di eseguire (se esiste) il solo file **`.bashrc`** collocato nella *home directory* dell'utente.

Per entrambe le shell, l'utente può modificare i file di configurazione nella propria home directory, per customizzare il comportamento della shell.

# shell bash "non interattiva"

**La shell non interattiva bash è la shell lanciata per eseguire uno script o un eseguibile.**

**Viene specificato un argomento `-c nomescrptdaeseguire`.**

Ad esempio, potrei usare questa sintassi per far eseguire un file con comandi:

```
/bin/bash -c nomemioscript.sh arg1 arg2 arg3
```

NB: Questa sintassi viene utilizzata anche nelle moderne applicazioni costituite da container per specificare quale è il comando che deve essere eseguito all'interno di un container subito dopo la creazione del container stesso.

**A differenza della shell interattiva, la shell non interattiva NON esegue i comandi contenuti in nessuno dei file (di sistema o dell'utente) che customizzano il comportamento della shell interattiva ( `/etc/profile` `.bash_profile` `.bash_login` `.profile` `.bashrc` ).**

# Parametri a riga di comando passati al programma (2)

## Separatore di comandi ; e Delimitatore di argomenti “

### **Problema: Comandi Multipli su una stessa riga di comando**

La bash permette che **in una stessa riga di comando possano essere scritti più comandi**, che verranno eseguiti uno dopo l'altro, **non appena termina un comando viene eseguito il successivo**.

Il carattere **;** è il **separatore tra comandi**, che stabilisce cioè dove finisce un comando e dove inizia il successivo.

Ad es: se voglio stampare a video la parola pippo e poi voglio cambiare la directory corrente andando in /tmp potrei scrivere in una sola riga i due seguenti comandi:

```
echo pippo ; cd /tmp
```

**Diventa però impossibile stampare a video frasi strane tipo:        pippo ; cd /bin**  
**che contiene purtroppo il carattere di separazione ;**

Per farlo si include la frase completa tra doppi apici. Infatti, **il doppio apice “ (double quote) serve a delimitare l'inizio e la fine di un argomento, così che il punto e virgola viene visto non come un delimitatore di comando ma come parte di un argomento**.

```
echo “pippo ; cd /tmp “
```

In questo modo si **visualizza** la frase **pippo ; cd /tmp** e non si cambia directory 😊

# Parametri a riga di comando passati al programma (3)

## Quoting di stringhe " e Quoting di singoli caratteri \

Consideriamo ancora il comando appena visto `echo " pippo ; cd /tmp "`

### NOTA BENE 1:

Oltre al delimitatore di argomenti `"` esiste anche il delimitatore `'`

Vedremo la differenza piu' avanti parlando di substitution.

### NOTA BENE 2:

Delimitandola con un delimitatore, la stringa `" pippo ; cd /tmp"` appare al comando `echo` come un unico argomento.

### NOTA BENE 3:

E' possibile disabilitare l'interpretazione del separatore `;` facendolo precedere da un ulteriore carattere speciale detto carattere di escape `\`

In tal modo, il comando `echo pippo \; cd /tmp`  
visualizzera' `pippo ; cd /tmp`

### NOTA BENE 4: ESCAPE per QUOTING di singoli caratteri speciali

In generale, precedendo un carattere speciale con il carattere `\` (quoting di singolo carattere) si ottiene che il carattere speciale verra' utilizzato come un carattere qualunque, senza la sua speciale interpretazione da parte della shell. **71**

# Brace Expansion - generazione stringhe (1)

## concetto di base

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.

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

Ad esempio, se la riga di comando è fatta così:

```
echo va{acaga,ffancu,ammori,catihafat}lo
```

il comando viene espando in questo modo

```
echo vaacagalo vaffanculo vaammorilo vacatihafatlo
```

ottenendo il gentile output

vaacagalo vaffanculo vaammorilo vacatihafatlo **72**



# Brace Expansion - generazione stringhe (2)

## spazi bianchi nelle stringhe

**Possono mancare preambolo o postscritto o entrambi**

a{bb,cc,ddd} diventa abb acc addd

**Non possono essere presenti spazi bianchi o tab** altrimenti la brace expansion non viene riconosciuta e non viene applicata l'espansione.

Il comando: echo a{bb,cc, dd}ee

ottiene questo output: a{bb,cc, dd}ee

Analogamente, il comando echo a{bb,cc,d d}ee

ottiene questo output : a{bb,cc,d d}ee

**Se voglio far generare stringhe con spazi bianchi, devo proteggere i singoli spazi** con dei caratteri backslash

Ad esempio, echo a{bb,cc,d\ d}ee

ottiene questo output: abbee accee ad dee

Ad esempio, echo aa\ a{bb,cc,dd}ee

ottiene questo output: aa abbee aa accee aa addee

**Non posso proteggere tutto l'ordine di brace expansion con doppi apici perché** disabilitano l'interpretazione dell'espansione, ma posso proteggere ciascuna singola stringa (quella tra due virgole)

Il comando echo "a{bb,cc,d d}ee" ottiene in output a{bb,cc,d d}ee

mentre il comando echo a{bb,cc,"d d"}ee ottiene in output abbee accee ad dee

# Brace Expansion - generazione stringhe (3)

## annidamento e variabili

### Annidamento delle brace expansion

E' possibile inserire delle brace expansion all'interno di altri ordini di brace expansion.

Il comando: `echo /usr/{ucb/{ex,edit},lib/{bin,sbin}}`

ottiene questo output:

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

E' possibile inserire più livelli di annidamento delle brace expansion.

**Vengono eseguite per prime le brace expansion più all'esterno.**

### Variabili nelle brace expansion

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

`A=bin`

`B=log`

`C=boot`

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

visualizza

```
binlogboota binboota binbinloga
```

**Vengono eseguite prima le brace expansions e poi le variable expansions.**

# Brace Expansion - generazione stringhe (4)

## brace expansion con Sequence Expression

Esiste una particolare forma di brace expansion che consente di generare stringhe elencando un intervallo di caratteri che possono essere usati come possibili scelte. L'intervallo viene indicato mediante i due estremi connessi con due punti

Il comando	<code>echo a{b..k}m</code>
ottiene questo output	<code>abm acm adm aem afm agm ahm aim ajm akm</code>

Il comando	<code>echo a{4..7}m</code>
ottiene questo output	<code>a4m a5m a6m a7m</code>

E' possibile annidare queste sequence expression all'interno di brace expansion annidate

Il comando	<code>echo a{b,c{4..7}}m</code>
ottiene questo output	<code>abm ac4m ac5m ac6m ac7m</code>

NB: Esiste una forma più complessa di sequence expression, in cui si può stabilire un incremento non unitario nell'intervallo di caratteri, ma questa forma non è supportata in tutte le versioni della bash, quindi non ne parliamo.

# Tilde Expansion ~

La tilde expansion riguarda 5 casi essenziali:

1. un carattere tilde isolato
  2. un carattere tilde seguito da slash non quotato
  3. un carattere tilde seguito da slash non quotato seguito da altri caratteri
- In questi casi la tilde viene sostituita dal percorso assoluto della home directory dell'utente che sta eseguendo la riga di comando, l'effective user.

Supponendo che l'utente che esegue i comandi sia l'utente vittorio

cd	~	cambia la directory corrente in	/home/vittorio
echo	~/	visualizza	/home/vittorio/
echo	~/vaff	visualizza	/home/vittorio/vaff

4. una parola che inizia con la tilde seguita da un nome utente
  5. una parola che inizia con la tilde seguita da un nome utente, seguita da slash non quotato a cui possono seguire altri caratteri
- In questi casi la tilde più nome utente viene sostituita dal percorso assoluto della home directory dell'utente specificato dal nome.

Supponendo che esista un utente panzieri e che a eseguire sia vittorio

echo	~panzieri	visualizza	/home/panzieri
echo	~panzieri/ciao	visualizza	/home/panzieri/ciao

Se l'utente specificato non esiste, l'espansione non viene fatta, stringa invariata

echo	~panzieriNonEsiste	visualizza	~panzieriNonEsiste
------	--------------------	------------	--------------------

# Wildcards \* ? [...]

## Pathname substitution

I Metacaratteri \* e ? sono caratteri che vengono inseriti dall'utente nei comandi digitati e che la shell interpreta cercando di sostituirli con una sequenza di caratteri per ottenere i nomi di files nel filesystem

Con cosa sono sostituiti?

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

? può essere sostituito da esattamente un singolo carattere.

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

Esempi: usiamo il comando **ls** che visualizza i nomi dei file nella directory specificata.

Nessuna sostituzione

ls /home/vittorio visualizza i nomi di tutti i file della directory

ls /home/vittorio/primo.c visualizza il nome del solo file primo.c

Sostituzione di \* con una qualunque sequenza di caratteri, anche vuota, che permetta di ottenere il nome di uno o più file

ls /home/vittorio/\*.exe visualizza il nome di quei file della directory vittorio il cui nome termina per .exe (cioè primo.exe)

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

Sostituzione di ? con un singolo carattere (NON vuoto),

ls /home/vittorio/pri?o.c visualizza il nome del file primo.c di directory vittorio

# Wildcards [...]

Cosa posso mettere dentro le parentesi quadre? E con cosa viene sostituito ?

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

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

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

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

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

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

Notare che le parentesi quadre selezionano uno solo tra i caratteri elencati dentro.

Nell'elenco possono comparire diverse sequenze di parentesi quadre.

Le sequenze speciali [:digit:] [:upper:] [:lower:] **devono stare dentro altre parentesi quadre esterne.**

Es: Supponiamo che in una directory ci siano i file: aB a1B a2B akB akmB akmtB

Allora:

```
ls a[:digit:]B
```

```
visualizza a1B a2B
```

```
ls a[:lower:][:lower:][:lower:]B
```

```
visualizza akmtB
```

Notare la differenza tra i seguenti comandi (occhio alle parentesi quadre annidate)

```
ls a[:lower:][:lower:]B
```

```
[[]] [[]]
```

```
visualizza akmB
```

```
ls a[:lower:][:lower:]B
```

```
[[] []]
```

```
visualizza akB
```

# Wildcards [...] - esempio di uso sbagliato

Non posso annidare delle parentesi quadre se non per contenere le parole riservate

[[:digit:]] [[:upper:]] [[:lower:]]

in una directory che contiene i file akB akmB a1B a2B

se scrivo

```
ls a[[:lower:]][[:digit:]]B
```

visualizzo akB a1B a2B

se scrivo

```
ls a[[:lower:]][12]B
```

le parentesi quadre [12] sono interne ad altre parentesi quadre, quindi non vengono sostituite con niente, ed il comando risponde:

```
ls: cannot access a[[:lower:]][12]B: No such file or director
```

# Comandi della bash ed eseguibili utili

## Comandi

### Comandi ed eseguibili

pwd cd mkdir rmdir ls rm echo cat set mv ps sudo du kill bg fg  
read wc killall

### Istruzioni di controllo di flusso

for do done while do done if then elif then else fi

### Espressione condizionale (su file o su variabile)

[ condizione di un file ]

### Valutazione di espressione matematica applicata a variabili d'ambiente

(( istruzione con espressione ))

## Eseguibili binari forniti dal sistema operativo

editor interattivi  
vi nano (pico)

### utilità

man more less grep find tail head cut tee ed tar gzip diff patch gcc make



# Comandi della bash (in ordine di importanza)

<b>pwd</b>	mostra directory di lavoro corrente .
<b>cd</b> <u>percorso_directory</u>	cambia la directory di lavoro corrente .
<b>mkdir</b> <u>percorso_directory</u>	crea una nuova directory nel percorso specificato
<b>rmdir</b> <u>percorso_directory</u>	elimina la directory specificata, se è vuota
<b>ls</b> -alh <u>percorso</u>	stampa informazioni su tutti i files contenuti nel percorso
<b>rm</b> <u>percorso_file</u>	elimina il file specificato
<b>echo</b> <u>sequenza di caratteri</u>	visualizza in output la sequenza di caratteri specificata
<b>cat</b> <u>percorso_file</u>	visualizza in output il contenuto del file specificato
<b>env</b>	visualizza le variabili ed il loro valore
<b>which</b> <u>nomefileeseguibile</u>	visualizza il percorso in cui si trova (solo se nella PATH) l'eseguibile
<b>mv</b> <u>percorso_file</u> <u>percorso_nuovo</u>	sposta il file specificato in una nuova posizione
<b>ps</b> aux	stampa informazioni sui processi in esecuzione
<b>du</b> <u>percorso_directory</u>	visualizza l'occupazione del disco.
<b>kill</b> -9 <u>pid_processo</u>	elimina processo avente identificativo pid_processo
<b>killall</b> <u>nome_processo</u>	elimina tutti i processi con quel nome nome_processo
<b>bg</b>	ripristina un job fermato e messo in sottofondo
<b>fg</b>	porta il job più recente in primo piano
<b>df</b>	mostra spazio libero dei filesystem montati
<b>touch</b> <u>percorso_file</u>	crea il file specificato se non esiste, oppure ne aggiorna data.
<b>more</b> <u>percorso_file</u>	mostra il file specificato un poco alla volta
<b>head</b> <u>percorso_file</u>	mostra le prime 10 linee del file specificato
<b>tail</b> <u>percorso_file</u>	mostra le ultime 10 linee del file specificato
<b>man</b> <u>nomecomando</u>	è il manuale, fornisce informazioni sul comando specificato
<b>find</b>	cercare dei files
<b>grep</b>	cerca tra le righe di file quelle che contengono alcune parole
<b>read</b> <u>nomevariabile</u>	legge input da standard input e lo inserisce nella variabile specificata
<b>wc</b>	conta il numero di parole o di caratteri di un file
<b>true</b>	restituisce exit status 0 (vero)
<b>false</b>	restituisce exit status 1 (non vero)

# Funzione di Autocompletamento della bash.

## Tasto TAB (tabulazione)

Mentre sto digitando dei comandi può capitare di dover specificare il percorso assoluto o relativo di un file o di una directory.

Se comincio a scrivere questo percorso, **posso premere il tasto TAB** (tabulazione) il quale attiva la funzione di autocompletamento dei nomi di files o directory.

Tale funzione produce ogni volta uno dei quattro seguenti risultati:

1. Se non esiste nessun percorso che comincia con la parte di percorso scritta  
**Non accade nulla, viene solo lanciato un beep di avvertimento.**
2. Se esiste un solo percorso che comincia con la parte di percorso scritta  
**Viene aggiunta alla parte scritta la parte mancante per completare il percorso.**
3. Se esistono più percorsi che cominciano con la parte di percorso scritta e tutti questi percorsi hanno una parte di percorso comune oltre a quella già scritta  
**Viene aggiunta alla parte scritta la parte comune a tutti i percorsi.**
4. Se esistono più percorsi che cominciano con la parte di percorso scritta ma tutti questi non hanno altra parte di percorso in comune oltre a quella già scritta  
**non accade nulla, viene solo lanciato un beep di avvertimento.**  
In questo caso, **se premo due volte TAB (rapidamente)**, viene visualizzato l'elenco dei **possibili percorsi che possono essere scelti per completare il percorso già scritto.**

# Parametri a riga di comando passati al programma (4)

## Parameter Expansion

**Come utilizzare in uno script gli argomenti a riga di comando passati allo script**

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

**\$#** il numero di argomenti passati allo script

**\$0** il nome del processo in esecuzione

**\$1** primo argomento, **\$2** secondo argomento, .....

**\$\*** tutti gli argomenti passati a riga di comando concatenati e separati da spazi

**\$@** come **\$\*** ma se quotato gli argomenti vengono quotati separatamente

Esempio: All'interno di uno script posso usarle così :

file esempio\_script.sh

```
echo "ho passato $# argomenti alla shell"
echo "tutti gli argomenti sono $*"
```

**NOTA BENE: I parametri NON POSSONO essere modificati**

**NOTA BENE: Il programma vede i parametri COSI' COME SONO DIVENTATI DOPO LA EVENTUALE SOSTITUZIONE DEI METACARATTERI \* e ?**

Ad esempio, se nella directory corrente ci sono i seguenti file x.c, y.c e z.c, ed io lancio lo script, che contiene le 2 righe sopra riportate, in due modi diversi, vengono stampati in output i seguenti argomenti:

**SENZA metacaratteri**

```
./esempio_script.sh pippo
ho passato 1 argomenti alla shell
tutti gli argomenti sono pippo
```

**CON metacaratteri**

```
./esempio_script.sh *.c
ho passato 3 argomenti alla shell
tutti gli argomenti sono x.c y.c z.c
```

# Parametri a riga di comando passati al programma (5)

File esempio\_args.sh

```
#!/bin/bash

echo "ho passato $# argomenti alla shell"
echo "il nome del processo in esecuzione e' $0"
echo "gli argomenti passati a riga di comando sono $*"

for name in $* ; do
    echo "argomento e' ${name}" ;
done
```

eseguitelo chiamandolo con diversi argomenti e delimitatori

./esempio_args.sh	
./esempio_args.sh	alfa beta gamma
./esempio_args.sh	"alfa beta gamma"
./esempio_args.sh	"alfa beta" gamma

NB: la variabile name dichiarata nel for rimane utilizzabile anche dopo l'uscita dal for poiché il for viene eseguito nella bash stessa, non in una bash figlia.

La variabile name fuori dal for avrà l'ultimo valore che le era stato assegnato

# Parametri a riga di comando passati al programma (6)

## Differenze tra \$\* e @\$

I parametri \$\* e @\$ sono uguali quando non quotati dai ", cioè sono la concatenazione separata da blank dei singoli argomenti.

`$* == @$ ---> $1 $2 $3 ... $n`

I parametri \$\* e @\$ si comportano diversamente quando sono quotati con " in particolare:

`"$*" ---> "$1 $2 $3 ... $n"` quotati tutti gli argomenti assieme

`"@$" ---> "$1" "$2" "$3" ... "$n"` quotato singolarmente ogni argomento

# Parametri a riga di comando passati al programma (6bis)

Il parametro `$@` e' simile a `$*` cioè contiene tutti gli argomenti passati allo script, **ma quando quotato con i " " mantiene quotati i singoli argomenti e permette di non spezzare gli argomenti che contengono degli spazi.**

File `esempio_\\$\\@.sh`.

Eseguitelo con diversi argomenti e delimitatori

`esempio_\\$\\@.sh` alfa beta gamma

`esempio_\\$\\@.sh` "alfa beta" gamma

```
echo 'for con $* non quotato'
for name in $* ; do
    echo "argomento ${name}" ;
done
echo 'for con "$*" quotato'
for name in "$*" ; do
    echo "argomento ${name}" ;
done
echo 'for con $@ non quotato'
for name in $@ ; do
    echo "argomento ${name}" ;
done
echo 'for con "$@" quotato'
for name in "$@" ; do
    echo "argomento ${name}" ;
done
```

**NON SOLO PER I  
MANIACI**

**NB: `$@` si usa anche  
quando uno script  
deve eseguire un  
altro comando  
passandogli tutti gli  
argomenti che ha  
ricevuto a riga di  
comando**

# Valutazione Aritmetica di espressioni **con soli interi (1):** operatori **(( ))** e **\$(( ))**

E' possibile valutare una stringa come se fosse una espressione costituita da operazioni aritmetiche **tra soli numeri interi**

**L'operatore (( )) racchiude TUTTA una riga di comando semplice**, che deve essere una espressione (più un eventuale assegnamento).

L'operatore (( )) esegue la riga di comando che racchiude valutando aritmeticamente gli operandi.  
Ad esempio

```
(( NUM=3+2 ))
```

assegna 5 alla variabile NUM

**L'operatore \$(( )) invece serve se dovete racchiudere solo UNA PARTE di una riga di comando, che deve essere una espressione (più un eventuale assegnamento).**

La bash

- prima valuta aritmeticamente l'espressione racchiusa dall'operatore \$(( ))
- poi, nella riga di comando, **mette al posto** del \$((...)) il risultato calcolato dall'operatore \$(( ))
- infine, esegue la riga di comando modificata

Ad esempio, nella riga di comando

```
echo stampa pippo$((3+2))
```

la bash

prima valuta l'espressione \$((3+2)) e ne calcola il risultato che è 5

poi nella riga di comando originale sostituisce 5 al posto di \$((3+2))

ottenendo la nuova riga di comando

```
echo stampa pippo5
```

infine esegue quest'ultima riga di comando visualizzando a video

```
stampa pippo5
```

# Valutazione Aritmetica di espressioni **con soli interi (2):** operatori **(( ))** e **\$(( ))**

Errori sintattici nell'espressione provocano errori.

```
NUM=1
NUM=${NUM}+3
echo
${NUM}
stampa a video la stringa 1+3
```

```
NUM=1
((NUM=${NUM}+3))
echo ${NUM}
uso corretto di operatore (( ))
stampa a video la stringa 4
```

```
NUM=1
NUM=(( ${NUM}+3))
echo ${NUM}
uso SBAGLIATO di operatore (( ))
stampa a video
"syntax error near unexpected token ("
```

```
NUM=1
NOMEFILE=pippo${NUM}+3
echo ${NOMEFILE}
stampa a video la stringa pippo1+3
```

```
NUM=1
(( NOMEFILE=pippo${NUM}+3 ))
echo ${NOMEFILE}
uso sbagliato di operatore (( ))
impossibile valutare aritmeticamente pippo
stampa a video la stringa 3
```

```
NUM=1
NOMEFILE=pippo$(( ${NUM}+3 ))
echo ${NOMEFILE}
uso corretto di operatore $(( ))
stampa a video la stringa pippo4
```



# Valutazione Aritmetica di espressioni **con soli interi (3):** operatori `(( ))` e `$(( ))`

Se dovete racchiudere tutto un comando, allora DOVETE usare `(( ))` e NON potete usare `$(( ))`  
Errori sintattici nell'espressione provocano errori.

Ad esempio:

```
A=1  
$(( B=$A+1 ))
```

`-bash: 2: command not found`

La bash valuta il contenuto dentro le tonde, poi sostituisce al contenuto quello che ha ottenuto dalla valutazione, infine esegue il comando dopo la sostituzione. Il comando dopo la sostituzione è 2. La bash cerca di eseguire 2 che non è un comando esistente e quindi genera un errore.

Analogamente:

```
B=3  
while read A && $(( A!=$B )) ; do echo $A ; done
```

digitare qualcosa a tastiera, ad esempio 3  
accade

`-bash: 0: command not found`

# Valutazione Aritmetica di espressioni **con soli interi** (4): operatori `(( ))` e `$( ( ))`

Le valutazioni aritmetiche possono contenere :

- degli operatori aritmetici `+` `-` `*` `/` `%`
- degli assegnamenti
- delle parentesi tonde `()` per accorpare operazioni e modificare precedenze.

Quando usate dentro una valutazione aritmetica, le parentesi tonde non creano una nuova bash.

Come vedremo più avanti, le valutazioni aritmetiche possono essere utilizzate come condizione di `while` e di `if`

```
NUM=33
while (( NUM=$((NUM*NUM*NUM))%10 ) != 2 )) ; do echo ${NUM} ; done
```

# Valutazione matematica più complesse: il programma Basic Calculator **bc** e il suo linguaggio di programmazione (1)

Rospi maledetti, questo mi basta che sappiate che esiste,  
nel caso vi dovesse servire ve lo studierete e approfondirete da soli:

Un esempio: la seguente concatenazione di comandi

```
echo " ( 32.16 - 14.03 ) / 3.33 " | bc -l -q
```

visualizza sullo standard output

```
5.444444444444444444444444
```

# Valutazione matematica più complesse: il programma Basic Calculator **bc** e il suo linguaggio di programmazione (2)

Rospi maledetti, questo mi basta che sappiate che esiste,  
nel caso vi dovesse servire ve lo studierete e approfondirete da soli:

Un esempio piu' complesso con la definizione di una funzione utente:  
la seguente concatenazione di comandi

```
echo ' /*  radice.b */
define r (x) {
  auto z, y
  z=0
  y=0
  while (1) {
    y=(z*z)
    if (y>x) {
      /* È stato superato il valore massimo. */
      z=(z-1)
      return (z)
    }
    z=(z+1)
  }
}
"calcolo la radice quadrata intera, r (x): "
r(625)' | bc -l -q
```

visualizza la parte intera della radice quadrata intera di 625, ovvero 25.

**calcolo la radice quadrata intera, r (x): 25**

# Variable expansion

Qui si sarebbe dovuto collocare la nozione di variable expansion, cioè il fatto che **la bash espande il nome delle variabili, quando precedute dal \$, sostituendolo con il contenuto delle variabili stesse.**

Ma questo argomento lo abbiamo anticipato e lo abbiamo già visto per poter usare le variabili negli esempi per spiegare altri argomenti.

Un caso particolare di variable expansion non è ancora stato trattato, il caso dei riferimenti indiretti a variabili, che viene presentato nella slide successiva.

# Riferimenti Indiretti a Variabili

## Indirect References to Variables

(1)

operatore **`${!}`**  
solo in bash versione 2

Supponiamo di avere una prima variabile `varA` che contiene un valore qualunque. Supponiamo di avere una altra variabile il cui valore è proprio il nome della prima variabile.

Voglio usare il valore della prima variabile sfruttando solo il nome della seconda variabile il cui valore è proprio il nome della prima variabile. Si dice che la seconda variabile è un riferimento indiretto alla prima variabile.

Accedere al valore di una prima variabile il cui nome è il valore di una seconda variabile è possibile nella bash a partire dalla versione 2.

Si sfrutta un operatore **`${!}`**

Esempio:

```
varA=pippo
```

```
nomevar=varA
```

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

# Riferimenti Indiretti a Variabili

## operatore **`${!}`**

### solo in bash versione 2

(2)

File `esempio_while.sh` permette di riferirsi alle var `$1` `$2` `$3` da provare chiamandolo con diversi argomenti e delimitatori

`esempio_while.sh`

`esempio_while.sh` alfa beta gamma

`esempio_while.sh` "alfa beta gamma"

`esempio_while.sh` "alfa beta" gamma

```
#!/bin/bash
```

```
echo "ho passato $# argomenti alla shell"
```

```
echo "il nome del processo in esecuzione e' $0"
```

```
echo "gli argomenti passati a riga di comando sono $*"
```

```
NUM=1
```

```
while (( "${NUM}" <= "$#" ))
```

```
do
```

```
    # notare il ! davanti al NUM
```

```
    echo "arg ${NUM} is ${!NUM} "
```

```
    ((NUM=${NUM}+1))
```

```
done
```

# Exit Status

## Valore restituito da un programma al chiamante

### Differenza tra valore restituito e output

Ogni programma o comando restituisce un valore numerico compreso tra **0** e **255** per indicare se c'è stato un errore durante l'esecuzione oppure se tutto è andato bene. Un risultato **0 indica tutto bene** mentre un risultato **diverso da zero indica errore**.

Tale risultato non viene visualizzato sullo schermo bensì viene passato alla shell che ha chiamato l'esecuzione del programma stesso. In tal modo il chiamante può controllare in modo automatizzato il buon andamento dell'esecuzione dei programmi

Il risultato non è l'output fatto a video, che invece serve per far vedere all'utente delle informazioni.

### Come restituire il risultato in un programma C – l'istruzione return;

vedi esempio primo.c

### Come restituire il risultato in uno script bash – il comando exit

esempio: **exit 9**            fa terminare lo script e restituisce **9** come risultato

### Come catturare il risultato di un programma chiamato da uno script

Si usa una variabile d'ambiente predefinita **\$?** che viene modificata ogni volta che un programma o un comando termina e in cui viene messo il risultato numerico restituito dal comando o programma

```
./primo.exe
```

```
echo "il processo chiamato ha restituito come valore di uscita $? "
```



# Exit Status riservati

Per convenzione, gli eseguibili restituiscono un valore intero compreso tra **0 e 255** per indicare se è accaduto un errore e quale errore è accaduto.

Il valore 0 indica che l'esecuzione si è svolta correttamente.

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

# Exit Status di Espressione valutata aritmeticamente

**La valutazione aritmetica** effettuata tramite gli operatori `$(( ))`, oppure anche `(( ))` se comprende tutta la riga di comando, **restituisce un valore di Exit Status che sarà:**

**Diverso da zero** se durante la valutazione aritmetica è **accaduto un errore**. Il valore restituito indica il tipo di errore restituito.

`(( ls 5 ))`                      **non valutabile aritmeticamente**. `$?` conterrà 1

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

`(( 5 >= 2 ))`                      `$?` conterrà 0

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

`(( 5 <= 2 ))`                      `$?` conterrà 1

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

`(( 4 ))`                              `$?` conterrà 0

`(( VAR=5+3 ))`                      `$?` conterrà 0

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

`(( 0 ))`                              `$?` conterrà 1

`(( VAR=6-2*3 ))`                      `$?` conterrà 1

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

`(( (VAR=6-2*3) != 1 ))`                      assegna 0 a VAR ma `$?` conterrà 0  
perché la condizione è vera

# OCCHIO: Errori con la valutazione aritmetica

```
$ NUM=13
```

```
$ echo $?
```

```
0
```

```
$ echo ${NUM}
```

```
13
```

```
$ (( NUM=${NUM}+4 ))
```

```
$ echo $?
```

```
0
```

```
$ echo ${NUM}
```

```
17
```

```
$ (( NUM=ALFA ))
```

<----- causa errore

```
$ echo $?
```

```
1
```

<----- valore di errore

```
$ echo ${NUM}
```

```
0
```

<----- contenuto errato

# Liste di comandi

**Una lista di comandi è un elenco di comandi da lanciare in esecuzione in successione. Sintatticamente, ciascun elemento dell'elenco deve essere separato dal successivo da un punto e virgola. Ciascun elemento dell'elenco può essere :**

- ❑ **comando semplice** (o chiamata di script o di eseguibile binario)  
es:    cd            ./script.sh            primo.exe
- ❑ **espressione valutata aritmeticamente**  
(( VAR=(5+3)\*(2+\$VAR) ))
- ❑ **sequenza di comandi connessi da | pipe**                    *(le vedremo poco più avanti)*  
cat file.txt | grep stringa
- ❑ **sequenza di comandi condizionali**                    *(le vedremo poco più avanti)*  
gcc file.c && ./file.exe
- ❑ **raggruppamento di comandi**                    *(le vedremo poco più avanti)*  
( cat file1.c ; cat file 2,c )
- ❑ **espressione condizionale**                    *(le vedremo poco più avanti)*  
[[ -e file.txt && \$1 -gt 13 ]]

Esempio di lista di comandi: NUM=7; ./pippo.exe || if((\${NUM}>2)); then exit 3; fi; echo ok

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

NB: poiché le sequenza condizionale e le espressioni condizionali contengono dei comandi semplici e possono essere terminate senza avere eseguito tutti i comandi, in funzione dei risultati restituiti dai comandi eseguiti in precedenza, **può capitare che l'ultimo comando eseguito non sia l'ultimo a destra elencato nella lista****101**

# Compound Commands – Comandi composti

## Costrutti per controllo di flusso di comandi (1)

*Versione semplificata*

for varname in elencoword ; do list ; done

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

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

while list; do list; done

NOTA BENE:

Le espressioni expr\* dentro il for (( )) sono **sempre** valutate aritmeticamente, quindi devono essere espressioni interpretabili aritmeticamente.

Invece, i comandi dentro list\* sono valutati aritmeticamente solo se racchiudo i singoli comandi dentro doppie parentesi tonde.

Il risultato restituito da list\* e' il risultato dell'ultimo comando eseguito della lista oppure zero se nessun comando viene eseguito.

# Compound Commands – Comandi composti

## Costrutti per controllo di flusso di comandi (2)

### Esempi

```
for name in a b c ; do echo ${name} ; rm ${name} ; done
```

```
for (( i=0; $i<13; i=$i+2 )) ; do echo ciao$i ; done
```

```
if ls ./main.c && gcc -o main.exe main.c && ./main.exe ; then  
    echo main returns $?
```

```
fi
```

```
if [[ -e main.c && -e Makefile ]] && make && ./main.exe ; then  
    echo compiled program returns $?
```

```
fi
```

```
if [ -e main.c -a -e Makefile ] && make && ./main.exe ; then  
    echo compiled program returns $?
```

```
fi
```

```
while OUT=`./script1.sh` ; script2.sh $OUT ; do echo ancora done
```

```
i=0; while (( $i < 30 )) ; do echo $i ; (( i=$i+1 )) ; done
```

# Sequenze di comandi condizionali e non

## Sequenze non condizionali (vedi una slide precedente)

Il metacarattere “;” viene utilizzato per eseguire due o più comandi in sequenza ed indica la fine degli argomenti passati a ciascun comando riga di coman

**date ; ls /usr/vittorio/ ; pwd**

## Sequenze condizionali

“||” 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)**

## Esempi

Eseguire il secondo comando in caso di successo del primo

**\$ gcc prog.c -o prog && prog**

Eseguire il secondo comando in caso di fallimento del primo

**\$ gcc prog.c || echo Compilazione fallita**

# prossime slides

File descriptor

tabelle dei file aperti

stream di I/O predefiniti

ereditarietà dei file aperti

command substitution

quoting di stringhe:       " "   ' '



# File Descriptors

Il **file descriptor** è un'astrazione per permettere l'accesso ai file.

Ogni file descriptor è rappresentato da un integer.

Il concetto di file descriptor esiste sia usando linguaggio C ed anche in bash, perché è un concetto fornito dal sistema operativo.

```
#define MAXSIZE 100
#define nomefile "/home/vic/piffero.txt"
void main(void) {
    int fd, ris; char buffer[MAXSIZE];
    fd = open( nomefile, O_RDONLY );
    if( fd < 0 ) {    perror("open failed: "); exit(1); }

    /* letture fino a raggiungere fine file */
    do {
        ris=read( fd, buffer, MAXSIZE );
        if( ris<0 ) {    perror("read failed: "); exit(1); }
    } while( ris>0);

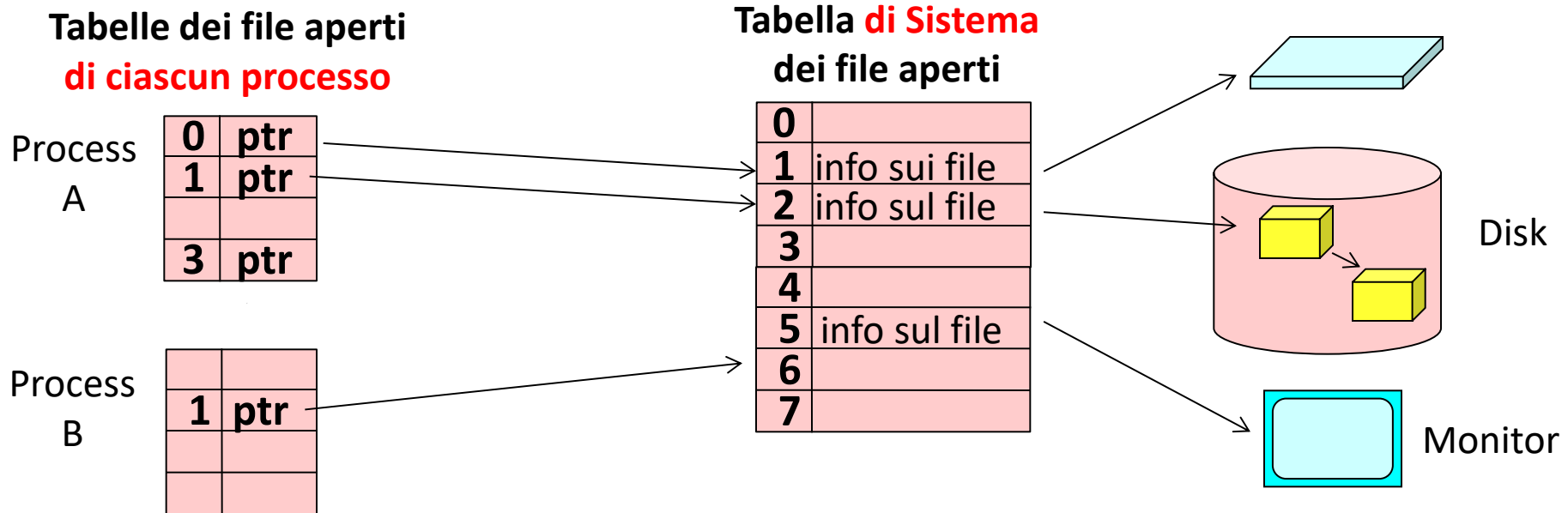
    close(fd);
}
```

# File Descriptors e Tabelle dei File Aperti

Il **file descriptor** è un'astrazione per permettere l'accesso ai file. Ogni file descriptor è rappresentato da un integer.

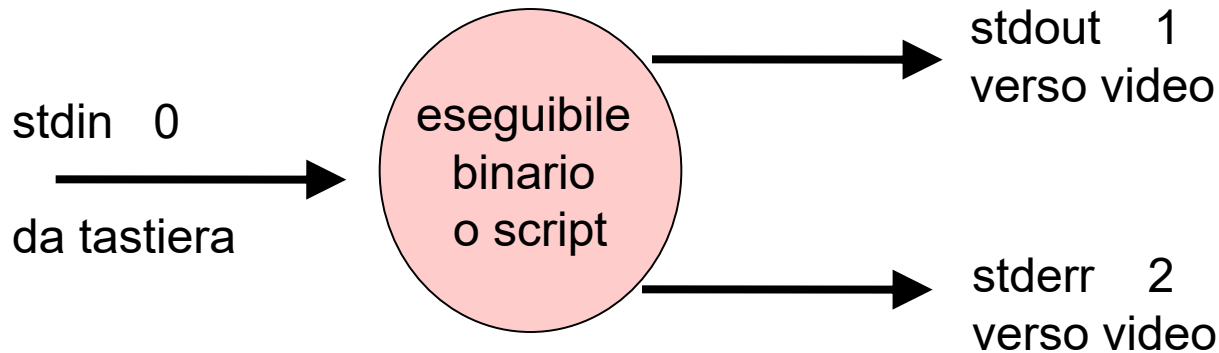
Ogni processo ha la propria **file descriptor table** che contiene (**indirettamente**) le informazioni sui file attualmente utilizzati (aperti) dal processo stesso. In particolare, la file descriptor table contiene un file descriptor (un intero) per ciascun file usato dal processo. Per ciascuno di questi file descriptor, la tabella del processo punta ad una **tabella di sistema** che contiene le informazioni sui tutti i file attualmente aperti dal processo stesso.

Si noti che due diversi processi possono avere due diversi file descriptor con stesso valore, ma che fanno riferimento a file diversi.



In POSIX ogni processo è inizialmente associato a 3 fd standard  
stdin(0), stdout (1), stderr (2)

# Stream di I/O predefiniti dei processi (1)



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

**STANDARD INPUT (stdin)** serve per l'input normale (per default da tastiera). Viene identificato da una costante valore numerico **0**.

**STANDARD OUTPUT (stdout)** serve per l'output normale (per default su schermo video). Viene identificato da una costante valore numerico **1**.

**STANDARD ERROR (stderr)** serve per l'output che serve a comunicare messaggi di errore all'utente (per default anche questo su schermo video). Viene identificato da una costante valore numerico **2**.

**Come fare output in un programma C – la funzione printf();**

esempio: vedere il file primo.c

**Come fare output in uno script bash – il comando echo**

esempio già visto: echo "ciao bella"

# Stream di I/O predefiniti dei processi (1bis)

## Chi e come produce dati diretti verso lo standard output

- Da linguaggio di scripting bash  
echo "stringa"  
printf "%f %.6s" 3.99999 piripicchio      NB: e' la printf della bash non del C  
vedere      man 1 printf  
confrontarlo con      man 3 printf
- Da linguaggio C  
printf ( "%f %.6s" , 3.99999 , "piripicchio");  
fprintf ( **stdout**, "%f %.6s" , 3.99999 , "piripicchio");

## Chi e come produce dati diretti verso lo standard error

- Da linguaggio di scripting bash, solo in caso di errori  
ls filechenonesiste  
produce "ls: cannot access filechenonesiste: no such file or directory"
- Da linguaggio C  
fprintf ( **stderr**, "%f %.6s" , 3.99999 , "piripicchio");

## Chi e come legge dati provenienti dallo standard input

- Da linguaggio di scripting bash  
read nomevar
- Da linguaggio C  
fgets( buffer , buffersize , **stdin** );  
scanf ( "%i" , &intvar );

# Stream di I/O predefiniti dei processi (1tris)

**Un esempio d'uso in C: legge da standard input e manda su standard error**

*/\* stdin\_stderr.c premere CTRL D per chiudere stdin \*/*

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
int main (void)
```

```
{    #define LINESIZE 1000
```

```
    char line[LINESIZE], *p;
```

```
    while (1) {
```

```
        p=fgets ( line , LINESIZE , stdin );
```

```
        if( p ) { /* lettura ok */
```

```
            fprintf ( stderr , "%s", line );
```

```
        }
```

```
        else if( feof ( stdin ) ) { /* EOF reached */
```

```
            return(0); /* return ok */
```

```
        }
```

```
        else { /* error */
```

```
            return(1);
```

```
        }
```

```
    }
```

```
    return(0); /* mai raggiunto */
```

```
}cwq
```

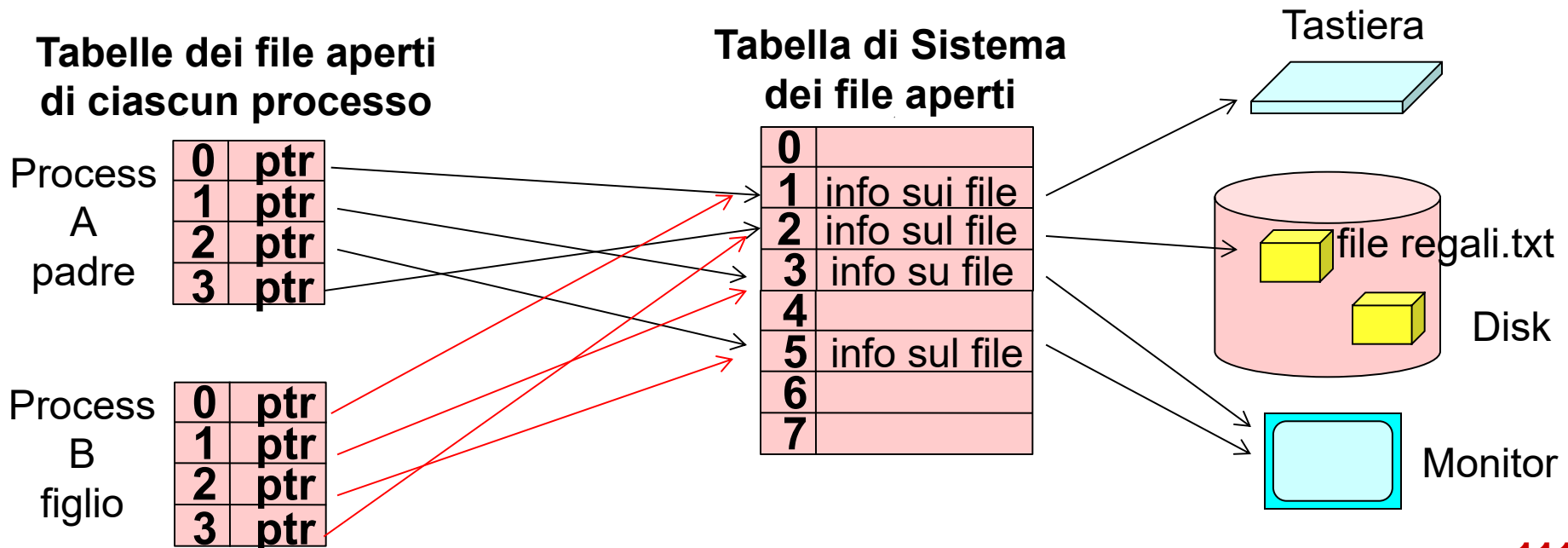
# Stream di I/O predefiniti dei processi (1ter)

## I processi figli ereditano gli Stream dei padri

Una shell B figlia di una shell padre A (e in generale, un processo B figlio di un processo A) eredita una copia della tabella dei file aperti del padre, quindi padre e figlio leggono e scrivono sugli stessi stream.

Se padre e figlio scrivono contemporaneamente potrebbero scrivere cose strane. Ereditare una copia degli stream del padre permette, ad esempio, che l'output di uno script sia visibile nella finestra di output della shell interattiva che l'ha lanciato.

**NB: Vengono ereditati gli stream predefiniti ed anche tutti gli altri stream.**



# Stream di I/O predefiniti dei processi (2): command substitution

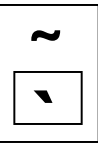
**Scopo:** sostituire a run time, in uno script, la riga di comando di un programma con l'output su stdout prodotto dall'esecuzione del programma stesso.

**Esempio:** come catturare, in una variabile, l'output di un programma chiamato da uno script. ( fa eseguire primo.exe)

```
OUT=`./primo.exe`  
echo "l'output del processo e' ${OUT}"
```

**Nota Bene:** L'apice giusto da usare è ``` quello che in alto tende a sinistra e in basso tende a destra. Viene chiamato **backticks** o **backquotes**.

Nelle tastiere americane si trova nel primo tasto in alto a sinistra, accoppiato e sotto alla tilde. Non è l'apostrofo italiano ' (detto single quote) , il tasto del backquotes nelle tastiere italiane non esiste. Usare tastiera americana.



**Sintassi alternativa per la command substitution :** `$( rigadicomando )`

esempio: `OUT= $( ./primo.exe )` *NB: sono parentesi TONDE*

In entrambe le sintassi, le eventuali wildcards `? *` `[]` vengono interpretate dalla shell e sostituite. **Per disabilitare (escape) la sostituzione delle wildcards, si circonda la riga di comando con dei caratteri opportuni (quotes).**

# Stream di I/O predefiniti dei processi (2): command substitution (2)

supponiamo

1. Che nella home directory dell'utente esista un file di nome `prova-6.txt` contenente una sola riga di testo "asdrubale non ragiona"
2. Che ci troviamo in una directory diversa dalla home directory.
3. Che in quella directory esista uno script `esempio_command_substitution.sh` contenente quello che segue

```
NUM=5
```

```
echo "output dello script: " `cat ~/$1-$({NUM} +1 ).txt | wc -l`
```

4. Infine supponiamo di eseguire quello script invocandolo con i seguenti argomenti:

```
./esempio_command_substitution.sh prova 5
```

L'output dello script sarà il seguente

output dello script: 1

Ciò dimostra che la bash, prima sostituisce in quest'ordine tilde, metacaratteri, variabili, e poi esegue la command substitution



# Quoting di stringhe: command substitution & escape

In una riga di comando, le eventuali wildcards `? *` `[]` variabili etc etc vengono interpretate dalla shell e sostituite. **Per disabilitare (escape) la sostituzione delle wildcards, si circonda la riga di comando con dei caratteri opportuni (quotes).**

Esempio di command substitution e sostituzione di wildcards e variabili

```
echo Dear ${USER} the files are a[[:digit:]]B and the date is: `date`
```

```
Dear vittorio the files are a1B a2B and the date is: Tue, Sep 15, 2015 12:34:52 PM
```

**" "** **Double quote** per escape wildcards e spazi (quoting parziale di stringhe)

Impedisce di usare il `;` come separatore di comandi,

**Impedisce la sostituzione di wildcards**

ma **permette** di sostituire le **variabili** con il loro contenuto

e **permette** l'esecuzione di **comandi** (command substitution).

```
echo " Dear ${USER} the files are a[[:digit:]]B and the date is: `date` "
```

```
Dear vittorio the files are a[[:digit:]]B and the date is: Tue, Sep 15, 2015 12:35:44 PM
```

**' '** **Single quote** escape wildcards, command substitution, variable substitution, spazi

Impedisce di usare il `;` come separatore di comandi,

Impedisce la sostituzione di wildcards

e Impedisce di sostituire le variabili con il loro contenuto

e Impedisce l'esecuzione di comandi (command substitution).

```
echo ' Dear ${USER} the files are a[[:digit:]]B and the date is: `date` '
```

```
Dear ${USER} the files are a[[:digit:]]B and the date is: `date`
```

# Quotes: command substitution & escape (2)

esempi

```
echo Dear ${USER} the date is: `date` ; echo sei in ritardo  
Dear vittorio the date is: Tue, Sep 15, 2015 12:58:21 PM  
sei in ritardo
```

```
echo " Dear ${USER} the date is: `date` ; echo sei in ritardo "  
Dear vittorio the date is: Tue, Sep 15, 2015 12:58:07 PM ; echo sei in ritardo
```

```
echo ' Dear ${USER} the date is: `date` ; echo sei in ritardo '  
    Dear ${USER} the date is: `date` ; echo sei in ritardo
```

# ATTENZIONE: la bash processa l'output prodotto dalla command substitution (1)

supponiamo che nella directory corrente esistano solo i due script seguenti,  
outputconasterisco.sh e chiama\_outputconasterisco.sh.

outputconasterisco.sh

```
echo '*'
```

chiama\_outputconasterisco.sh

```
./outputconasterisco.sh  
echo "riga di separazione"  
echo `./outputconasterisco.sh`
```

Eseguendo lo script `./chiama_outputconasterisco.sh` ottengo il seguente output

\*

riga di separazione

chiama\_outputconasterisco.sh outputconasterisco.sh

**Notare che nel comando contenente `echo `./outputconasterisco.sh`` l'asterisco buttato sullo standard output è stato interpretato dalla bash e sostituito con i nomi dei file presenti nella directory corrente.**

# Espressioni CONDIZIONALI (1)

- Le espressioni condizionali sono dei particolari comandi che valutano alcune condizioni e restituiscono un exit status di valore 0 per indicare la verità dell'espressione o di valore diverso da zero per indicarne la falsità.
- Ciascuna espressione condizionale si scrive mettendo le condizioni da valutare tra doppie parentesi quadre `[[ .... ]]`
- Le condizioni che possono essere inserite in una espressione condizionali sono condizioni create appositamente, e **NON POSSONO ESSERE COMANDI**.
- Le condizioni all'interno delle `[[ ]]` possono essere composte tra loro mediante degli operatori logici ! (not), && (and), || (or) e mediante parentesi tonde per stabilire l'ordine di valutazione. ad esempio `[[ ( condA)&&(condB) ) || condC ]]`
- All'interno delle espressioni condizionali `[[ ]]` la bash **effettua solo** le interpretazioni **variable expansion**, **arithmetic expansion con `$(( ))`** ma non `(( ))`, **command substitution**, process substitution, and quote removal, **ma solo negli operandi**.
- Sono invece **disabilite** le interpretazioni **Word splitting**, **brace expansion** e **pathname expansion** e **NON POSSONO COMPARIRE DEI COMANDI**
- Gli operatori di condizione (quelli con il - davanti, ad esempio `-e nomefile` ) per essere riconosciuti e valutati correttamente devono essere **NON quotati**. e non possono essere generati da command substitution.

Quindi:

- **non posso eseguire comandi ma posso eseguire command substitution (i backtick)** ma solo negli operandi, non per generare operatori.
- le doppie parentesi tonde senza \$ sono interpretate non come valutazione aritmetica ma come accorpamento logico di comandi per definire l'ordine di valutazione.

# Espressioni CONDIZIONALI (2)

## COMPORRE CONDIZIONI DENTRO ESPRESSIONI CONDIZIONALI

Notare che in una stessa espressione condizionale `[] []` io posso effettuare sia confronti aritmetici (**specificando opportuni operatori** `-eq -ne -le -lt -ge -gt`), ma anche confronti non aritmetici (specificando altri operatori).

Posso infatti verificare **condizioni non aritmetiche che riguardano stringhe**, usando gli **operatori di confronto lessicografico tra stringhe** `== != < <= >= >`

Posso anche verificare se delle stringhe sono vuote o no (con gli operatori `-z -n`).

Posso verificare **condizioni sui file** (operatori `-d -e -f -h -r -s -t -w -x -O -G -L`).

Posso **confrontare le date di ultima modifica di due files** (operatori `-nt -ot`).

Quindi, con le espressioni condizionali posso verificare espressioni aritmetiche ma **NON POSSO eseguire assegnamenti a variabili**.

NOTA BENE: **NEGLI OPERANDI** dei confronti aritmetici possono comparire valutazioni aritmetiche effettuate mediante `$(( ))` ma **NON POSSO USARE VALUTAZIONI ARITMETICHE COME CONDIZIONI ISOLATE** perché vengono fraintese.

SI `[] $NUM -lt $(( $VAR * ( 3 + $MUL ) )) []`

**NO** `[] $(( $VAR < 3 + $MUL )) []` non errore sintattico ma fraintende

**NO** `[] pippo []` **fraintende**, restituisce exit status 0

**NO** `[] 3 []` **fraintende**, restituisce exit status 0

**NO** `[] 0 []` **fraintende**, restituisce exit status 0

# Espressioni CONDIZIONALI (3)

## DIFFERENZE TRA ESPRESSIONI CONDIZIONALI E VALUTAZIONI ARITMETICHE

Gli operatori di valutazione aritmetica `(( ))` o `$(( ))` valutano aritmeticamente tutto il comando che specifico al loro interno, quindi **l'operatore di valutazione aritmetica `(( ))` o `$(( ))` :**

- ☐ **puo'** contenere solo espressioni valutabili aritmeticamente
- ☐ **può** comporre espressioni aritmetiche mediante operatori logici `&&` (AND) `||` (OR) e `!` (NOT).
- ☐ **non può** contenere espressioni condizionali
- ☐ **non può** contenere comandi

SI `(( $VAR * ( 3 + $MUL ) < 4 && 7 > $VAR ))`

NO `(( $VAR < 3 + $MUL && [[ $NUM -lt 5 ]] ))` errore sintattico

# Espressioni CONDIZIONALI (4)

Esempi di espressioni corrette o sbagliate

<code>[[ ls / ]]</code>	<b>syntax error</b>
<code>[[ \$(( "11" &gt; "2" )) ]]</code>	<b>Fraintende.</b> produce exit status 0 (true) sempre
<code>[[ (( "11" &gt; "2" )) ]]</code>	<b>Fraintende.</b> produce exit status 1 (false) perché le tonde parentesi non valutano aritmeticamente e la valutazione è lessicografica (primo 1 a sinistra precede primo 2 a destra).
<code>[[ -e /usr/include/stdio.h ]]</code>	OK. Esiste il file /usr/include/stdio.h ? true va bene
<code>[[ "-e /usr/include/stdio.h" ]]</code>	<b>Fraintende</b> , restituisce true ma per sbaglio, infatti...
<code>[[ "-e /usr/include/stdio" ]]</code>	<b>Fraintende</b> , restituisce true ma stdio non esiste
<code>[[ "-e" /usr/include/stdio.h ]]</code>	<b>syntax error</b>

Per effettuare confronti aritmetici, quindi, non posso usare la coppia di doppie parentesi tonde senza \$ ma devo utilizzare gli appositi operatori di confronto aritmetico, che sono **-eq**, **-ne**, **-lt**, **-le**, **-gt**, or **-ge**

<code>NUM=11; [[ \${NUM} &lt; 3 ]];</code>	<b>Sembra Strano</b> ma va bene. Exit status 0 (true) perché il confronto è eseguito lessicograficamente tra stringhe.
<code>NUM=11; [[ \${NUM} -le 3 ]];</code>	OK. ha exit status 1 (false) perché confronta aritmeticamente le stringhe.

Non posso annidare espressioni condizionali perché sono anch'esse dei comandi.

<code>[[ -e /usr/include/stdio.h &amp;&amp; [[ -e /usr/include/stdio.h ] ]]</code>	<b>syntax error</b>
--	---------------------

Non posso usare command substitution per generare operatori

<code>[[ `echo -e /usr/include/stdio.h` ]]</code>	<b>Fraintende</b>
<code>[[ -e `echo /usr/include/stdio.h` ]]</code>	OK

# Espressioni CONDIZIONALI (5) - versioni

*enhanced brackets*

**[[ *condiz* ]]**

*single brackets*

**[ *condiz* ]**

*test*

**test *condiz***

Esistono 3 operatori analoghi per le espressioni condizionali, al cui interno specificare gli operatori per Condizioni di file, Confronto tra stringhe, Confronto aritmetico. L'operatore **[[ ]]** è più recente e non esiste nelle bash molto vecchie.

L'operatore **[[ ]]** permette di **comporre tra loro piu' condizioni**, utilizzando gli stessi **operatori logici** usati in C ovvero **!** (negazione) **&&** (and) e **||** (or).

Inoltre, **all'interno del blocco [[ ]] posso usare parentesi tonde** per raggruppare espressioni e modificare la precedenza degli operatori e **posso andare a capo**.proseguendo l'espressione

Diversamente, i due operatori **[ ]** e **test** permettono di comporre tra loro piu' condizioni, utilizzando però **operatori logici diversi**, e precisamente **!** (negazione) **-a** (and) e **-o** (or).

Inoltre, con questi due operatori, **NON POSSO usare parentesi tonde** per raggruppare espressioni e modificare la precedenza degli operatori e **NON posso andare a capo** se non usando il **\** a fine riga .



# Espressioni CONDIZIONALI (6)

**Le espressioni condizionali** `[[ expression ]]` restituiscono un exit status 0 or 1 in dipendenza della valutazione della condizione esplicitata nella espressione. Tale risultato si ritrova come al solito nella variabile \$?

All'interno delle espressioni condizionali `[[ ]]`

- sono **disabilitate** le interpretazioni di tipo **Word splitting** e **pathname expansion**
- sono **abilitate** invece `variable expansion`, `arithmetic expansion` (solo quelle con `$()`) ma non con `(( ))` senza `$`), `command substitution`, `process substitution`, e `quote removal`.
- Gli operatori di condizione (quelli con il - davanti, ad esempio **-e** nomefile ) per essere riconosciuti e valutati correttamente devono essere **NON quotati** in alcun modo.

**Usando l'operatore doppia parentesi quadra** `[[ ]]` **le espressioni condizionali possono essere collegati con gli operatori logici usati anche in C, quali ! && || e raggruppati con parentesi tonde.**

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

NOTA BENE per le prossime pagine:

Se tra gli argomenti degli operatori compare un file allora:

- se per questo argomento viene specificato `/dev/fd/n` allora l'operatore controlla il file descriptor n.
- se per questo argomento viene specificato uno dei file `/dev/stdin` `/dev/stdout` `/dev/stderr` allora l'operatore controlla il file descriptor di valore 0 1 e 2.

# Espressioni CONDIZIONALI (7)

Alcuni operatori per verificare **condizioni su file** :

<b>-d</b> <u>file</u>	True if <u>file exists and is a directory.</u>
<b>-e</b> <u>file</u>	True if <u>file exists.</u>
<b>-f</b> <u>file</u>	True if <u>file exists and is a regular file.</u>
<b>-h</b> <u>file</u>	True if <u>file exists and is a symbolic link.</u>
<b>-r</b> <u>file</u>	True if <u>file exists and is readable.</u>
<b>-s</b> <u>file</u>	True if <u>file exists and has a size greater than zero.</u>
<b>-t</b> <u>fd</u>	True if file descriptor fd is open and refers to a terminal.
<b>-w</b> <u>file</u>	True if <u>file exists and is writable.</u>
<b>-x</b> <u>file</u>	True if <u>file exists and is executable.</u>
<b>-O</b> <u>file</u>	True if <u>file exists and is owned by the effective user id.</u>
<b>-G</b> <u>file</u>	True if <u>file exists and is owned by the effective group id.</u>
<b>-L</b> <u>file</u>	True if <u>file exists and is a symbolic link. (deprecated, see -h)</u>

file1 **-nt** file2            True if file1 is newer (according to modification date) 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.

Nota Bene: Ne esistono altre, per vedere quali guardare man bash nella parte intitolata  
CONDITIONAL EXPRESSIONS.

# Espressioni CONDIZIONALI (8)

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

**-o optname**      True if shell option optname is enabled. See the list of options under the description of the **-o** option to the `set` command.

## 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**      **OP** is one of **-eq**, **-ne**, **-lt**, **-le**, **-gt**, or **-ge**. These arithmetic binary operators return true if arg1 is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to arg2, respectively. Arg1 and arg2 may be positive or negative integers.

# Esempio d'uso di Espressioni CONDIZIONALI (1)

Due porzioni di codice bash che fanno la stessa cosa usando in un caso le espressioni condizionali ed una valutazione aritmetica come condizione dell'if, e nel secondo caso invece usando direttamente l'espressione condizionale come lista di comandi dell'if che ne valuta l'exit status :

```
[[ -e ${name} ]]
if (( $? == 0 )) ; then
    echo "esiste ${name}, lo elimino" ;
    rm -f ${name}
fi ;
```

analogamente

```
if [[ -e ${name} ]] ; then
    echo "esiste ${name}, lo elimino" ;
    rm -f ${name}
fi ;
```

# Esempio d'uso di Espressioni CONDIZIONALI (1b)

- 1)      `var="prova"`  
      `[[ -n ${var} ]] ; echo $?`                      **output    0**
- 2)      `var=""`  
      `[[ -n ${var} ]] ; echo $?`                      **output    1**
- 3)      `[[ -n "" ]] ; echo $?`                      **output    1**
- 4)      `unset var`  
      `[[ -n ${var} ]] ; echo $?`                      **output    1**
- 5)      `[[ -n ]] ; echo $?`  
          **output**  
          **bash: unexpected argument `]]' to conditional unary operator**  
          **bash: syntax error near `]]'0**

**NOTARE LA DIFFERENZA TRA I CASI 4 e 5\*:**

**nel caso 4 la bash capisce che si tratta di espr condizionale, riconosce l'operatore -n e, prima di tentare di fare l'espansione della variabile var, correttamente considera 0 la lunghezza della variabile che non esiste.**

**Nel caso 5, invece, la bash dice che manca l'argomento.**

# Esempio d'uso di Espressioni CONDIZIONALI (2)

Due porzioni di codice bash che fanno la stessa cosa usando due diversi tipi di espressioni condizionali: le single brackets **[ ]** e le extended brackets **[[ ]]**

```
if [ -d ${name} -a ${#name} -lt 10 ] ; then
    echo "ok ${name}" ;
else
    echo "no ${name}";
fi ;
```

Analogamente

```
if [[ -d ${name} && ${#name} -lt 10 ]] ; then
    echo "ok ${name}" ;
else
    echo "no ${name}";
fi ;
```

# Attenzione all'interpretazione di && può essere

operatore AND logico in Espressioni CONDIZIONALI  
o operatore in SEQUENZE di COMANDI condizionali

## QUI && è AND LOGICO

```
if [[ -e ${name} && ${#name} -lt 10 ]] ; then  
    echo "ok ${name}" ;  
fi
```

## QUI && è operatore in sequenza di comandi condizionale

```
if [[ -e prova.c ]] && gcc -o prova.exe prova.c ; then  
    echo "posso eseguire prova.exe"  
fi
```

# Evitare Errori Più Comuni (1)

**METTETE SPAZI PRIMA E DOPO le `[[` e le `]]`  
ed anche PRIMA E DOPO gli operatori (es. `-`)**

Ad esempio, l'espressione condizionale seguente

```
if [[-d /usr/include/stdio.h ]] ; then echo esiste ; fi
```

provoca errore, perché la bash crede che `[[`-d sia un comando da eseguire, infatti avvisa:

**`[[`-d: command not found**

perché MANCA LO SPAZIO TRA le `[[` e il `-d`

**Infatti, il parser della bash riconosce le espressioni condizionali prima separando la riga in parole delimitate da spazi. Se mancano gli spazi le espressioni condizionali non vengono riconosciute.**

Analogamente per gli spazi prima e dopo qualunque operatore, aritmetico, lessicografico, su file, insomma tutti.

Il comando

```
if [[ -dFILECHENONESISTE ]] ; then echo esiste ; fi
```

esegue e mette in output esiste.

Sembra dire che FILECHENONESISTE esiste perché senza spazi viene valutata (E NON ESEGUITA) la stringa -dFILECHENON ESISTE che essendo non vuota restituisce vero.



# Evitare Errori Più Comuni (2)

## Cosa posso mettere dentro un if ?

### Liste di Comandi o Espressioni condizionali

OK	if ls prova ; then echo ciao ; else echo fanculo ; fi
OK, ma strano	if `ls prova` ; then echo ciao ; else echo fanculo ; fi
OK	if [[ -e prova ]] ; then echo ciao ; else echo fanculo ; fi

## Cosa posso mettere dentro le [[ ]] ?

solo gli operatori delle espressioni condizionali,

NON dei comandi,

se non come command substitution negli operandi

OK	if [[ "aaa" == `cat prova` ]] ; then echo ciao ; else echo fanculo ; fi
OK	if [[ -e prova ]] ; then echo ciao ; else echo fanculo ; fi
<b>NOOO</b>	if [[ ls prova ]] ; then echo ciao ; else echo fanculo ; fi
OK	if [[ -e `ls pro?a.c` ]] ; then echo esiste prova.c ; fi

# Evitare Errori Più Comuni (3)

## NO parentesi tonde tra le [ ] semplici

OK `if [[ ( $var ne 13 || $var -ne 17 ) && -d /tmp ]] then echo ciao ; fi`  
**Syntax error! NO!** `if [ ( $var ne 13 -o $var -ne 17 ) -a -d /tmp ] then echo ciao; fi`

## SI composizioni logiche dentro a valutazioni aritmetiche (( ))

OK `if (( $var < 13 || $var > 17 )) ; then echo ciao ; fi`

## SI composizioni logiche dentro a espressioni condizionali [[ ]]

ma con operatori -a -o senza tonde () in []

OK `if [[ $var -lt 13 || $var > 17 ]] then echo ciao ; fi`

# Espressioni CONDIZIONALI (9)

NOTA BENE:

**RIBADISCO CHE NON POSSO eseguire dei COMANDI all'interno dell'operatore espressione condizionale. Posso mettere solo delle espressioni con operatori di confronto o di condizioni su file oppure delle command substitution.**

Cioe' **non posso eseguire** un comando (ad esempio come quello che segue) perche' il comando specificato tra le parentesi quadre semplicemente **NON VIENE ESEGUITO** poiche' **all'interno delle espressioni condizionali non e' previsto che venga eseguito un comando**. Diverso è il caso delle command substitution.

**NO !!!!!!!!!!!!! ERRORE !!!!!!!**

```
if [[ ls nomefile.txt ]] ; then echo "il file esiste" ; else echo "il file non esiste" ; fi
```

**Se occorre eseguire un comando e valutarne il risultato, NON DEVO METTERE il comando all'interno di una espressione condizionale ma devo metterlo semplicemente come condizione di un if, così:**

**SI**

```
if ls nomefile.txt ; then echo "il file esiste" ; else echo "il file non esiste" ; fi
```

# Quoting '\$stringa'

## Usare caratteri non stampabili in una stringa

Parole aventi forma `'$charsequence'` sono trattate in modo speciale.

La sequenza `charsequence` puo' contenere backslash-escaped characters.

1) La parola viene espansa in una stringa single-quoted, cioe' **perde il \$ all'inizio ma mantiene i due ' all'inizio e alla fine** (come se \$ non esistesse).

2) le backslash-escaped characters, se presenti, sono sostituite come specificato nello standard ANSI C:

`\a` **alert (bell)**

`\e`

`\f` **form feed**

`\r` **carriage return**

`\v` **vertical tab**

`\'` **single quote**

`\b` **backspace**

`\E` **an escape character**

`\n` **new line**

`\t` **horizontal tab**

`\\` **backslash**

`\"` **double quote**

`\nnn` the eight-bit character whose value is the octal value `nnn` (one to three digits)

`\xHH` the eight-bit character whose value is the hexadecimal value `HH`

(one or two hex digits)

`\cx` a control-x character, as if the dollar sign had not been present.

Sta cosa tornera' utile per specificare il contenuto della variabile **IFS** poiche' questa contiene anche caratteri non stampabili.

# Word Splitting: CARATTERI SEPARATORI IN ELENCHI

La variabile IFS contiene i caratteri che fungono da separatori delle parole negli elenchi, IFS='\$' '\t\n'

Notare che IFS di default contiene uno spazio bianco, un tab e un newline (a capo).

Se devo lanciare dei comandi in cui devo trattare dei nomi di file che contengono degli spazi bianchi, se non posso fare diversamente devo i) usare degli elenchi separati da newline o tab, e ii) togliere dai separatori lo spazio bianco.

IFS='\$' '\t\n'

Poi eseguirò il comando che dovevo lanciare e dopo rimetterò lo IFS come era prima.

Es: directory che contiene due files, "aa bb.txt " e "aa cc.txt"

Vedere che succede se lancio

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

Visualizzo

```
aa
bb.txt
aa
cc.txt
```

TRUCCO OSCENO MA FUNZIONA

```
OLDIFS=${IFS}
```

```
IFS='$'\t\n'
```

```
for name in `ls -1 aa*` ; do echo ${name} ; done
```

```
IFS=${OLDIFS}
```

# Word Splitting: CARATTERI SEPARATORI IN ELENCHI (2)

## Precisazione per TarloI (sconosciuto)

perche', se tolgo spazio da IFS, bash riconosce grammatica?

perché i separatori contenuti in IFS sono utilizzati per individuare in modo specifico le separazioni tra gli elenchi di nomi.

I separatori usati per individuare le separazioni tra parole chiave del linguaggio sono invece non modificabili e sono sempre gli spazi bianchi, i tab e le andate a capo

# read - Lettura da standard input (tastiera o file) (1)

Uno script può leggere dallo standard input delle sequenze di caratteri usando un comando chiamato **read**.

Il comando `read` riceve la sequenza di caratteri digitate da tastiera fino alla pressione del tasto INVIO (RETURN) e mette i caratteri ricevuti in una variabile che viene passata come argomento alla `read` stessa. Se invece lo standard input è stato ridiretto da un file, allora la `read` legge una riga di quel file ed una eventuale `read` successiva legge la riga successiva.

La `read` restituisce un risultato che indica se la lettura è andata a buon fine, cioè restituisce:

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

```
while true ; do
    read RIGA
    if (( "$?" != 0 ))
    then
        echo "eof reached "
        break
    else
        echo "read \"${RIGA}\" "
    fi
done
```

va bene anche `while (( 1 )) ;`

## read - Precisazione su raggiungimento di fine file (2)

Può accadere che la lettura incontri la fine del file senza incontrare l'andata a capo.  
Capita se leggo da file e nel file l'ultima riga **non** ha l'andata a capo `\n` alla fine.

In tal caso la `read` mette, nella variabile che gli viene passata, tutti i caratteri non ancora letti che precedono la fine del file, e poi restituisce un valore  $>0$  per indicare che il file è stato usato fino alla fine.

Quindi, quando la `read` dice che è arrivata alla fine del file di input, possono essere accaduti due eventi diversi:

- 1) la `read` ha incontrato subito la fine del file e quindi nella variabile non è stato messo nulla, e quindi nella variabile la `read` mette la stringa vuota `""`.
- 2) la `read` ha letto dei caratteri e poi ha incontrato la fine del file, e quindi nella variabile troverò i caratteri letti ma la `read` restituisce comunque un valore  $>0$ .

**Perciò, quando si fa una lettura con la `read`, se la `read` dice di essere arrivata a fine file occorre comunque controllare se nella variabile letta c'è qualcosa dentro.**

Vedere la pagina successiva per esempi di controllo.



# read - Precisazione su raggiungimento di fine file (3)

## procedure di lettura corrette e tra loro equivalenti

Occorre una nozione aggiuntiva: la stringa **`${#VAR}`** viene interpretata dalla bash sostituendola con la stringa di cifre che rappresenta il numero di caratteri di cui la variabile **VAR**, se esiste, è formata (la lunghezza della variabile **VAR**).

Se la variabile non esiste allora la stringa viene sostituita dalla stringa vuota.

Esempio: `VAR="ciao"; echo ${#VAR};` produce in output `4`

**Quando si fa una lettura con la read, se la read dice di essere arrivata a fine file occorre comunque controllare se nella variabile letta c'è qualcosa dentro.**

Il controlli seguenti sono equivalenti:

- accettano in input anche righe vuote (riga con la sola andata a capo) che lasciano la variabile **RIGA** vuota nel caso che non si sia ancora raggiunta la fine del file.

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

OR Logico dentro espressione condizionale

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

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

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

Sequenza di comandi condizionale, prosegue se exit status != 0

# read - Lettura da standard input (tastiera o file) (4)

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

- Se la riga letta contiene più parole del numero delle variabili passate alla read, allora ciascuna variabile viene riempita con una parola estratta dalla linea letta, tranne l'ultima variabile che riceve tutto quello che resta della linea.
- Se invece la riga letta contiene meno parole del numero di variabili passate alla read, allora solo le prime variabili ricevono una parola, alle altre è assegnato il valore vuoto. Esistono alcune opzioni interessanti **-u -N -r** ed altre meno necessarie **-e -n -t**. Il risultato restituito da read è sempre 0 tranne che in caso di eof (o fd non valido o timeout scaduto, se specificati).

Se eseguo il comando `read varA varB varC`

e scrivo a tastiera la frase `prima seconda terza`  
le variabili assumono valore `varA="prima" varB="seconda" varC="terza"`

se invece scrivo a tastiera la frase `prima seconda terza quarta`  
le variabili assumono valore `varA="prima" varB="seconda" varC="terza quarta"`

se invece scrivo a tastiera la frase `prima seconda`  
le variabili assumono valore `varA="prima" varB="seconda" varC=""`

## read - Precisazione su lettura di riga con spazi iniziali (5)

Può accadere che una riga digitata da tastiera oppure anche una riga di un file, abbiano all'inizio degli spazi bianchi oppure delle tabulazioni.

Ad esempio, un file denominato `file_con_spazio_iniziale_nella_riga.txt` potrebbe contenere una riga così fatta: "**K 23F G2**"

In tal caso, una `read` (a cui viene passata UNA variabile) che legge quella riga NON mette nella variabile di lettura gli spazi iniziali ma solo i caratteri a partire dal primo carattere non bianco.

Perciò, una lettura fatta come qui sotto indicato

```
read RIGA < file_con_spazio_iniziale_nella_riga.txt
echo "${RIGA}"
```

causerebbe l'output seguente:

```
" K 23F G2"
```

in cui, evidentemente, non sono stati letti i caratteri bianchi e le tabulazioni iniziali.

Ciò è dovuto al fatto che la `read` tenta di leggere le parole in una riga, non la riga.

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"
```

# read - Lettura fino a un numero massimo di caratteri (6)

L'opzione **-n** della **read** **permette di specificare il numero massimo di caratteri che devono essere letti.**

Esempio: Legge **fino a 4 caratteri** dallo standard input e li mette nella variabile **STRINGA**  
**read -n 4 STRINGA**

Notare che non vengono più separate le word, anche gli spazi e i tab sono considerati caratteri qualsiasi.

Invece l'andata a capo **\n** viene ancora considerata un terminatore che interrompe la read.

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

Se una read chiede K caratteri, alla fine della digitazione dei K caratteri richiesti la read termina anche se non è stato premuto l'invio (return) cioè anche se non è stata raggiunta la fine riga.

Che fine fanno i caratteri residui?

**Ciascuna read comincia la lettura proprio dai caratteri residui (cioè non letti) dalla read che l'ha preceduta.**

## Esempi di comandi e i loro output

```
vic@vic:~$ cat mioinput.txt
messaggio1
messaggio2
vic@vic:~$ exec 103< mioinput.txt
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o1
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o2
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $?
1
vic@vic:~$ echo $RIGA

vic@vic:~$ exec 103<&-
vic@vic:~$
```

# read - Lettura di un numero specificato di caratteri (7)

L'opzione **-N** della `read` permette di specificare il numero esatto di caratteri che devono essere letti.

Esempio: Legge 4 caratteri dallo standard input e li mette nella variabile `STRINGA`

```
read -N 4 STRINGA
```

Notare che non vengono più separate le word, anche gli spazi e i tab sono considerati caratteri qualsiasi.

L'andata a capo `\n` NON viene considerata un terminatore che interrompe la `read` bensì un carattere letto.

**Se durante la lettura viene raggiunta la fine riga, la `read` non termina e aspetta altri caratteri per raggiungere il numero di caratteri richiesto e mette anche il `\n` nella variabile**

Se una `read` chiede meno caratteri di quelli presenti nella riga (del file o digitata dall'utente), che fine fanno i caratteri residui?

**Ciascuna `read` comincia la lettura proprio dai caratteri residui (cioè non letti) dalla `read` che l'ha preceduta.**

Vedere l'esempio qui di lato e commentarlo.

Notare la terza lettura che legge l'andata a capo

## Esempi di comandi e i loro output

```
vic@vic:~$ cat mioinput.txt
messaggio1
messaggio2
vic@vic:~$ exec 103< mioinput.txt
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o1 m
vic@vic:~$ printf "%q" "${RIGA}"
$'o1\nm'
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
essa
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
ggio
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $?
1
vic@vic:~$ echo $RIGA
2
vic@vic:~$ exec 103<&-
vic@vic:~$
```

# read – confronto tra opzioni -n e -N (7)

## Esempi di comandi e loro output

```
vic@vic:~$ cat mioinput.txt
messaggio1
messaggio2
vic@vic:~$ exec 103< mioinput.txt
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o1
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o2
vic@vic:~$ read -n 4 -u 103 RIGA
vic@vic:~$ echo $?
1
vic@vic:~$ echo $RIGA

vic@vic:~$ exec 103<&-
vic@vic:~$
```

```
vic@vic:~$ cat mioinput.txt
messaggio1
messaggio2
vic@vic:~$ exec 103< mioinput.txt
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
mess
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
aggi
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
o1 m
vic@vic:~$ printf "%q" "${RIGA}"
$'o1\nm'
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
essa
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
ggio
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $RIGA
2
vic@vic:~$ read -N 4 -u 103 RIGA
vic@vic:~$ echo $?
1
vic@vic:~$ echo $RIGA

vic@vic:~$ exec 103<&-
vic@vic:~$
```

# Stream di I/O Non Predefiniti (1)

## Apertura, File Descriptor, accesso

Uno script bash può avere necessità di usare dei file su disco per fare I/O anche se non vengono passati mediante stdin ed stdout.

Ricordiamo che ad stdin, stdout e stderr sono associati dei file descriptor (rispettivamente 0, 1, 2) che permettono di accedere a quegli stream.

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

All'apertura del file, l'utente può decidere il file descriptor (il numero) che rappresenterà il file aperto, ma se tale fd è già usato avvengono problemi. In alternativa (procedura vivamente consigliata) l'utente può chiedere al sistema operativo di scegliere un fd libero.

Gli operatori sono indicati nella seconda e terza colonna della seguente tabella:

Modo Apertura	Utente sceglie fd (n è il numero scelto dall'utente)	Sistema sceglie fd libero e lo inserisce in variabile
Solo Lettura	<b>exec</b> n< PercorsoFile	<b>exec</b> {NomeVar}< PercorsoFile
Scrittura	<b>exec</b> n> PercorsoFile	<b>exec</b> {NomeVar}> PercorsoFile
Aggiunta in coda	<b>exec</b> n>> PercorsoFile	<b>exec</b> {NomeVar}>> PercorsoFile
Lettura e Scrittura	<b>exec</b> n<> PercorsoFile	<b>exec</b> {NomeVar}<> PercorsoFile

NB: i simboli < > >> <> devono essere attaccati (**senza spazi**) al numero o alla }

# Stream di I/O Non Predefiniti (2)

## Esempi: Apertura di file in Lettura e in Scrittura

uso una variabile che chiamo FD in cui verrà messo il file descriptor del file aperto. Nel comando **read** specifico l'opzione **-u** (seguita dal file descriptor del file aperto) per indicare al comando read da quale file aperto deve essere effettuata la lettura.

*# esempio:*

*# effettuo le letture dal file mioinput.txt aprendolo in lettura*

```
exec {FD}< /home/vittorio/mioinput.txt
while read -u ${FD} StringaLetta ;
do
    echo "ho letto: ${StringaLetta}"
done
```

*# esempio:*

*# scrivo l'output dei comandi echo sul file miooutput.txt aprendolo in scrittura*

```
exec {FD}> /home/vittorio/miooutput.txt
for name in pippo pippa pippi ; do
    echo "inserisco ${name}" 1>&${FD}
done
```



# Stream di I/O Predefiniti e Non (1)

## Dove trovo i file descriptor aperti da una bash?

Suppongo di avere una bash interattiva aperta.

La variabile \$\$ mi dice il PID process identifier della shell corrente.

Supponiamo che il PID della mia shell sia 1231.

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

Quindi il seguente comando visualizza il contenuto della directory corrispondente alla shell corrente

```
ls /proc/$$/
```

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

Guarda caso, trovo sempre (tranne casi speciali) i file aperti aventi nome 0 1 2

Se nella mia bash apro un altro file, ad esempio col comando

```
exec {FD}< /tmp/caz.txt
```

e scopro che il file aperto ha file descriptor 7

```
echo ${FD}
```

```
7
```

se guardo nella directory /proc/1231/fd/ vedo che è stato aggiunto un file speciale di nome 7.

```
ls /proc/$$/fd/
```

# Stream di I/O Predefiniti e Non (2)

## Chiusura di file mediante il suo file descriptor

**Qualunque sia il modo di apertura** con cui ho aperto un file (lettura scrittura o entrambi), la chiusura di un file puo' essere effettuata utilizzando il comando `exec` con il seguente strano operatore

```
exec n>&-
```

dove `n` e' il file descriptor del file da chiudere.

Analogamente, se la variabile `FD` contiene il valore del file descriptor da chiudere, posso chiudere quel file utilizzando la seguente strana sintassi:

```
exec {FD}>&-
```

Nota bene, se dopo la chiusura del file utilizzo il file descriptor, la `bash` produce un errore.

```
exec 103> /home/vittorio/mioinput.txt
```

```
echo "messaggio1" 1>&103
```

```
# chiudo
```

```
exec 103>&-
```

```
echo "messaggio2" 1>&103
```

```
bash: 103: Bad file descriptor
```

← produce un messaggio di errore

# Ridirezionamenti di Stream di I/O (0)

## 1) RIDIREZIONAMENTO DI FILE DESCRIPTOR DI PROCESSI FIGLI

Quando lanciamo un comando o processo o script all'interno di una bash il processo figlio ottiene una copia di tutti i file descriptor del padre, quindi lavora con gli stessi stream aperti del padre.

Però il padre, nel momento in cui lancia il processo figlio, può decidere di cambiare gli stream da far usare al figlio, associando, a ciascun file descriptor da passare al figlio, uno stream diverso.

In tal caso, i file descriptor passati al figlio avranno lo stesso identificatore numerico di quelli del padre, ma saranno associati a stream diversi.

Il padre continuerà ad usare i suoi file descriptor associati ai vecchi stream.

## 2) AUTO-RIDIREZIONAMENTO , ovvero RIDIREZIONAMENTO DI PROPRI FILE DESCRIPTOR

Un processo può decidere di aprire nuovi file, chiudere file aperti o di associare un proprio file descriptor ad un altro stream (ridirezionamento).

In quest'ultimo caso, i file descriptor ridirezionati mantengono il loro valore ma sono associati a stream diversi. Da quel momento in avanti, il processo usando i vecchi file descriptor accederà ai nuovi stream.

Apertura chiusura e ridirezionamento di propri file sono effettuati mediante il comando `exec`

# Ridirezionamenti di Stream di I/O (1)

## Ridirezionamenti:

- < ricevere input da file.
- > mandare std output verso file eliminando il vecchio contenuto del file
- >> mandare std output verso file aggiungendolo al vecchio contenuto del file
- | ridirigere output di un programma nell' input di un altro programma

## Ricevere input da file.

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

```
program < nome_file_input
```

**il programma vedrà il contenuto del file nome\_file\_input come se venisse digitato da tastiera.**

## Emulare la fine del file di input da tastiera: **Ctrl+D**

Notare che usando il ridirezionamento dell'input, c'è un momento in cui tutto il contenuto del file di input è stato passato al programma ed il programma si accorge di essere arrivato alla fine del file.

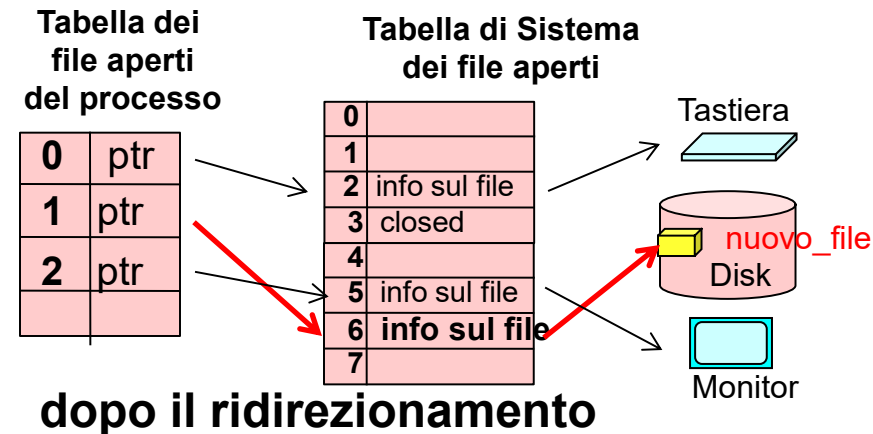
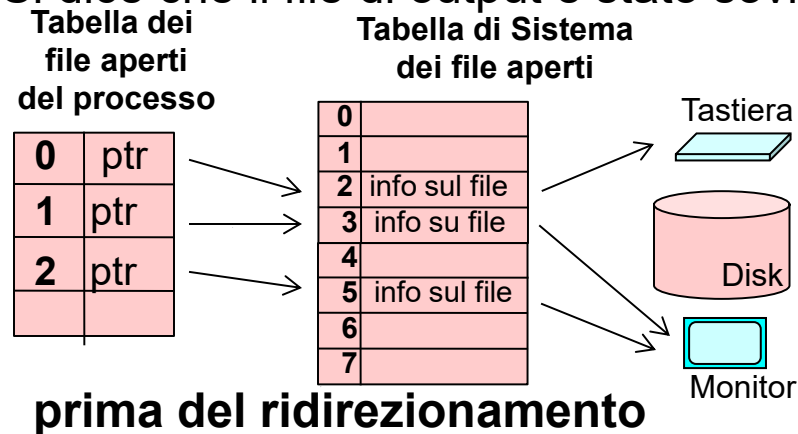
Se invece non faccio il ridirezionamento dell'input e fornisco l'input da tastiera, non incontro mai una fine del file di input. Per produrre da tastiera lo stesso effetto della terminazione del file di input devo digitare contemporaneamente i tasti CTRL e D che inviano un carattere speciale che indica la fine del file.

# Ridirezionamenti di Stream di I/O (2)

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

program > nome\_file\_output

Nel file nome\_file\_output troveremo i caratteriche il programma voleva mandare a video. Il precedente contenuto del file verrà perso ed **alla fine dell'esecuzione del programma nel file troveremo solo l'output generato dal programma stesso.** Si dice che il file di output è stato sovrascritto.



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

program >> nome\_file\_output

# Ridirezionamenti di Stream di I/O (3)

Supponiamo che un programma *program* mandi in output le due seguenti righe:

```
pippo  
pappa
```

Supponiamo che il file `nome_file_output` contenga queste tre righe:

```
uno  
due  
tre
```

Se eseguiamo il programma ordinando, con il ridirezionamento, di aggiungere l'output in coda al file `nome_file_output`, così:

```
program >> nome_file_output
```

alla fine dell'esecuzione il contenuto del file `nome_file_output` sarà:

```
uno  
due  
tre  
pippo  
pappa
```

Se eseguiamo il programma ridirezionando l'output sul file `nome_file_output`, così:

```
program > nome_file_output
```

alla fine dell'esecuzione il contenuto del file `nome_file_output` sarà solo:

```
pippo  
pappa
```

# Ridirezionamenti di Stream di I/O (4)

I **due ridirezionamenti (input ed output)** possono essere fatti **contemporaneamente**

```
program < nome_file_input > nome_file_output
```

O analogamente

```
program > nome_file_output < nome_file_input
```

In questo modo, il contenuto del `nome_file_input` viene usato come se fosse l'input da tastiera per il programma `program`, e l'output del programma viene scritto nel file `nome_file_output`.

Inoltre, in bash si può **ridirezionare assieme standard output e standard error su uno stesso file**, sovrascrivendo il vecchio contenuto:

```
program &> nome_file_error_and_output
```

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

```
program 2> nome_file_error > nome_file_output
```

# Ridirezionamenti di Stream di I/O (5)

Si noti che **i ridirezionamenti non modificano il valore dei file descriptor che vengono ridiretti** e nemmeno il numero di file descriptor dello script che li effettua. Questo accade perché il sistema operativo usa proprio il valore intero del file descriptor da ridirigere per indicare il file su cui viene ridiretto.

Ad esempio, se una bash lancia uno script `fd_di_script.sh` ridirezionando entrambi gli stream di output predefiniti su uno stesso file, quello script continua a vedere i due file descriptor 1 e 2 anche se questi fanno scrivere su uno stesso file.

ad esempio, lo script `fd_di_script.sh` visualizza l'elenco dei suoi file descriptor, così:

```
#!/bin/bash  
ls /proc/$BASHPID/fd/
```

Eseguiamo lo script ridirezionando entrambi gli stream di output predefiniti su uno stesso file:

```
./fd_di_script.sh &> out.txt
```

Nel file `out.txt` vedremo che sono indicati i tre file descriptor 0 1 2 come sempre.



# Ridirezionamenti di Stream di I/O (6)

## Generalizzazione operatori > <

In generale,

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

### **N>** NomeFileTarget

ridireziona il file descriptor N sul file **con nome** NomeFileTarget  
come già visto, se si omette N si intende standard output.  
usare >> per append

### **<N** NomeFileSource

ridireziona il file con nome NomeFileSource sul file descriptor N del  
programma specificato alla sinistra dell'operatore.  
Come già visto, se si omette N si intende standard input.

# Ridirezionamenti di Stream di I/O (7)

## PIPE | Connettere due processi mediante Ridirezione di stdout su stdin

E' possibile far eseguire contemporaneamente due processi mandando lo standard output del primo nello standard input del secondo, mediante l'operatore pipe |.

```
program1 | program2
```

I due processi eseguono in contemporanea.

Quando il primo processo termina o chiude il suo standard output, allora il secondo processo vede chiudersi il proprio standard input.

I due processi sono entrambi figli della bash in cui è invocata la riga di comando con la pipe.

E' possibile connettere più processi in una sequenza di pipe

```
program1 | program2 | program3
```

I tre processi eseguono in contemporanea e sono tutti figli della bash in cui è invocata la riga di comando con le pipe.

Verificare la fratellanza dei processi con il comando: `sleep 4 | sleep 5 | (sleep 2; ps al )`

# Ridirezionamenti di Stream di I/O (8)

## nuova bash per eseguire comandi composti in PIPE

Attenzione! In uno script dove compare una pipe, se il comando a destra o a sinistra della pipe è un comando composto (ad es. un loop) oppure è uno script, **ALLORA QUEL COMANDO ESEGUE IN UNA SHELL BASH FIGLIA.**

Guardiamo il seguente script **pipe.sh**

```
MIAVAR="iniziale"
echo prima ${MIAVAR}
ps
echo "poffarre" | while read MIAVAR ; do ps; echo dentro ${MIAVAR}; done
echo dopo ${MIAVAR}
```

eseguendo lo script pipe.sh, ottengo il seguente output

**prima** iniziale

PID	TTY	TIME	CMD
-----	-----	------	-----

1830	pts/0	00:00:00	bash
------	-------	----------	------

3990	pts/0	00:00:00	bash
------	-------	----------	------

3991	pts/0	00:00:00	ps
------	-------	----------	----

PID	TTY	TIME	CMD
-----	-----	------	-----

1830	pts/0	00:00:00	bash
------	-------	----------	------

3990	pts/0	00:00:00	bash
------	-------	----------	------

3993	pts/0	00:00:00	bash
------	-------	----------	------

3994	pts/0	00:00:00	ps
------	-------	----------	----

**dentro poffarre** la variabile MIAVAR creata in terza bash ha assunto valore "poffarre"

**dopo iniziale** MIAVAR ha mantenuto valore originale, non ha contenuto "poffarre" 157

<- bash di shell interattiva

<- bash che esegue pipe.sh

<- bash di shell interattiva

<- bash che esegue pipe.sh

<- bash che esegue while

# Ridirezionamenti di Stream di I/O (9)

## Operatore `>&` per ridirezionamento tra file descriptor

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

Dopo di questo, ciò che scrivo su N viene scritto su M.  
Lo stream N mantiene il valore N del suo file descriptor  
ma, nella tabella dei file aperti del processo,  
il suo puntatore alla tabella di sistema viene sostituito  
da una copia del puntatore dello stream M

Esisteranno da quel momento due file descriptor N ed M  
di valore diverso ma che puntano allo stesso file aperto.

Tabella dei  
file aperti  
del processo

0	ptr
N	ptr
M	ptr

Tabella di Sistema  
dei file aperti

0	
1	info sui file
2	info sul file
3	
4	
5	info sul file
6	
7	

prima del ridirezionamento

dopo il ridirezionamento

0	ptr
N	ptr
M	ptr

0	
1	info sui file
2	info sul file
3	
4	
5	info sul file
6	
7	

`ls pippo.txt 2>&1`

ridireziona lo stderr di ls sullo stdout di ls

# Ridirezionamenti di Stream di I/O (10)

## Attenti alla differenza tra `>` e `>&`

Attenzione a non confondersi

```
cat out.txt 2>1
```

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

```
cat out.txt 2>&1
```

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

# Ridirezionamenti di Stream di I/O (11)

## Ordine dei Ridirezionamenti con file descriptor (1)

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

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

### Il primo ridirezionamento 2> error.txt

apre il file error.txt e mette le info sul file aperto nella nuova posizione 6 della tabella di sistema.

Poi copia nello spazio occupato dal puntatore del file descriptor 2 l'indirizzo della riga nuova della tabella di sistema.

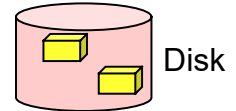
Quindi ora fd 2 punta al file error.txt mentre fd 1 scrive ancora su video

Tabella dei file aperti del processo

0	ptr
1	ptr
2	ptr

Tabella di Sistema dei file aperti

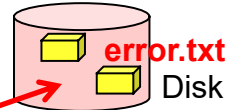
0	
1	
2	info sul file
3	info su file
4	
5	info sul file
6	
7	



**prima del ridirezionamento**

0	ptr
1	ptr
2	ptr

0	
1	
2	info sul file
3	info su file
4	
5	info su file
6	info su file
7	



**dopo il primo pezzo di ridirezionamento**

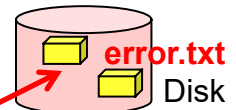
### Il secondo ridirezionamento 1>&2

copia il puntatore del file descriptor 2 nello spazio occupato dal puntatore del file descriptor 1

Quindi ora sia fd 1 che fd 2 puntano alla stessa riga della tabella di sistema ed entrambi scrivono sul file error.txt

0	ptr
1	ptr
2	ptr

0	
1	
2	info sul file
3	closed
4	
5	info su file
6	info su file
7	



**dopo il secondo pezzo di ridirezionamento**

# Ridirezionamenti di Stream di I/O (12)

## Ordine dei Ridirezionamenti con file descriptor (2)

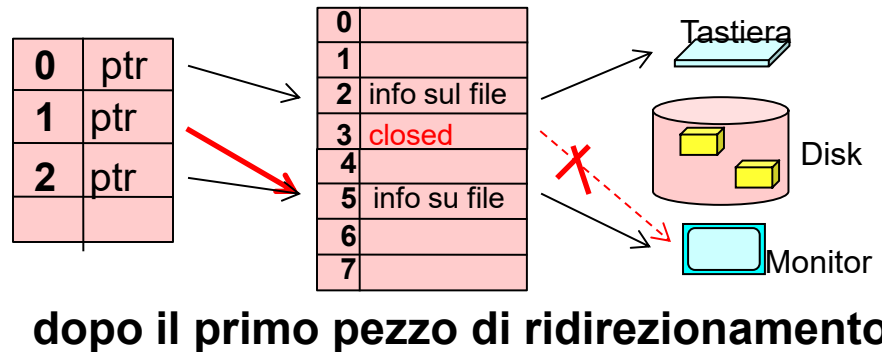
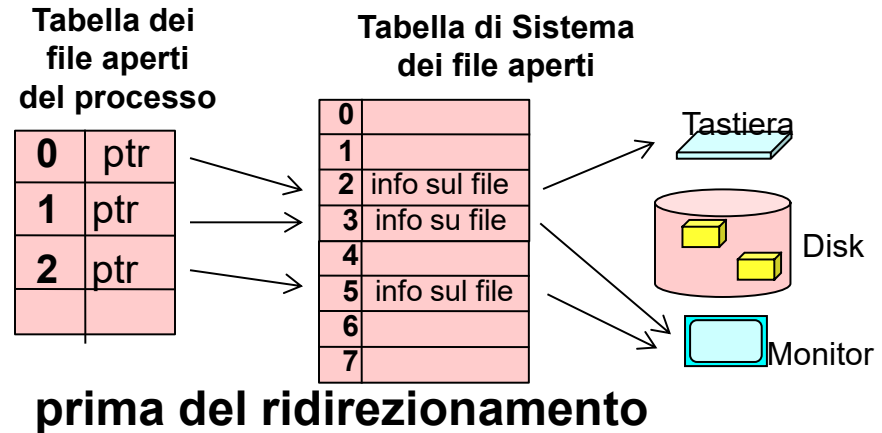
Scambiamo l'ordine degli operatori di ridirezionamento dell'esempio precedente.

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

### Il primo ridirezionamento 1>&2

copia il puntatore del file descriptor 2 nello spazio occupato dal puntatore del file descriptor 1

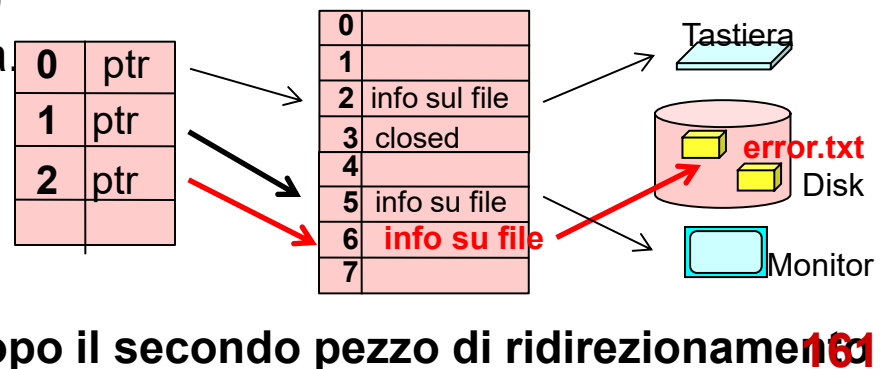
Quindi ora sia fd 1 che fd 2 puntano alla stessa riga della tabella di sistema ed entrambi scrivono su video



### Il secondo ridirezionamento 2>error.txt

apre il file error.txt e mette le info sul file aperto nella nuova posizione 6 della tabella di sistema. Poi copia nello spazio occupato dal puntatore del file descriptor 2 l'indirizzo della riga nuova della tabella di sistema.

Quindi ora fd 1 scrive su video mentre fd 2 scrive sul file error.txt



# Ridirezionamenti di Stream di I/O (13)

## Ordine dei Ridirezionamenti con file descriptor (3)

Confrontiamo i risultati dei due precedenti ridirezionamenti in cui cambia solo l'ordine. Si esegue da sinistra a destra.

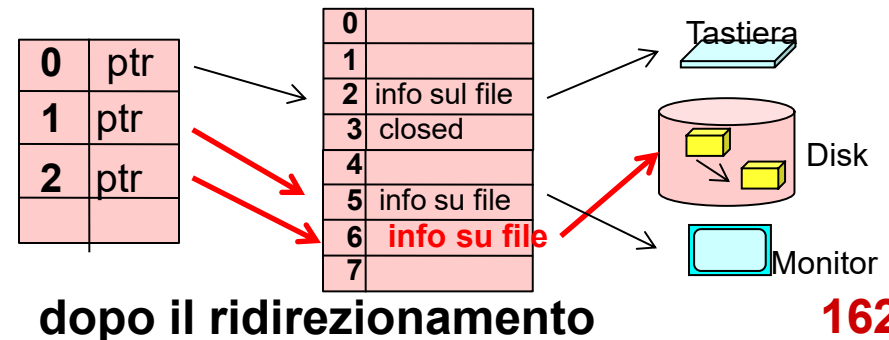
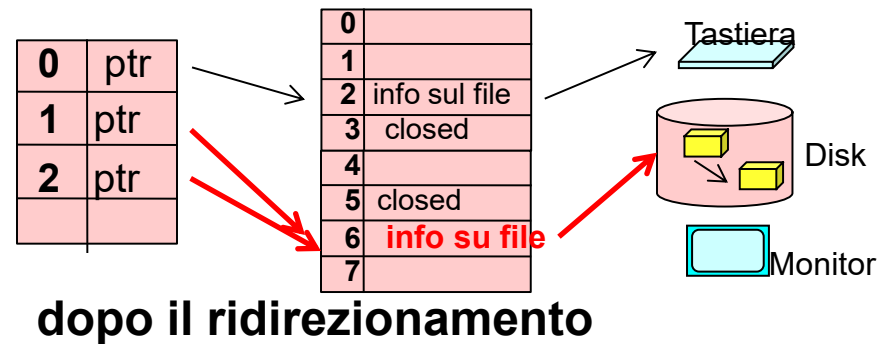
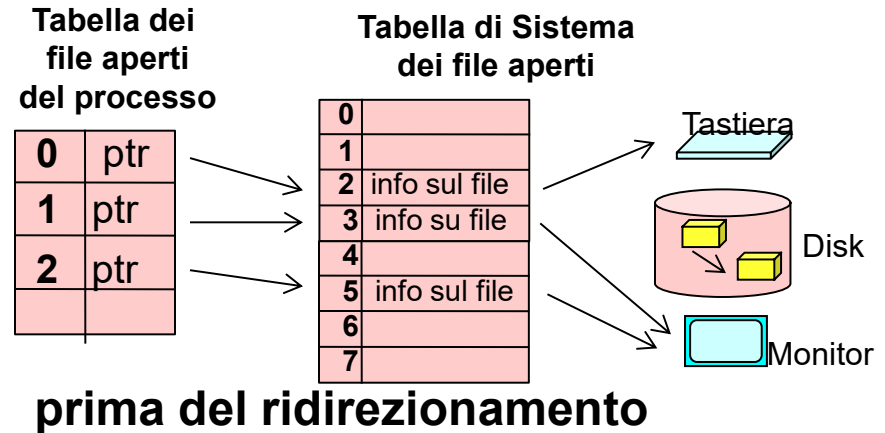
```
ls pippo.txt 2> error.txt 1>&2
```

ridireziona stderr di ls sul file error.txt e poi ridireziona stdout di ls su stderr di ls che attualmente punta su error.txt, quindi **entrambi stdout e stderr di ls finiscono nel file error.txt**

invece

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

ridireziona stdout di ls su stderr di ls che attualmente punta a video  
ridireziona stderr di ls su error.txt  
quindi alla fine **stderr punta ad error.txt mentre stdout scrive su video**





# Ridirezionamenti di Stream di I/O (14)

## Esempio: ridirigere il solo stderr sullo stdin di un altro programma

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

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

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

Notare le differenze tra il precedente ridirezionamento e quello qui sotto, in cui l'ordine dei due ridirezionamenti è invertito:

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

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

# Ridirezionamenti di Stream di I/O (15)

## Scorciatoia |& invece di composizione 2>&1 |

Come visto nella pagina precedente, il Ridirezionamento di entrambi gli stream di output ed error verso lo stdin di altro programma, puo' essere fatto utilizzando la combinazione di operatori

**2>&1 |**

Provarlo eseguendo i due seguenti comandi in una directory in cui non esiste un file che si chiama pippo.txt, in modo da generare un messaggio di errore "no such file or directory", ridirezionarlo sullo standard output, collegarlo allo standard input di grep, selezionare la sola riga che contiene such e mandarla a video.

```
ls pippo.txt 2>&1 | grep such
```

```
ls * 2>&1 | grep such
```

Nota bene: **esiste un operatore apposito** che rappresenta una scorciatoia **dell'operatore composito 2>&1 |** che fa la stessa cosa, ed e' **|&**

```
ls pippo.txt |& grep such
```

# Ridirezionamenti di Stream di I/O (16)

E' importante notare **la differenza nelle tempistiche di esecuzione** dei programmi quando questi sono collegati dal separatore `;` o dalla `|`

```
program1 ; program2 ; program3
```

Con il separatore `;` io faccio eseguire i diversi programmi uno dopo l'altro, cioè prima che parta il secondo programma deve finire il primo e così via.

Inoltre l'output di un programma non viene ridirezionato nell'input del programma successivo.

```
program1 | program2 | program3
```

Invece, con la `|` i programmi specificati partono assieme e l'output di un programma viene ridirezionato nell'input del programma successivo.

# Ridirezionamenti di Stream di I/O (17)

E' possibile ridirigere contemporaneamente standard output e standard error di un programma `program1` ridirigendolo verso lo standard input di un programma `program2`, utilizzando l'operatore `|&`

`program1 |& program2`

Con l'operatore composito `|&` i due programmi specificati partono assieme e l'output (sia `stdout` che `stderr`) del primo programma sono ridirezionati insieme nell'input del secondo programma.

# Auto-Ridirezionamento (18)

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

Esempio:

Supponiamo di avere un file `rightedicomando.txt` così fatto:

```
echo ciao
echo ${BASHPID}
ls /proc/${BASHPID}/fd/
sleep 10
```

Proviamo ad eseguire i seguenti comandi in una shell interattiva

```
exec {NUOVOFD}< rightedicomando.txt
exec 0>&${NUOVOFD}
```

La bash usa come stdin il file `rightedicomando.txt`, quindi visualizzerà i fd della bash stessa, poi aspetterà 10 secondi e infine terminerà chiudendo la finestra interattiva.

# Ridirezionamenti di Stream di I/O (19)

## Ridirezionamento per blocchi di comandi

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

Ad esempio, eseguendo il seguente script, applico un ridirezionamento dello standard output a tutti i comandi all'interno del loop del while :

(file ridirezione1BloccoWhile.sh)

```
NUM=1
echo "${NUM}"
while (( "${NUM}" <= "3" )) ; do
    echo "${NUM}"
    ((NUM=${NUM}+1))
done > pippo.txt
echo "${NUM}"
```

Dopo l'esecuzione dello script, nel file pippo.txt troveremo le 3 righe:

```
1
2
3
```

mentre a video vediamo le 2 righe

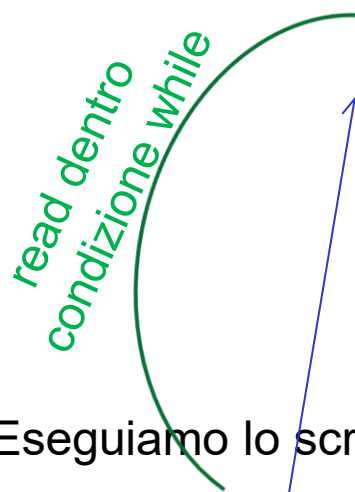
```
1
4
```

# Ridirezionamenti di Stream di I/O (20)

## Ridirezionamento per blocchi di comandi

E' possibile applicare un **ridirezionamento** anche **per flussi di input**.

Supponiamo di avere un file *Assoluzionilnaspettate.txt* strutturato come nell'esempio: una riga con il reo, seguita da una riga con l'accusa, e così via:



vittorio ghini	17/07/1994
blasfemia	assolto per insufficienza di prove
giovanni pau	15/07/1995
ubriachezza	denuncia ritirata
luca andreucci	03/01/1992
vilipendio	assolto per scomparsa del denunciante

Eseguiamo lo script `ridirezioneInputBloccoFor.sh` che segue

```
while read nome cognome data ; do
  if read accusa verdetto ; then
    echo $cognome errore $verdetto
  else
    echo terminazione inaspettata del file di input
    exit 1
  fi
done < Assoluzionilnaspettate.txt
```

# Ridirezionamenti di Stream di I/O (21)

## Esempi Ridirezionamento per blocchi di comandi

ridirezione1BlocchiIfElse.sh

### Ridirezionamento per blocchi di comandi

```
NUM=1
echo "${NUM}"
if (( "${NUM}" <= "3" )) ; then
    ((NUM=${NUM}+1))
    echo "${NUM}"
else
    ((NUM=${NUM}+2))
    echo "${NUM}"
fi > pippo.txt
echo "${NUM}"
```

L'output del ramo if e del ramo else finiscono entrambi nel file pippo.txt. Il primo e l'ultimo echo vanno in stdout

ridirezione1Annidata.sh

### Annidamento dei ridirezionamenti

```
NUM=1
echo "${NUM}"
if (( "${NUM}" <= "3" )) ; then
    ((NUM=${NUM}+1))
    echo "${NUM}"
    echo "vaffa" > cacchio.txt
else
    ((NUM=${NUM}+2))
    echo "${NUM}"
fi > pippo.txt
echo "${NUM}"
```

L'output in pippo.txt rimane quello dell'esempio di sinistra, mentre nel file cacchio.txt mi ritrovo la riga con vaffa



# Ridirezionamenti di Stream di I/O (22)

## Esempi Ridirezionamento per blocchi di comandi

ridirezioneBlocchiIfElse.sh

**NB: Anche i comandi dentro la condizione dell' if vengono ridirezionati assieme a quelli del blocco if then else**

```
if ls ./out1.txt ; then
    echo "esiste"
else
    echo "non esiste"
fi &> pippo.txt
```

Se il file ./out1.txt esiste, allora nel file pippo.txt trovo,

```
./out1.txt
esiste
```

Se invece il file ./out1.txt non esiste, allora nel file pippo.txt trovo,

```
ls: cannot access './out1.txt':
    No such file or directory
non esiste
```

ridirezioneBloccoWhile.sh

**NB: Anche i comandi dentro la condizione di for e while vengono ridirezionati come quelli del blocco do done**

```
while ls ./out1.txt ; do
    echo "esiste ma ora ..."
    rm ./out1.txt
done &> pippo.txt
echo "non esiste "
```

Se il file ./out1.txt esiste, allora nel file pippo.txt trovo,

```
./out1.txt
esiste ma ora ...
```

```
ls: ./out1.txt': No such file or directory
```

Se invece il file ./out1.txt non esiste, allora nel file pippo.txt trovo solo,

```
ls: ./out1.txt': No such file or directory
```

# Ridirezione dell'input passato a riga di comando

## Here Documents << (ridireziona più righe) 1/2

La word che segue il metacarattere << viene utilizzata come delimitatore finale dell'input da passare al comando a sinistra del metacarattere.

Viene ridirezionato nell'input tutto quello che compare tra la word esplicitata dopo il metacarattere << e fino a dove quella word compare ancora all'inizio di una riga.

NB: NON vengono espansi eventuali metacaratteri presenti nella word.

**Utilità del ridirezionamento: poter passare input come se provenisse da un file ma senza dover scrivere un file.**

**Esempio, il seguente script:**

```
while read A B C ; do echo $B ; done <<FINE
uno due tre quattro
alfa beta gamma
gatto cane
FINE
echo ciao
```

produce in output  
due  
beta  
cane  
ciao

# Ridirezione dell'input passato a riga di comando

## Here Documents << (ridirezione più righe) 2/2

Notare che, dentro alle righe passate come input, posso inserire delle variabili che vengono espanse. Non occorre che le variabili siano di ambiente perché vengono processate dalla bash corrente prima di creare la subshell che esegue il gruppo di comandi

**Esempio: il seguente script**

```
VAR=contenuto  
while read A B C ; do echo $B ; done <<FINE  
uno due tre quattro  
alfa ${VAR} gamma  
gatto cane  
FINE  
echo ciao
```

produce in output:

```
due  
contenuto  
cane  
ciao
```

# Ridirezione dell'input passato a riga di comando

## Here Strings <<< (ridireziona una sola parola)

Viene ridirezionato nell'input la PRIMA parola che compare subito dopo il metacarattere <<<

Se dopo il <<< c'è una variabile questa viene espansa.

Utilità del ridirezionamento: poter passare input contenuto in una variabile

Esempio: una sola parola

```
read A B C <<< alfa
```

```
echo 1 $A 2 $B 3 $C
```

produce in output

```
1 alfa 2 3
```

**Esempio con espansione variabili:**

```
VAR=paperino
```

```
read A B C <<< ${VAR}
```

```
echo 1 $A 2 $B 3 $C
```

produce in output

```
1 paperino 2 3
```

Esempio: più parole con quoting

```
read A B C <<< "alfa beta gamma"
```

```
echo 1 $A 2 $B 3 $C
```

produce in output

```
1 alfa 2 beta 3 gamma
```

Esempio più parole senza quoting:

```
read A B C <<< alfa beta gamma
```

```
echo 1 $A 2 $B 3 $C
```

produce in output

```
1 alfa 2 3
```

# Raggruppamento di comandi (1)

**Se si racchiude una sequenza di comandi tra parentesi tonde, allora in esecuzione viene creata una subshell per eseguire quella sequenza dei comandi.**

**Il risultato restituito dalla subshell (restituito dalla parentesi tonda) è il risultato dell'ultimo comando eseguito nella subshell**, cioè l'ultimo comando eseguito tra quelli dentro le parentesi tonde.

Tutti i comandi condividono gli stessi stdin/stdout/stderr **utilizzandoli in sequenza**. Quindi, all'output del primo comando dentro le parentesi viene concatenato l'output del secondo comando poi quello del terzo etc etc etc

**Ciò permette di trattare / ridirigere input ed output di tutti i comandi dentro le parentesi tonde come se fossero un solo comando.**

Esempio: con il comando:

```
ls; pwd; whoami > out.txt
```

visualizzo

```
nomi files in directory corrente
```

```
/home/vittorio
```

E dentro il file out.txt trovo

```
vittorio
```

Esempio: con il comando tra parentesi

```
( ls; pwd; whoami ) > out.txt
```

non visualizzo nulla

e dentro il file out.txt trovo

```
a1B a2B aB akB akmB akmtB
```

```
/home/vittorio
```

```
vittorio
```

# Raggruppamento di comandi (2)

stdin stdout e stderr dei singoli comandi dentro le parentesi tonde vengono concatenati

## **concatenazione stdout**

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

il comando `grep` legge, come se provenissero da tastiera, le righe di entrambi i file prima le righe di `file1.txt` e poi le righe di `file2.txt`

## **concatenazione stdout e stderr**

```
( cat file1.txt ; cat file2.txt ) |& grep stringa
```

il comando `grep` legge, come se provenissero da tastiera, le righe di entrambi i file prima le righe di `file1.txt` e poi le righe di `file2.txt`

## **concatenazione stdin**

```
cat file1.txt | ( read RIGA1 ; usa RIGA1 ; read RIGA2 ; usa RIGA2 )
```

i due comandi `read` leggono uno la prima e l'altro la seconda riga prodotte da `cat`

# Raggruppamento di comandi (3)

la bash è un po' stronza e cerca di ottimizzare, nonostante la nostra volontà

Esempio per far vedere che :

- **se dentro la parentesi tonda metto un solo comando, allora la bash padre non crea una altra shell figlia in cui far eseguire il singolo comando**

```
ps ; ( ps )
```

vedere che l'output dei due ps mostra una sola bash in entrambi i casi

lanciare poi

```
ps ; ( ps ; ps )
```

e vedere che nell'output dei due ps interni compaiono due bash, quella padre e una figlia

# Commenti Aggiuntivi da fare sulla PROSSIMA SLIDE

## Quella su uso di ridirezionamenti e GNU Coreutils

- Far vedere il comando **man man** ed evidenziare il concetto **di sezione del man** e le due **sezioni 1** (executable programs and shell commands) e **sezione 3** (library calls).
- Far capire che conviene (se possibile) usare in cascata più eseguibili già esistenti invece che scrivere un programma apposta da zero.
- Citare il pacchetto **coreutils** e ricordare che proviene dall'unione di più pacchetti precedenti (fileutils, shellutils, textutils) e suggerire di guardare cosa altro c'è .
- Descrivere l'esercizio proposto e la soluzione
- Descrivere le utilities **head tail sed cut cat grep** e poi **tee**.
- Occhio al parametro 4- passato a cut.
- Evidenziare che la maggior parte delle utilities prevedono un doppio comportamento, ovvero:
  - possono accettare dopo le opzioni uno o più argomenti che specificano il nome del/dei file da cui leggere input.
  - possono leggere input da stdin se non c'è l'argomento nomefile o se c'è un **-** come argomento finale.
- Fare un esempio in cui uso tail senza specificare un argomento nomefile per fargli usare input proveniente da stdin ed evidenziare il comportamento differente rispetto a come l'ho usato nell'esercizio
  - `cat dati.txt | tail -n 3`
- Aggiungere un esempio con l'utility **tee**
- Far vedere `tail -f` e mostrare che non viene gestita mutua esclusione sui file



# Esempio di uso di ridirezionamenti e GNU Coreutils

Cosa si puo' fare con sequenze di comandi, raggruppamenti, ridirezionamenti, ...?

Ne approfitto per farvi vedere alcune utilities che manipolano file di testo, appartenenti al progetto GNU Coreutils (unificazione dei vecchi pacchetti Fileutils, Shellutils e Textutils).

Vedi <http://www.gnu.org/software/coreutils/coreutils.html>

-----

Dato un file di testo, di nome dati.txt, utilizzare programmi disponibili comunemente in un sistema linux per ottenere il seguente risultato:

Prendere le prime 2 righe del file e le ultime 3 righe del file stesso.

Di queste righe selezionare solo quelle che contengono la sequenza AL

Nelle righe rimaste, sostituire le lettere AL con le lettere CUF

Nelle righe cosi' modificate eliminare i primi 3 caratteri (mantenere dal 4° in poi)

Scrivere le righe modificate nel file output.txt

## **dati.txt**

pappappero  
caracavALo  
piripiccione  
ammappete  
uffauffa  
pedalando  
avvALLare  
remare

## **output.txt**

acavCUFo  
CUFlare

Soluzione:

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

# Esempi di uso di utilities in Coreutils

```
cat dati.txt | tail -n 3
```

```
tail -f /var/log/messages
```

```
cat dati.txt | tee a.txt b.txt
```

-----

**ESEMPIO di creazione e riempimento di file in una directory in cui devo assumere i permessi di root.**

- Comando che provoca errore, no permessi di scrittura  
echo "contenuto del file" > /root/prova.txt

- Ancora errore, sudo viene eseguito dopo ridirezionamento:  
sudo echo "contenuto del file" > /root/prova.txt

- OK  
sudo tee /root/prova.txt <<END  
"contenuto del file"  
END

# Qualche dettaglio: coreutils e codifica dei caratteri

Alcune delle coreutils, in particolare **cut** e **wc**, consentono all'utente di chiedere di lavorare sui bytes oppure sui caratteri. Qual è la differenza?

I caratteri possono essere codificati con **8 bit** (Latin-1 (iso-8859-1) cioè codifica ascii estesa) oppure con **due bytes** (UNICODE UTF-16) oppure con **un numero variabile di bytes** (codifiche multibyte, UNICODE UTF-8, compatibile con ASCII ).

Se chiediamo a cut o wc di lavorare con i caratteri, i processi analizzano il flusso di byte e determinano come sono codificati i caratteri e poi li contano o rimuovono o lasciano a seconda del comando richiesto.

Se in un file sono contenuti due caratteri codificati con due bytes ciascuno e due caratteri codificati con 1 byte ciascuno, allora il numero di caratteri sarà 4 mentre il numero di bytes di quel files sarà 6.

Normalmente wc usa la variabile LC\_CTYPE (tipo locale dei caratteri) per decidere che tipo di codifica dei caratteri deve usare, normalmente è UTF-8. Ad esempio, LC\_CTYPE=en\_US.UTF-8.

Se la codifica è UTF-8 allora caratteri e bytes sono due cose diverse.

Se la codifica è C allora si considerano char di 1 byte ed allora byte e caratteri sono la stessa cosa:

Esempio in cui uso carattere è codificato con i due caratteri 0xc3 0xa9.

comando	output
printf '\xc3\xa9'   LC_CTYPE=en_US.UTF-8 wc -c	2
printf '\xc3\xa9'   LC_CTYPE=en_US.UTF-8 wc -m	1
printf '\xc3\xa9'   LC_CTYPE=C wc -c	2
printf '\xc3\xa9'   LC_CTYPE=C wc -m	2

# sed - stream editor (coreutils) (1)

l'editor di stream, **sed** prevede una quantita' assurda di possibili comandi, vediamo alcuni come esempi. Lanciare man sed per ulteriori dettagli.

Nell'esempio precedente, `sed 's/AL/CUF/g'` Il carattere **g** serve a far sostituire, in ciascuna linea processata, **tutte** le occorrenze delle stringhe AL. In assenza del carattere g, in ciascuna riga verrebbe modificata solo la prima stringa AL incontrata.

## Esempi con sed

Sostituisce **la prima occorrenza** di toglì con metti in ciascuna riga del file nomefile.

```
sed 's/togli/metti/' nomefile
```

Rimuovere **il primo tra i caratteri** a che trova in ciascuna riga del file nomefile

```
sed 's/a/' nomefile
```

Rimuovere il carattere in prima posizione di ogni linea che si riceve dallo standard input.

^ significa inizio linea, . significa un carattere qualunque

```
sed 's/^./'
```

Rimuovere **l'ultimo carattere** di ogni linea ricevuta dallo stdin. \$ significa fine linea, il . significa un carattere qualunque

```
sed 's/.$/'
```

Esegui **due rimozioni insieme** (;) Rimuovere il primo e l'ultimo carattere in ogni linea.

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

Rimuove i primi 3 caratteri ad inizio linea.

```
sed 's/.../'
```

Rimuove i primi 4 caratteri ad inizio linea.

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

# sed - stream editor (coreutils) (2)

## Continua esempi con sed

To remove last n characters of every line (nell'esempio 3 caratteri)

```
sed -r 's/{3}$//'
```

To remove everything except the 1st n characters in every line

```
sed -r 's/(.{3}).*\1/'
```

To remove everything except the last n characters in a file

```
sed -r 's/.*({3})\1/'
```

To **remove multiple characters** present in a file (**g means all occurrences**):

```
sed 's/[aoe]//g'
```

To remove **all occurrences** of a pattern

```
sed 's/lari//g'
```

To delete only nth occurrences of a character in every line (2 occurrences in the example)

```
sed 's/u//2'
```

To delete everything in a line followed by a character

```
sed 's/a.*//'
```

To remove all alpha-numeric characters present in every line

```
sed 's/[a-zA-Z0-9]//g'
```

# Process Identifier di Shell bash: \$\$ e \$BASHPID

- ✱ Ogni Processo è identificato da un identificatore numerico univoco, il PID.
- ✱ Anche le shell in esecuzione posseggono un loro PID.
- ✱ Se occorre conoscere il PID di una shell in esecuzione, si usa la variabile \$\$ che viene espansa con il PID della shell corrente.

## Esempio:

```
echo il pid della shell è $$  
il pid della shell è 14726
```

- ✱ Se uso la variabile \$\$ all'interno di uno script, vedo il PID della bash che esegue lo script.
- ✱ **Esiste una eccezione di comportamento.** Se raggruppo dei comandi facendoli eseguire in una subshell (), allora dentro le () la variabile \$\$ mi fa vedere il PID della bash più esterna, non della subshell ().

## Esempio:

```
echo -n pid fuori $$ ; ( echo -n pid dentro $$ ; pwd )  
pid fuori 14726 pid dentro 14726 /home/studente
```

- ✱ Se occorre usare il PID di una subshell specificata con (), occorre usare la variabile d'ambiente BASHPID. **La variabile BASHPID contiene il PID della bash corrente, anche se questa è una bash creata con ().**

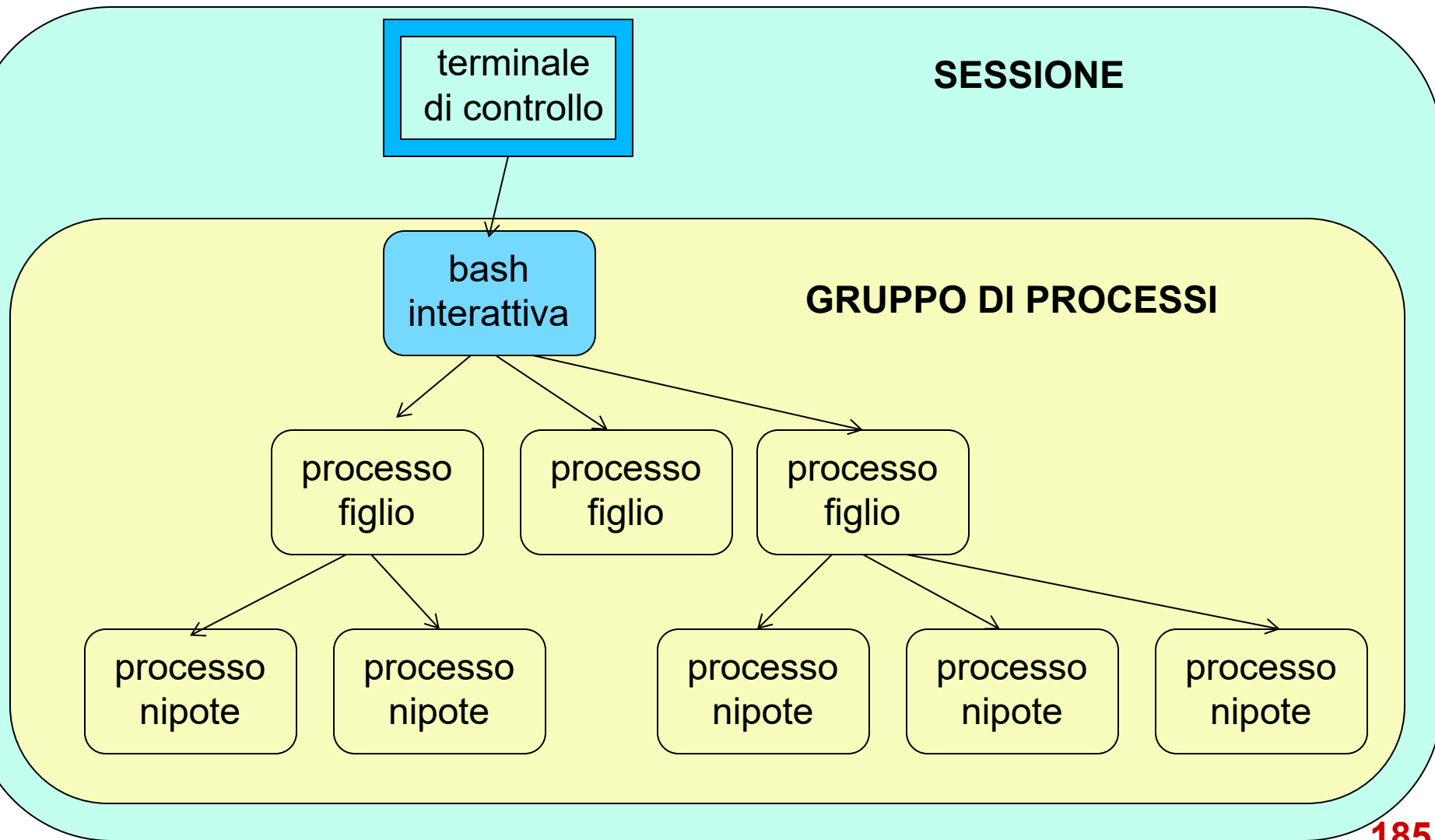
## Esempio:

```
echo -n pid fuori $$ ; ( echo -n pid dentro $BASHPID; pwd )  
pid fuori 14726 pid dentro 22399 /home/studente
```

- ✱ Posso usare sempre la variabile BASHPID al posto della \$\$, però la var BASHPID è definita solo in bash e solo in bash delle versioni > 4. BASHPID è meno portabile **184**

# Concetti di Processo, Sessione, Gruppo di Processi, Terminale di controllo del gruppo di processi (0)

## ☀ Scenario



# Concetti di Processo, Gruppo di Processi, Terminale di controllo del gruppo di processi (1)

- ✿ **Processi:** Un processo è la più piccola unità di elaborazione completa ed autonoma che può essere eseguita in un computer.
- ✿ Un **processo** è un insieme di thread di esecuzione operanti all'interno di un contesto, il quale comprende uno spazio di indirizzamento in memoria ed una tabella dei descrittori di file aperti per quel processo.
- ✿ **Gruppo di processi** (process group): Un processo può lanciare l'esecuzione di altri processi: questi altri processi appartengono allo stesso gruppo di processi del processo padre, a meno che non si svolgano azioni che modificano il gruppo di appartenenza.
- ✿ **Terminale di controllo:** Un processo lanciato in esecuzione può avere un "controlling terminal" (un terminale da cui è controllato, che è l'astrazione di terminale (console=video+tastiera) da cui prende gli standard input output ed error.
- ✿ **Terminale di controllo del gruppo di processi** Un processo lanciato in esecuzione eredita lo stesso "controlling terminal" dal padre che lo ha lanciato, a meno che non si svolgano azioni che sganciano il processo dal terminale di controllo. Quindi tutti i processi di uno stesso gruppo di processi condividono lo stesso terminale di controllo.



# Creazione della Sessione

- ✱ Per gestire l'insieme di processi, il kernel necessita di raggruppare i processi in modo più complesso che non la semplice relazione padre-figlio.
- ✱ Perciò viene usata l'astrazione di "**sessione**" e di "**gruppo di processi**".
- ✱ Un utente normalmente apre una **shell interattiva** (di login o no) che viene inserita dal sistema operativo in un terminale (la window che rappresenta la console monitorvideo+tastiera) e da questa shell eseguirà operazioni varie, tra cui l'esecuzione di comandi, eseguibili binari e script.
- ✱ Il terminale in cui la shell interattiva esegue è detto "**terminale di controllo**" (controlling terminal).
- ✱ La shell interattiva all'inizio **crea una a sessione**, specificando un nuovo sessionId, e **lega la sessione al terminale di controllo**. Questa shell interattiva diventa così il leader della sessione.
- ✱ I processi discendenti di quella shell apparterranno alla stessa sessione, a meno che non si eseguano operazioni di distacco.
- ✱ La shell interattiva prende come stdin stdout ed stderr quelli che il terminale gli fornisce.
  - ✱ I comandi, eseguibili binari e script lanciati dalla shell interattiva ereditano i file aperti del padre, quindi ereditano anche gli stdin, stdout ed stderr del terminale.

# Motivazione della Sessione

- ✱ Quando un utente si disconnette da un sistema, il kernel deve terminare tutti i processi che l'utente sta ancora eseguendo
  - ✱ altrimenti, gli utenti lascerebbero un gran numero di vecchi processi in attesa di input che non potranno mai arrivare.
- ✱ Come determinare quali processi terminare?
- ✱ Per semplificare questa attività, i processi sono organizzati in sessioni.
- ✱ Il leader della sessione è il processo che ha creato la sessione.
- ✱ L'ID della sessione è uguale al pid del processo che ha creato la sessione tramite la chiamata di sistema `setsid ()`.
  - ✱ La funzione `setsid ()` non accetta argomenti e restituisce il nuovo ID di sessione., cioè il PID del processo leader.
- ✱ Tutti i discendenti di quel processo sono quindi membri di quella sessione a meno che non si rimuovano specificamente da essa.

# Terminale di controllo (Controlling Terminal) di una sessione

- ☀ Ogni sessione è legata a un terminale da cui i processi nella sessione ricevono il loro input e ai quali inviano il loro output.
- ☀ Quel terminale può essere la console locale della macchina, un terminale connesso su una linea seriale o uno pseudo terminale che si collega a una finestra del window manager locale o attraverso una rete.
- ☀ Il terminale a cui è correlata una sessione è chiamato terminale di controllo (o control tty) della sessione.
- ☀ **Un terminale può essere il terminale di controllo per una sola sessione alla volta.**
- ☀ Sebbene il terminale di controllo per una sessione possa essere modificato, solitamente l'operazione di stabilire il terminale di controllo di una sessione viene eseguita solo dal processo che gestiscono l'accesso iniziale di un utente a un sistema o ad una finestra del window manager.

# Chiusura del Terminale e conseguente Terminazione dei processi della sessione

- ✱ Quando viene chiuso il terminale di una sessione, i processi che appartengono a quella sessione, non possono più continuare l'esecuzione e vengono terminati.
- ✱ L'operazione di terminazione viene realizzata dal sistema operativo mandando ad ogni processo un segnale di terminazione detto SIGHUP ("signal hang up").
- ✱ Ricevendo questo segnale, ogni processo termina.
- ✱ In tal modo si liberano le risorse del sistema operativo legate ai processi legati alla sessione terminata a causa della chiusura (voluta o inaspettata) del terminale di controllo della sessione.
- ✱ Nota Bene: un processo può essere configurato, da programma, per reagire diversamente all'arrivo di un segnale.

# Processi in background e in foreground (1)

## ☀ Processi in foreground

- Processi che “controllano” il terminale dal quale sono stati lanciati, nel senso che hanno il loro standard input collegato al terminale e impegnano il terminale non permettendo l'esecuzione di altri programmi collegati a quel terminale, fino alla loro terminazione
- In ogni istante, un solo processo è in foreground

## ☀ Processi in background

- Vengono eseguiti in parallelo rispetto all'esecuzione della bash, nel senso che permettono alla bash che li ha lanciati di leggere ed eseguire altri comandi e di lanciare altri programmi. Usano una copia dei fd della bash che li ha lanciati, quindi condividono con questa stdin/stdout/stderr. Poiché lo standard input rimane collegato al terminale della bash che li ha lanciati, quando terminiamo l'esecuzione della finestra del terminale, il terminale chiude lo stdin ed i processi vengono terminati. Per far proseguire l'esecuzione dei processi in background dopo la terminazione del terminale occorre sganciare i processi dal terminale col comando disown (vedi piu' avanti).

## ☀ Jobs di una shell

- Processi in background o sospesi **solo figli** di quella shell (vedi jobs).

## ☀ Job control

- Permette di portare i processi da background a foreground e viceversa

# Processi in background e in foreground (2)

- ✱ **&** lancia un processo direttamente in background (in **\$!** trovero' il pid)

Esempio: `prova &`

- ✱ **^Z** sospende un processo in foreground
- ✱ **^C** termina un processo in foreground
- ✱ **bg** riprende l'esecuzione in background di un processo sospeso
- ✱ **jobs** produce una lista numerata dei processi in background o sospesi nella shell corrente
  - il numero tra parentesi quadre che indica ciascun processo **non e' il pid** bensì **un indice del job** che si usa per gestire il job, premettendo il carattere %
- ✱ **fg %n** porta in foreground un processo sospeso
- ✱ **kill** elimina il processo specificato dal proprio identificatore pid oppure specificato dal numero del job

Esempio:

```
kill 6152    dove 6152 e' il pid del processo
kill %2      dove 2 e' l'indice del processo nella
              lista visualizzata dal comando jobs
```

- ✱ **disown -[ar] jobs** sgancia il job dalla shell interattiva che lo ha lanciato i jobs sono specificati con pid o **%indice**

- **Nota:** **jobs** si applica ai soli processi attualmente in esecuzione (o sospesi) nella shell, mentre la lista completa dei processi in esecuzione nel sistema può essere ottenuta con i comandi **ps -ax** oppure con **top**.

## Precisazione:

# \$! PID dell'ultimo processo lanciato in background

Quando lancio un processo direttamente in background, mediante l'operatore **&** allora il Process Identifier PID del processo appena lanciato viene messo nella variabile **\$!** della shell che ha lanciato quel processo.

La variabile **\$!** della shell mantiene quel valore fino a che la stessa shell non lancia in background un altro processo.

```
program.exe arg1 arg2 arg3 .... argN &  
echo il PID di program.exe e\' $!
```

Anche quando lancio il programma in background, sganciandolo dalla shell mediante l'utility **nohup** (vedi dopo), nella variabile **\$!** trovo il PID del processo appena lanciato in background

```
nohup program.exe arg1 arg2 arg3 .... argN &  
echo il PID di program.exe e\' $!
```

Nota Bene

Al contrario, se metto in background con l'utility **bg** un processo che era stato sospeso con **CTRL+Z**, allora dentro la variabile **\$!** non viene messo il PID di quel processo, cioè la variabile **\$!** non viene modificata.

# Processi in background e in foreground (3)

- ✱ **Nota:** l'operatore `&` inserito nella riga di comando serve anche a indicare la fine degli argomenti passati al comando, così come fanno gli operatori `;` `&&` `||` etc etc.
- ✱ Se si usa l'operatore `&` bisogna omettere il `;`
- ✱ Es:  
`./fai.sh 3 & ; ./fai.sh 2 &`  
la presenza del `;` produce **errore**  
    `-bash: syntax error near unexpected token `;'`  
Il comando va lanciato senza il `;`  
`./fai.sh 3 & ./fai.sh 2 &`

---

## Prova chiarificatrice

Provare a lanciare la sequenza di comandi

`sleep 23 & sleep 20`

e immediatamente digitare

`CTRL+Z`

poi lanciare il comando

`jobs`

poi lanciare il comando

`bg`

e nuovamente lanciare il comando

`jobs`



# Processi in background e in foreground (3bis)

## Precisazione per Tarlo3 (andrea)

### Concetto di current job e simbolo %% per usarlo

✿ Tratto da `man bash`, nella parte relativa a Job Control

The symbols `%%` and `%+` refer to the shell's notion of the current job, which is the last job stopped while it was in the foreground or started in the background.

The previous job may be referenced using `%-`. If there is only a single job, `%+` and `%-` can both be used to refer to that job.

In output pertaining to jobs (e.g., the output of the `jobs` command), the **current job is always flagged with a +**, and the **previous job with a -**.

**A single % (with no accompanying job specification) also** refers to the current job.

```
(sleep 50 ; echo UNO )
```

```
^Z
```

```
bg
```

```
(sleep 50 ; echo DUE) &
```

```
disown %%                (o disown % o anche solo disown ) sgancia quello con DUE
```

Se invece lanciassi

```
disown %-    sgancerei quello con UNO
```

Nota bene: man mano che i job terminano o che li sgancio, cambiano il job corrente e il job precedente.

# Processi in background e in foreground (4)

- ✿ **Nota:** quando un processo viene lanciato da una shell interattiva e non opera più in foreground, allora e' un job che può trovarsi in stato sospeso oppure eseguire in background. In entrambi i casi, **quando la shell interattiva viene terminata**, questa invia un segnale di tipo **SIGHUP** ai suoi jobs ordinando loro di terminare.
- ✿ **Per impedire che i jobs di una shell interattiva vengano terminati quando la shell termina** occorre "sganciare" i jobs dalla shell prima che questa termini.
- ✿ Il comando **disown** lanciato dalla shell interattiva serve a dichiarare che un determinato job non appartiene piu' alla shell, quindi al job non dovrà più essere inviato il segnale di SIGHUP quando la shell interattiva termina.
  - ✿ **disown** **sgancia dalla shell l'ultimo job messo in background**
  - ✿ **disown -r** **sgancia dalla shell tutti i job running**
  - ✿ **disown -a** **sgancia dalla shell tutti i job running e sospesi**
  - ✿ **disown %jobid** **sgancia dalla shell il job specificato**

**Vedere esempi slide successiva**

# Processi in background e in foreground (5)

**Verifichiamo che i jobs possono anche essere delle subshell lanciate per eseguire dei comandi raggruppati:**

Lanciare una nuova shell interattiva, eseguire il seguente comando e osservare:

```
( sleep 50 ; echo finito > merda.txt ) &
```

```
jobs
```

```
ps
```

 (notare che esiste una shell che esegue sleep)

Dopo 50 secondi verificare che esiste il file merda.txt e poi eliminarlo.

**Verifichiamo che i jobs muoiono con la shell:**

Lanciare una nuova shell interattiva, eseguire il seguente comando e osservare:

```
( sleep 50 ; echo finito ; echo scrivo > merda.txt ) &
```

```
jobs ; ps
```

```
exit
```

Da una nuova shell lanciare `jobs ; ps`

Aspettare 51 secondi e poi verificare se esiste il file merda.txt. **Non c'è'.**

**Attenzione: il segnale viene processato solo quando il processo è attivo, quindi solo al termine della sleep. probabilmente esegue anche echo finito**

Se elimino "echo finito" vedrò ugualmente il file merda.txt

**Controprova Verifichiamo che i jobs sganciati sopravvivono alla morte della shell**

Lanciare una nuova shell interattiva, eseguire il seguente comando e osservare:

```
( sleep 50 ; echo finito ; echo scrivo > merda.txt ) &
```

```
jobs ; ps
```

```
disown
```

o `disown %1` se il job ha indice 1

```
jobs ; ps
```

```
exit
```

Da una nuova shell lanciare `jobs ; ps`

Aspettare 50 secondi e poi verificare se esiste il file merda.txt. **C'è'.**

# Processi in background e in foreground (6)

## Lanciare processo già “sganciato” dalla shell

Esiste un comando esterno alla shell che puo' essere utile per sganciare un processo dalla shell che lo ha lanciato il tutto in una sola istruzione. Puo' essere comodo per non doversi occupare di reperire il pid del processo creato o l'indice del job creato.

In pratica, lanciando il comando:

**nohup** comando arg1 arg2 .... argN **&**

Ottengo lo stesso effetto che avrei lanciando la sequenza di comandi

**comando** arg1 arg2 .... argN **&**

**disown** *sgancio dalla shell l'ultimo processo mandato in background*

nohup e' poco comodo perche' non permette, se non con dei trucchi, di lanciare una sequenza di comandi ma solo un comando singolo (con i suoi argomenti).

# Segnali - kill (1) Interruzioni software mandati ai processi per notificare eventi asincroni)

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

Il comando `kill` invia segnali a processi.

Il tipo di segnale viene specificato con un simbolo oppure con il valore numerico.

Il processi destinazione vengono specificati mediante il loro pid.

Supponiamo che 7777 sia il pid di un processo.

I seguenti comandi inviano il segnale SIGTERM al processo destinazione, con ciò ordinandogli di terminare. Il segnale viene accettato solo se proviene dall'utente che esegue il processo (effective user). Il processo può intercettare il segnale e gestirlo diversamente o addirittura ignorarlo. Come si vede, se non si specifica un segnale, **per default** viene inviato il segnale **SIGTERM** (valore 15) di terminazione del processo.

<code>kill</code>		<code>7777</code>
<code>kill</code>	<b>-15</b>	<code>7777</code>
<code>kill</code>	<b>-SIGTERM</b>	<code>7777</code>

Esiste un altro segnale di terminazione del processo, SIGKILL o KILL che però il processo non può ignorare.

<code>kill</code>	<b>-9</b>	<code>7777</code>
<code>kill</code>	<b>-SIGKILL</b>	<code>7777</code>
<code>kill</code>	<b>-KILL</b>	<code>7777</code>

E' possibile inviare un segnale di terminazione a tutti i processi che il processo bash corrente può far terminare (cioè tutti i suoi figli diretti, non i nipoti)

<code>kill</code>	<b>-9</b>	<code>-1</code>
-------------------	-----------	-----------------

# Segnali - kill (2)

Interruzioni software mandati ai processi  
per notificare eventi asincroni)

Altri possibili segnali sono:

kill	<b>-SIGTSTP</b>	pid	(è il segnale lanciato con CTRL Z )
kill	<b>-SIGCONT</b>	pid	(riprende processo sospeso, usato da fg e bg)
kill	<b>-SIGINT</b>	pid	(è il segnale lanciato con CTRL C)

ed altri ancora.

E' possibile visualizzare l'elenco dei segnali gestiti eseguendo il comando

kill -l

NOTA BENE: RISPOSTA DILAZIONATA AI SEGNALI.

**Se un processo è bloccato in attesa di input da rete oppure fermo in una sleep (cioè in stato waiting), riceverà e gestirà il segnale solo al termine dell'I/O o della sleep.**

# Segnali - trap (3)

Ricezione di signal ed esecuzione di azione

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.

**trap      action    lista\_di\_segnali**

ESEMPIO:            eseguire così:            `./lancia_riceviSIGUSR1e2.sh`

## **riceviSIGUSR1e2.sh**

```
#!/bin/bash
trap "echo \"ricevuto SIGUSR1 !!! Termino !!!\"; exit 99"  SIGUSR1

ricevutoSIGUSR2() {
    echo "ricevuto SIGUSR2, continuo";
}
trap ricevutoSIGUSR2 SIGUSR2

while true ; do    echo -n "."; sleep 1; done
```

## **lancia\_riceviSIGUSR1e2.sh**

```
#!/bin/bash
./riceviSIGUSR1e2.sh &
CHILDPID=$!
sleep 5 ; echo "mando SIGUSR2"; kill -s SIGUSR2 ${CHILDPID}
sleep 5 ; echo "mando SIGUSR2"; kill -s SIGUSR2 ${CHILDPID}
sleep 5; echo "mando SIGUSR1"; kill -s SIGUSR1 ${CHILDPID}
sleep 4;
```

# Segnali - trap (4)

La configurazione dell'azione si mantiene-  
Annullare l'esecuzione dell'azione

## riceviSIGUSR1e2.sh

```
#!/bin/bash
trap "echo \"ricevuto SIGUSR1 !!! Termino !!!\"; exit 99" SIGUSR1

NUMRICEVUTISIGUSR2=0
ricevutoSIGUSR2() {
    ((NUMRICEVUTISIGUSR2=${NUMRICEVUTISIGUSR2}+1))
    if ((${{NUMRICEVUTISIGUSR2}}==2)) ; then
        echo "ricevuto 2° SIGUSR2, disabilito gestione SIGUSR2";
        trap ":" SIGUSR2 # non faccio fare piu' niente
    else
        echo "ricevuto ${NUMRICEVUTISIGUSR2}-esimo SIGUSR2, continuo";
    fi
}
trap ricevutoSIGUSR2 SIGUSR2
while true ; do echo -n "."; sleep 1; done
```

## lancia\_riceviSIGUSR1e2.sh

```
#!/bin/bash
./riceviSIGUSR1e2.sh &
CHILDPID=$!
for (( i=0; ${i}<3; i=${i}+1 )) ; do
    sleep 3; echo "mando SIGUSR2"; kill -s SIGUSR2 ${CHILDPID} ;
done
sleep 3; echo "mando SIGUSR1"; kill -s SIGUSR1 ${CHILDPID};
wait ${CHILDPID}
```

-Mando il primo SIGUSR2 per dimostrare che il trap setta la funzione da lanciare alla ricezione del signal.

-Mando il secondo SIGUSR2 per dimostrare che il trap mantiene il settaggio anche dopo l'arrivo di un primo segnale.

-Mando il terzo SIGUSR2 per dimostrare che posso cambiare la routine di gestione anche da DENTRO la routine stessa.



# Attesa terminazione di processo figlio di shell: **wait** (1)

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

Per fare questo devo 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.

```
program.exe arg1 arg2 arg3 .... argN &  
PIDsalvato=$!  
echo "il PID di program.exe e\' ${PIDsalvato} "  
... faccio qualcosa d'altro ...  
wait ${PIDsalvato}
```

**<-- attende terminazione di program.exe**

**NB: il comando wait può essere invocato SOLO dalla shell che ha lanciato il processo di cui vogliamo attendere la terminazione. Se l'attesa per la terminazione viene invocata da una shell diversa, il comando wait termina subito e restituisce 127.**

**NB: il comando wait restituisce come exit status l'exit status restituito dal processo figlio specificato dal PID. Se il PID specificato non corrisponde ad un processo figlio in esecuzione, il comando wait termina subito e restituisce 127.** esempio:

```
sleep 20 &  
wait $! <-- attende fine di sleep 20 e restituisce risultato di sleep
```

# Attesa terminazione di processo figlio di shell: **wait** (2)

Il comando `wait` può 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 risultato il risultato dell'ultimo processo nella lista di 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.

```
(sleep 20 ; exit 3 ) &
```

```
wait $!
```

```
echo "il valore restituito da wait è: " $?
```

**<-- attende terminazione di exit 3**

**<-- visualizza 3**

```
(sleep 20 ; false ) &
```

```
wait $!
```

```
echo "il valore restituito da false è: " $?
```

**<-- attende terminazione di false**

**<-- visualizza 1**

```
progr &
```

```
pid1=$!
```

```
prog2 &
```

```
pid2=$!
```

```
prog3 &
```

```
pid3=$!
```

```
wait ${pid1} ${pid3} ${pid2} <-- attende terminazione dei 3 prog e restituisce l' exit status
```

di prog2 cioè del programma il cui pid compare **per ultimo nell'elenco** 206

# wait, processi Zombie, processi Orfani, processo init (1)

Un processo (padre) genera un processo figlio e lo esegue in background.

Il processo figlio prima o poi **termina** e restituisce un valore intero.

Il processo padre vuole sapere quale è il valore restituito dal figlio per sapere se tutto è andato bene o no. A questo scopo, il processo padre esegue il comando **wait pidfiglio** che attende la terminazione del processo figlio avente pid pidfiglio e restituisce il risultato del processo figlio.

**1. 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 anche il **pid** e il **risultato** del processo figlio. In questo momento il processo figlio è uno **zombie** poiché è terminato ma la sua struttura descrittiva è ancora presente nelle tabelle del sistema operativo.

- **La struttura pcb viene eliminata solo quando il padre invoca il comando wait** (o una system call waitpid che fa la stessa cosa) per attendere la terminazione del figlio. A quel punto il processo figlio sparisce dalle tabelle e il suo pid può essere riutilizzato per altri processi.

**2. Se invece il processo padre termina senza fare la wait per il figlio, il processo figlio diventa orfano**. Quando un processo orfano termina, oppure quando uno zombie diventa orfano perché termina il padre, **l'orfano viene adottato dal processo init**, quello con pid 1 che ha originato tutti i processi al boot. Gli orfani diventano figli di init. **Ogni tanto il processo init chiama la wait sui propri figli**, anche quelli adottati ovviamente, **e così fa rilasciare i pcb degli orfani**.

- Quindi, **morto il padre, i figli che terminano vengono eliminati (no zombie)** **207**

# Processi Zombie, Processi Orfani, processo init (2)

Per quale motivo un processo padre non fa una wait per attendere la terminazione di un processo figlio?

1. o per sbaglio, perché il programmatore se ne è dimenticato,
2. oppure appositamente, per impedire che un altro suo nuovo figlio prenda dal sistema operativo proprio il pid del processo figlio terminato.

In entrambi i casi, si rischia di esaurire i pid disponibili nel sistema operativo.

# killall - Terminazione di tutti i processi con un dato nome

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 al comando killall. Il comando killall per default invia a quei processi il segnale SIGTERM, a meno che non si specifichi un diverso segnale.

In realtà, il comando killall può servire più genericamente ad inviare ai processi un segnale specificato mediante un argomento a riga di comando.

Inoltre, alcuni parametri permettono di individuare in modo diverso i processi a cui inviare il segnale.

Esempio:

Supponiamo che, **nella directory corrente** esistano i file binari eseguibili di nome **attendiA.exe** e **attendiC.exe** e che esista uno script di comandi **attendiB.sh**, e che l'esecuzione di ciascuno di questi duri qualche minuto.

Posso lanciare in background più istanze di ciascun eseguibile/script così:

```
for (( i=1; $i < 10 ; i=$((i+1)) )); do ./attendiA.exe & ./attendiB.sh & done
```

Se voglio, posso terminarli tutti i processi **attendiA.exe** ed **attendiB.sh** usando il seguente comando:

```
killall -r 'attendi[[:upper:]]*'
```

Il comando ordina di usare l'espressione regolare (che segue l'opzione -r) per selezionare il nome dei processi tra attualmente in esecuzione da eliminare.

**Notare che 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.**

# killall - Interpretazione del nome dei processi in memoria

Verifichiamo che la killall interpreta i metacaratteri cercando di **associarli ai nomi dei processi in memoria e non al nome dei file nel filesystem.**

Supponiamo che, **nella directory corrente** esistano i file binari eseguibili di nome **attendiA.exe** e **attendiC.exe** e che esista uno script di comandi **attendiB.sh**, e che l'esecuzione di ciascuno di questi duri qualche minuto.

Posso lanciare in background più istanze di ciascun eseguibile/script così:

```
for (( i=1; $i < 10 ; i=$((i+1)) )); do ./attendiA.exe & ./attendiB.sh & done
```

Ora lanciamo il seguente comando, SI NOTI CHE NON CI SONO APICI SINGOLI:

```
killall -r attendi[[:upper:]]*
```

**Senza gli apici semplici**, che impedirebbero alla bash di interpretare i metacaratteri, l'interpretazione viene **PRIMA** effettuata dalla bash con i nomi dei file della directory corrente e, **SOLO IN UN SECONDO MOMENTO**, viene effettuata dalla killall con i nomi dei processi in memoria.

la bash interpreta i metacaratteri e, in pratica lancerei

```
killall -r attendiA.exe attendiB.sh attendiC.exe
```

Allora, la killall mostra un output che fa capire che ha cercato in memoria il processo attendiC.exe ma non l'ha trovato

```
attendiC.sh: no process found
```

# Precedenze degli operatori

## Separatori di comando, condizionali e non

Gli operatori che delimitano la fine di una riga di comando sono:

; & e l'andata a capo **newline**

Gli operatori che connettono due comandi sono:

&& || ; & e l'operatore speciale |

Gli operatori && e || hanno la stessa precedenza.

Gli operatori & e ; hanno la stessa precedenza.

Gli operatori && e || hanno maggior precedenza degli operatori & e ;

L'operatore speciale | **ha la precedenza maggiore di tutti.**

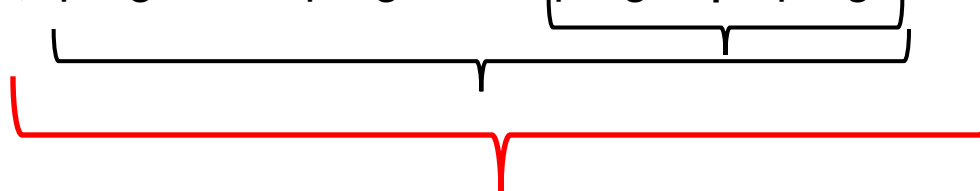
Che significa questo?

Significa che, se scrivo una sequenza di comandi connessi da alcuni separatori,

l'esecuzione dei comandi precede da sinistra verso destra, ma

- Ciascun insieme di comandi connessi da separatori a maggior precedenza va visto come un blocco unico,
- A ciascuno di questi blocchi "connessi da maggior precedenza" va applicato l'effetto dei separatori a minor precedenza presenti sulla destra del blocco.

Esempio: progA ; progB && progC && progD | progE & progF



# Per i puristi

A rigore, l'operatore `|` non dovrebbe essere considerato un operatore come gli altri, poiché l'esecuzione dei comandi che esso connette non è separabile, quindi i due o più comandi connessi dalle `|` dovrebbero essere trattati come un unico comando.

Per semplicità, consideriamo la `|` un operatore a priorità maggiore di `;` e `&` ed anche di `||` e `&&`



# Esempi di Precedenze degli operatori (1)

```
$ echo A && sleep 10 && echo B & echo C
```

```
[1] 30472
```

```
A
```

```
C
```

*qui termina il comando echo C*

```
$
```

*qui passano 10 secondi*

*messaggio di echo B*

```
[1]+ Done
```

```
$ echo A && sleep 10 && echo B
```

Notare che avendo l'operatore & minore precedenza di && accade che l'operatore & viene applicato a tutta la sequenza di comandi

```
$ echo A && sleep 10 && echo B & echo C
```

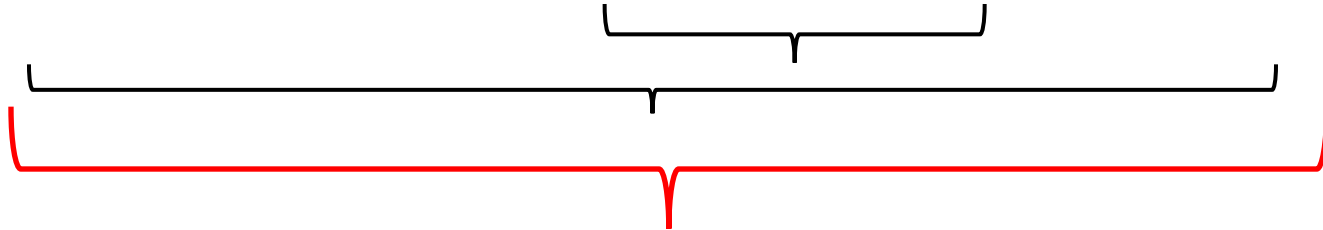
Quindi tutta la sequenza di tre comandi qui sopra viene messa in background mentre l'eseguibile echo C viene eseguito immediatamente.

NB: il \$ che compare a inizio riga e' il prompt della shell.

# Esempi di Precedenze degli operatori (2)

Esempio:

```
$ echo aaA ; echo abB && sleep 10 && echo abC | grep C && echo aaF & echo aaG
```



Ricordando che echo restituisce 0 (cioe' ok),

Allora a video si vedra':

aaA

[1] 42388

aaG

abB

abB non e' stato "greppato"

qui passano 10 secondi prima di vedere qualcosa ....

\$ abC

abC e' passato dal grep

aaF

aaF non e' stato greppato

[1]+ Done

echo abB && sleep 10 && echo abC | grep C && echo aaF

da qui si capisce cosa e' stato lanciato in background

\$

# Modifica di Precedenze degli operatori mediante raggruppamento con { }

```
$ ps ; echo A && sleep 10 && { ps ; echo B & echo C ; }
```

PID	PPID	PGID	WINPID	TTY	UID	STIME	COMMAND
22652	40532	22652	41208	pty1	1001	12:45:12	/usr/bin/ps
40532	41172	40532	40384	pty1	1001	Oct 6	/usr/bin/bash
41172	1	41172	41172	?	1001	Oct 6	/usr/bin/mintty

A

PID	PPID	PGID	WINPID	TTY	UID	STIME	COMMAND
22548	40532	22548	22560	pty1	1001	12:45:22	/usr/bin/ps
40532	41172	40532	40384	pty1	1001	Oct 6	/usr/bin/bash
41172	1	41172	41172	?	1001	Oct 6	/usr/bin/mintty

```
[1] 29540
```

B

C

```
[1]+ Done echo B
```

```
$
```

Notare che nell'output del secondo **ps** rimane la sola shell bash con pid 40532 dovuto al fatto che eseguo il gruppo di comandi nella stessa shell (quella che legge le parentesi graffe) poiche' li racchiudo tra **parentesi graffe**.

**NB: prima della graffa chiusa SERVE il ;**

NB: il \$ a inizio riga e' il prompt

# Modifica di Precedenze degli operatori mediante raggruppamento con { }

**Evidenziamo, con un esempio, che serve il ; prima della graffa chiusa. Invece, se uso parentesi tonde, il ; finale prima della tonda chiusa non e' necessario.**

```
$ $ ps ; echo A && sleep 10 && { ps ; echo B & echo C }  
>  
$
```

Poiche' manca il punto e virgola prima della parentesi graffa chiusa, la shell non capisce dove finisce la riga di comando e chiede all'utente di digitare il resto della riga di comando, visualizzando il prompt |

# Modifica di Precedenze degli operatori mediante raggruppamento con ( )

```
$ ps ; echo A && sleep 10 && ( ps ; echo B & echo C )
```

PID	PPID	PGID	WINPID	TTY	UID	STIME	COMMAND
40532	41172	40532	40384	pty1	1001	Oct 6	/usr/bin/bash
23792	40532	23792	22456	pty1	1001	12:46:05	/usr/bin/ps
41172	1	41172	41172	?	1001	Oct 6	/usr/bin/mintty

A

PID	PPID	PGID	WINPID	TTY	UID	STIME	COMMAND
21968	41428	41428	21804	pty1	1001	12:46:15	/usr/bin/ps
40532	41172	40532	40384	pty1	1001	Oct 6	/usr/bin/bash
41172	1	41172	41172	?	1001	Oct 6	/usr/bin/mintty
41428	40532	41428	41428	pty1	1001	12:46:15	<b>/usr/bin/bash</b>

B

C

\$

**Notare** nell'output del secondo **ps** il processo aggiuntivo **/usr/bin/bash** con pid 41428 dovuto al fatto che eseguo un gruppo di comandi in una sottoshell poiche' li racchiudo tra **parentesi tonde**

Notare che dopo l'ultimo comando dentro le parentesi tonde, posso omettere il ; finale che infatti nell'esempio non compare.

# Occhio al significato di "eseguito nella stessa shell" quando uso { }

Supponiamo che lo script **chiamaps.sh** contenga solo la chiamata a **ps**

```
#!/bin/bash
```

```
ps
```

```
$ ps ; echo A && sleep 10 && { ./chiamaps.sh ; echo C ; }
```

PID	PPID	PGID	WINPID	TTY	UID	STIME	COMMAND
40532	41172	40532	40384	pty1	1001	Oct 6	/usr/bin/bash
41172	1	41172	41172	?	1001	Oct 6	/usr/bin/mintty
17500	40532	17500	44100	pty1	1001	13:19:53	/usr/bin/ps

A

PID	PPID	PGID	WINPID	TTY	UID	STIME	COMMAND
40532	41172	40532	40384	pty1	1001	Oct 6	/usr/bin/bash
41172	1	41172	41172	?	1001	Oct 6	/usr/bin/mintty
<b>17368</b>	40532	17368	44104	pty1	1001	13:20:03	<b>/usr/bin/bash</b>
17424	17368	17368	44064	pty1	1001	13:20:03	/usr/bin/ps

C

```
$
```

**Notare**, nell'output del comando **ps** contenuto nello script **./chiamaps.sh** , il processo aggiuntivo **/usr/bin/bash** con pid **17368** dovuto al fatto che eseguo uno script (per cui serve una subshell) anche se il gruppo di comandi e' racchiuso tra **parentesi graffe** **219**

# Occhio ancora al significato di "eseguito nella stessa shell" quando uso { source ... }

```
$ ps ; echo A && sleep 10 && { source ./chiamaps.sh ; echo C ; }
```

PID	PPID	PGID	WINPID	TTY	UID	STIME	COMMAND
40532	41172	40532	40384	pty1	1001	Oct 6	/usr/bin/bash
19072	40532	19072	17380	pty1	1001	13:25:59	/usr/bin/ps
41172	1	41172	41172	?	1001	Oct 6	/usr/bin/mintty

A

PID	PPID	PGID	WINPID	TTY	UID	STIME	COMMAND
18132	40532	18132	17288	pty1	1001	13:26:09	/usr/bin/ps
40532	41172	40532	40384	pty1	1001	Oct 6	/usr/bin/bash
41172	1	41172	41172	?	1001	Oct 6	/usr/bin/mintty

C

\$

**Notare**, nell'output del comando **ps** contenuto nello script `./chiamaps.sh` , non compare una shell aggiuntiva poiche' ho eseguito lo script `./chiamaps.sh` mediante il comando `source`

# Parameter Expansion – Uso di stringhe (1)

## Estrazione di sottostringhe da variabili

La bash fornisce alcuni operatori che vengono specificati all'interno dei simboli che vengono utilizzati per ottenere il contenuto di una variabile, la variable expansion.

**Tali operatori forniscono una stringa che e' una parte del contenuto della variabile oppure una parte modificata del contenuto della variabile.**

Il contenuto originale della variabile non viene modificato, a meno che io non effettui un assegnamento alla variabile originale.

**Se l'estrazione di parte del contenuto o la sostituzione non sono possibili, gli operatori restituiscono l'intero contenuto della variabile, senza modifiche.**

Gli operatori che effettuano una sostituzione, normalmente ne effettuano una soltanto, anche se fossero possibili piu' sostituzioni. Però qualche operatore consente di effettuare più sostituzioni.

Per effettuare piu' sostituzioni complesse si può però invocare piu' volte il comando salvando di volta in volta il risultato ottenuto.

Anticipiamo alcune definizioni:

definiamo      suffisso=che sta alla fine      prefisso=che sta all'inizio.



# Parameter Expansion – Uso di stringhe (2)

## Estrazione di sottostringhe da variabili

Esempio di uso: estrarre numero tra [] in VAR="[13] qualcosa con [ o ] fine"

Definiamo suffisso=che sta alla fine prefisso=che sta all'inizio.

**pattern** puo' contenere wildcard che cercano di matchare con sottostringhe in VAR

**pattern** puo' contenere anche variabili

```
${VAR%%pattern}
```

```
$ echo ${VAR%%[13]*}
```

Rimuovo il piu' **lungo suffisso** che fa match con stringa orig  
[13] Rimuovo fino alla fine a destra  
la sottostringa **piu' lunga** che inizia con ]

```
${VAR%pattern}
```

```
$ echo ${VAR%[13]*}
```

Rimuovo il piu' **corto suffisso** che fa match con stringa orig  
[13] qualcosa con [ o Rimuovo fino alla fine della stringa originale  
la sottostringa **piu' corta** che inizia con ]

```
${VAR##pattern}
```

```
$ echo ${VAR##[ o ] fine}
```

Rimuovo il piu' **lungo prefisso** che fa match con stringa orig  
Rimuovo, dall'inizio dell'originale, la sottostringa piu' lunga  
che finisce con [

```
${VAR#pattern}
```

```
$ echo ${VAR#[13] qualcosa con [ o ] fine}
```

Rimuovo il piu' **corto prefisso** che fa match con stringa orig  
Rimuovo, dall'inizio dell'originale, la sottostringa piu' corta  
che finisce con [

# Parameter Expansion – Uso di stringhe (3)

## sottostringhe da variabili

es: VAR="alfabetagamma"

`${#VAR}` viene espansa nella stringa che esprime la lunghezza in byte del contenuto di VAR. Con la variabile VAR di esempio produce la stringa "17". Produce la stringa "0" se la variabile VAR non esiste o se è vuota.

`${VAR/pattern/string}` cerca nel contenuto di VAR la sottostringa piu' lunga che fa match con il pattern specificato (con wildcard) e lo **sostituisce** con string. Se sono possibili piu' sostituzioni viene sostituita solo la sottostringa piu' vicina all'inizio.  
ALTRA="vaf"; echo "\${VAR/b\*a/k\${ALTRA}p}"  
produce in output:               alfakvafp

`${VAR:offset}` sottostringa che parte dal offset-esimo carattere del contenuto di VAR  
L'offset del primo carattere e' zero. Gli argomenti vengono valutati aritmeticamente. Lo stesso per il prossimo operatore

`${VAR:offset:length}` sottostringa lunga length che parte dal offset-esimo carattere del contenuto di VAR  
es: VAR="alfabetagamma" ; DA="2" ; FINOA="5" ;  
echo "\${VAR:\${DA}:\${FINOA}+3}" ;  
produce in output:               fabetaga

# Parameter Expansion – Uso di stringhe (4)

## estrazione con sostituzione da variabili

esempio: come effettuare piu' sostituzioni nel contenuto di una variabile

```
#!/bin/bash
VAR="a1BDaxxx2BaDxxx3BbDxx4BcDxxx5BDxxx6BdD"
PREVIOUS=${VAR}
while true ; do
    VAR=${VAR/B?D/ZZZ}
    if [ ${VAR} == ${PREVIOUS} ] ; then
        break ;
    else
        PREVIOUS=${VAR}
    fi
done
echo "VAR=${VAR}"
```

Lo script visualizza

```
VAR=a1BDaxxx2ZZZxxx3ZZZxx4ZZZxxx5BDxxx6ZZZ
```

---

NB: più semplicemente si poteva usare:

`VAR=${VAR//B?D/ZZZ}` (vedi dopo)

# Parameter Expansion – Uso di stringhe (4bis)

## ulteriori dettagli su sostituzione in variabili

L'operatore di sostituzione varia il suo comportamento se inserisco uno dei 3 caratteri speciali subito dopo il primo / dopo il nome di variabile, in particolare:

`${VAR//pattern/string}`      quando dopo il primo / c'e' un altro / allora TUTTE le sottostringhe che fanno match con il pattern specificato vengono sostituite, non solo la prima.  
esempio: sostituisce tutti i cane con gatto `{VAR//cane/gatto}`  
esempio: sostituisce tutti gli asterischi \* con \\* per impedire che il contenuto della variabile possa essere interpretato.  
`${VAR/\\*/\\*}`

`${VAR/#pattern/string}`      quando dopo il primo / c'e' un # allora il pattern viene sostituito SOLO SE si trova all' INIZIO della variabile.

`${VAR/%pattern/string}`      quando dopo il primo / c'e' un % allora il pattern viene sostituito SOLO SE si trova alla FINE della variabile.

# Parameter Expansion – Uso di stringhe (5)

## operatori vari su variabili

Espansione verso **nomi** di variabili corrispondenti ad un prefisso pattern

`${!VarNamePrefix*}` restituisce un elenco con tutti i nomi delle variabili  
il cui nome inizia con il prefisso specificato VarNamePrefix

Esempio: se esistono le seguenti variabili

```
BASH=/bin/bash
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSION='4.1.17(9)-release'
CYG_SYS_BASHRC=1
ed eseguo il comando
echo ${!BASH_AR*}
vedro' in output
BASH_ARGC BASH_ARGV
```

# Confronto tra prestazioni

## Meglio comandi built-in che lanciare eseguibili

```
# file  usa_bash.sh
```

```
SUM=0
for (( i=0; i<10000; i++ )) ; do
OUT="tiglio gatto gatto pino
pesco pesce gatto zebra
iena tigre pera gatto"
OUT=${OUT//gatto/cane}
(( SUM=SUM+${#OUT} ))
done
echo ${SUM}
```

```
# file  usa_sed2.sh
```

```
SUM=0
for (( i=0; i<10000; i++ )) ; do
OUT="tiglio gatto gatto pino
pesco pesce gatto zebra
iena tigre pera gatto"
OUT=`echo ${OUT} | sed 's/gatto/cane/g'`
(( SUM=SUM+${#OUT} ))
done
echo ${SUM}
```

Confronto in esecuzione

```
$ time ./usa_bash.sh
650000
```

```
real    0m0,110s
user    0m0,107s
sys     0m0,000s
```

```
$ time ./usa_sed2.sh
650000
```

```
real    0m14,142s
user    0m11,904s
sys     0m2,072s
```

Lanciare eseguibili consuma tempo di CPU  
circa 100 volte in più che usare comandi  
built-in.

Notare anche il maggior tempo di CPU  
impiegato dal kernel.

# APPUNTI DA RISISTEMARE:

## escludere directory da ricerca con find

escludo da find la ricerca nella directory ./BUTTA

```
find ./ -path ./BUTTA -prune -o -type f -name "*.c" -exec grep -H rand '{}' \; |  
more
```

escludo da find la ricerca nella directory ./ESAMI e anche nella ./BUTTA

```
find ./ -path ./ESAMI -prune -o -path ./BUTTA -prune -o -type f -name "*.c" -  
exec grep -H rand '{}' \;
```

# APPUNTI DA RISISTEMARE: Follie bash ovvero Scorciatoie

**!!** riesegue l'ultimo comando appena eseguito

**!n** riesegue l'ennesimo comando presente nella storia, dove 'n' e' il numero del comando da rieseguire

**!stringa** riesegue l'ultimo comando che inizia con i caratteri specificati in 'stringa'

**!stringa:p** visualizza l'ultimo comando che inizia con i caratteri specificati in 'stringa'

**!?comando?** ricerca il comando specificato tra punti interrogativi

**fc 4** permette di modificare in comando numero 4 con l'editor predefinito (solitamente vi)

**fc -e 'nome' 4** permette di modificare in comando numero 4 con l'editor 'nome' specificato

**^comando1^comando2** riesegue l'ultimo comando eseguito che contiene la parola 'comando1' sostituendola con 'comando2'.

**history** visualizza l'elenco di tutti i comandi eseguiti

**CTRL-U** cancella tutta la riga dalla posizione del cursore all'inizio della riga

**CTRL-K** cancella tutta la riga dalla posizione del cursore alla fine della riga

**CTRL-W** cancella una parola dalla posizione del cursore all'inizio della riga

**ALT-D** cancella una parola dalla posizione del cursore alla fine della riga

**freccia sinistra** sposta il cursore di un carattere a sinistra

**freccia destra** sposta il cursore di un carattere a destra

**freccia su** scorre la storia dei comandi a ritroso

**freccia giu'** scorre la storia dei comandi in avanti

**tasto home** sposta il cursore all'inizio della riga

**CTRL-A** sposta il cursore all'inizio della riga

**tasto fine** sposta il cursore alla fine della riga

**CTRL-E** sposta il cursore alla fine della riga

**ALT-F** sposta il cursore alla fine della parola successiva (F sta per forward, successivo)

**CTRL-B** sposta il cursore all'inizio della parola precedente (B sta per backward, precedente)

**CTRL-T** inverte gli ultimi due caratteri a sinistra del cursore, cioe' ab diventa ba (T sta per transpose)

**ALT-T** inverte le ultime due parole a sinistra del cursore, cioe' cp pippo pluto diventa cp pluto pippo

**ALT-U** trasforma in maiuscolo la parola su cui si trova il cursore (U sta per uppercase, cioe' maiuscolo)

**ALT-L** trasforma in minuscolo la parola su cui si trova il cursore (L sta per lowercase, cioe' minuscolo)

**tasto TAB** espande il nome di un file o di un comando