La Shell Unix

SISTEMA OPERATIVO UNIX

Nasce nei Bell Laboratories dell'AT&T e poi evolve in varie versioni:

- **System V**, sviluppato dall'AT&T
- **BSD 4.x** (**B**erkeley **S**oftware **D**istribution)
- **Linux**

Caratteristiche principali:

- **sistema multiutente**
- **sistema time-sharing**
- **memoria virtuale**

Architettura e strumenti:

- processore comandi (**shell**)
- linguaggio di sistema (**C)**
- nucleo (**primitive di sistema**)

utente	**programmatore**
processore comandi (**shell**)	linguaggio di sistema (**C)**
nucleo (**primitive di sistema**)	

UNIX

In UNIX ogni **attività** è delegata a un **processo** di *utente applicativo* o di *sistema* che la esegue accedendo alle **risorse virtuali** del sistema.

Tra le risorse, fondamentale la **risorsa memoria virtuale,** un'astrazione di memoria più grande della memoria centrale (la memoria fisica, la RAM) disponibile.

Altra **risorsa virtuale** fondamentale è il **file system** che ci rappresenta la macchina virtuale completa.

FILE SYSTEM UNIX

- FILE COME **SEQUENZA DI BYTE**
  NON sono pensate organizzazioni logiche o accessi a record

- FILE SYSTEM **gerarchico**
  ALBERO di sottodirettori

- OMOGENEITÀ dispositivi e file
TUTTO è file

FILE

astrazione unificante del sistema operativo

- file ordinari
- file direttori, accesso ad altri file
- **file speciali (dispositivi fisici), contenuti nel direttorio /dev**

ORGANIZZAZIONE del FILE SYSTEM

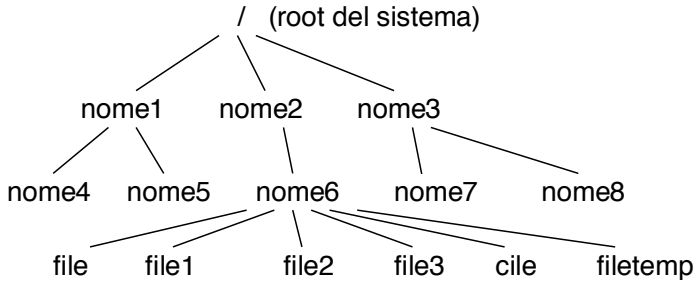
NOMI di file
- **ASSOLUTI**: espressi a partire dalla radice
  /nome2/nome6/file
- **RELATIVI**: dal direttorio corrente
  nome6/file

I nomi (e la sintassi in generale) sono **case-sensitive**

Si possono usare abbreviazioni nei nomi (**wild card**)
***** per una qualunque stringa
? per un qualunque carattere

Direttorio corrente identificato da **.**
Padre del direttorio corrente identificato da **..**

Ogni utente ha un direttorio a default
direttorio in cui l'utente si colloca all'ingresso nel sistema



file* ==> file, file1, file2, file3, filetemp
file? ==> file1, file2, file3
*ile ==> file, cile

Esempio: Il File System di Linux

Nell'uso (e amministrazione) di un sistema Unix è molto importante conoscere e rispettare le convenzioni del file system. Ad esempio, per un sistema Linux si ha:

/bin	Comandi principali del sistema
/boot	File per il boot del sistema (kernel e bootloader)
/dev	Dispositivi (file che rappresentano le periferiche del sistema)
/etc	File di configurazione a livello di sistema
/home	Cartelle personali degli utenti
/lib e/o /lib64	Librerie di sistema
/media e/o /mnt	Per montare dischi e/o chiavette USB
/opt	Per l'installazione di programmi opzionali o secondari
/proc e /sys	File speciali per il monitoraggio e/o la riconfigurazione on-the-fly del sistema
/root	Cartella dell'utente root
/sbin	Eseguibili di base riservati all'amministratore
/tmp	File temporanei
/usr	Contiene la maggior parte dei programmi installati sul sistema
/var	Dati di dimensione variabile ad uso di vari servizi (es. server Web, DNS, Email)

Si veda il Linux Filesystem Hierarchy Standard all'indirizzo: http://www.pathname.com/fhs/

Accesso al Sistema Operativo Unix

Un sistema multi-utente deve gestire correttamente l'accesso di diversi utenti alla macchina.

Quindi **autenticazione** (login) per controllare l'accesso di un utente al sistema e corretta **protezione** delle risorse (file) dei singoli utenti.

Autenticazione in Unix con username/password

Username: cesare
Password: **********

L'uscita avviene mediante logout, tramite uno dei comandi:

- **exit**
- **logout**
- **CTRL-D**

Durante una sessione di lavoro, i comandi dati dall'utente sono interpretati da una **shell (interprete dei comandi)**

Esiste un utente, **root**, privilegiato rispetto agli altri:
il gestore del sistema (detto anche super-user) che non ha limiti di azione, non è sottoposto alle regole di sicurezza di tutti gli altri utenti.

Le credenziali dell'utente (username e passowrd) sono memorizzate in un file di sistema chiamato /**etc/passwd**, che costituisce il "database" degli utenti registrati su quel sistema (con tutte le loro caratteristiche, compreso il tipo di shell che l'utente vuole utilizzare).

FORMATO DEL FILE /etc/passwd

utente:password:UID:GID:commento:direttorio:comando

utente ➔ nome che bisogna dare al login
password ➔ password che occorre dare al login **(memorizzata in forma cifrata)**

UID ➔ **User IDentifier**
numero che identifica in modo univoco l'utente nel sistema

GID ➔ **Group IDentifier**
numero che identifica il gruppo di cui fa parte l'utente

commento ➔ campo di commento

direttorio ➔ direttorio iniziale in cui si trova l'utente al login (home directory)

comando ➔ comando che viene eseguito automaticamente all'atto del login **(in genere è la shell scelta da utente)**

ESEMPIO:

root:.XPc4HKe0nPQA:0:1:Operator:/:/bin/sh
cesare:rTIW65BOuQ9ng:230:30:Cesare Stefanelli:/home/cesare:/bin/sh

PROTEZIONE dei File

Molti utenti ➔ Necessità di regolare gli ***ACCESSI*** alle informazioni

Per un file, si specificano 3 classi di utilizzatori:
- il proprietario, **owner (o user)**
- il gruppo del proprietario, **group**
- tutti gli altri utenti, **others**

Per ogni tipo di utilizzatore, si distinguono tre modi di accesso al file:
- **lettura (r)**
- **scrittura (w)**
- **esecuzione (x)**

Ogni utente è caratterizzato da:
- l'identificatore di utente (**UID, user ID**) e
- l'identificatore di gruppo (**GID, group GID**)

Ogni file è marcato con
- lo User-ID e il Group-ID **del proprietario**
- un insieme di 12 bit di protezione

12	*11*	*10*		*9*	*8*	*7*		*6*	*5*	*4*		*3*	*2*	*1*
0	0	0	I	1	1	1	I	1	0	0	I	1	0	0
SUID	SGID	sticky	I	R	W	X	I	R	W	X	I	R	W	X
				Owner				**G**roup				**O**thers		

SIGNIFICATO DEI 12 BIT

12	*11*	*10*		*9*	*8*	*7*		*6*	*5*	*4*		*3*	*2*	*1*
0	0	0	I	1	1	1	I	1	0	0	I	1	0	0
SUID	SGID	sticky	I	R	W	X	I	R	W	X	I	R	W	X
				Owner				**G**roup				**O**thers		

9 bit più a destra (bit 1-9) ➔ 3 triple di permessi:
- **lettura / scrittura / esecuzione**

rispettivamente per
- **il proprietario / quelli del suo gruppo / gli altri**

dodicesimo bit
SUID (Set-User-ID) (identificatore di utente effettivo)

Si applica a un file di **programma eseguibile**
Se vale 1, fa sì che *l'utente che sta eseguendo* quel programma venga considerato *il proprietario* di quel file (solo per la durata della esecuzione)

Questo bit è necessario per consentire, durante l'esecuzione di quel programma, operazioni di lettura/scrittura su file di sistema, che l'utente non avrebbe il diritto di leggere / modificare.

Esempio: **mkdir** crea un direttorio, ma per farlo deve anche modificare alcune aree di sistema (file di proprietà di **root**), che non potrebbero essere modificate da un utente. Solo il SUID lo rende possibile.

SGID bit: come SUID bit, per il gruppo
sticky bit

il sistema cerca di mantenere in memoria l'immagine del programma, anche se non è in esecuzione

I Processi in Unix

Definizione di processo

Il termine processo viene usato per indicare un **programma in esecuzione**

Esecuzione sequenziale del processo; istruzioni eseguite una dopo l'altra

Differenza tra processo e programma

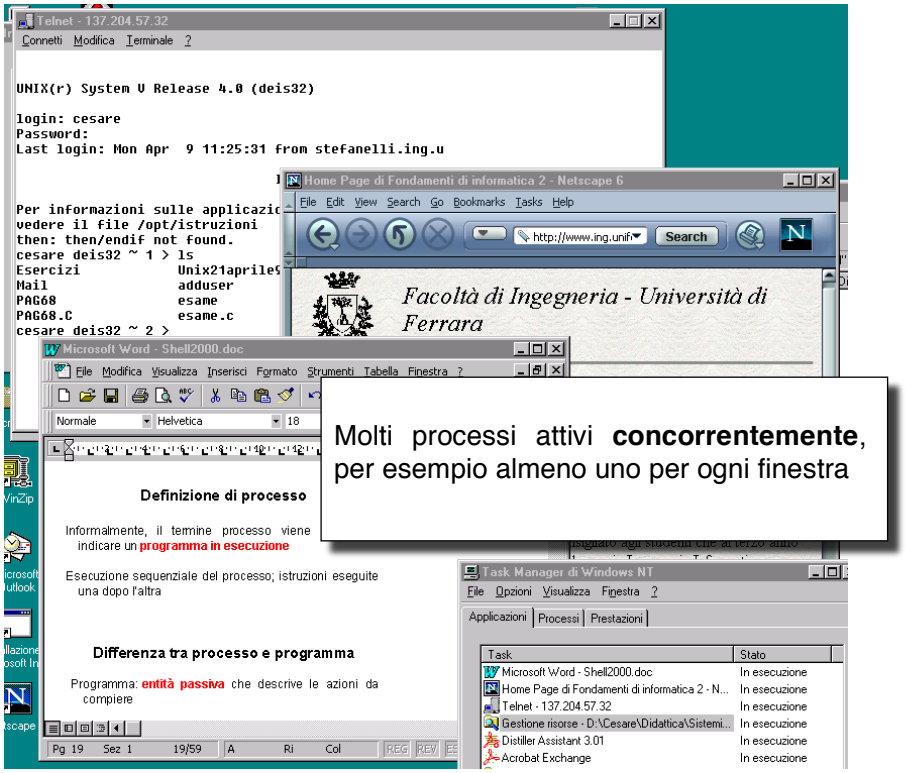
Programma: **entità passiva** che descrive le azioni da compiere

Processo: **entità attiva** che rappresenta l'esecuzione di tali azioni

I processi in un sistema concorrente

Un sistema operativo svolge **molte attività allo stesso tempo** (per es. gestire una stampante, più terminali e varie applicazioni).

Presenza di molte attività (processi) **concorrenti**



I processi in un sistema concorrente

Tutti i sistemi operativi moderni sono **concorrenti**, con molti processi in esecuzione "contemporaneamente".

È come se ci fosse un processore diverso per ognuno dei processi attivi nella macchina. In realtà, questo è reso possibile dall'utilizzo della CPU in **time-sharing**. La CPU viene assegnata ciclicamente a tutti i processi attivi, ma solo per un breve intervallo di tempo. L'elevata velocità del processore fornisce l'illusione di reale parallelismo nell'esecuzione di tutti i processi.

Attenzione a non fare confusione con i termini concorrente, time-sharing, multitasking, parallelo.

I modelli di processo in un sistema concorrente

Il sistema operativo deve garantire che i diversi processi concorrenti (in esecuzione "contemporaneamente" su una stessa macchina) **non interferiscano** tra loro.

Inoltre, i processi devono anche comunicare tra di loro. La soluzione a questo problema è diversa a seconda del **modello di processo** adottato.

Modello ad **ambiente locale**, detto anche a **scambio di messaggi**, in quanto diversi processi possono comunicare tra loro solo attraverso lo scambio di messaggi (es. pipe). Nessuna condivisione di aree di memoria. (es. **Unix**)

Modello ad **ambiente globale**, detto anche a **memoria comune**, in cui diversi processi possono comunicare tra loro utilizzando una memoria di lavoro comune (cioè delle variabili condivise). (es. i thread in **Java**).

Attenzione alle interferenze nell'accesso ad aree di memoria comuni a più thread.

I processi in Unix

I processi Unix sono basati sul modello ad **ambiente locale**, per cui sono tutti indipendenti tra loro.

Ogni processo Unix ha un proprio **spazio di indirizzamento separato** da quello degli altri (ogni processo vede le proprie variabili e solo lui può modificarle).

Si noti la "**protezione**" dello spazio di indirizzamento di un processo da interferenze provenienti da altri processi.

Ogni processo Unix è **univocamente identificato** da un nome, che è il **PID** (Process IDentifier), definito come un intero maggiore di zero e assegnato autonomamente dal Sistema Operativo al momento della nascita del processo.

Esempio:

Si provi a lanciare lo stesso comando (per es. ls) su due finestre differenti su una stessa macchina. Il sistema produce due diversi processi che mettono in esecuzione il comando ls, senza interferenze tra loro.

Si noti che il codice del comando ls è eseguito contemporaneamente da due processi senza errori e interferenze. Un codice che si comporta in questo modo è detto ***rientrante***.

Principali comandi

SHELL ovvero
il **PROCESSORE COMANDI**

Lo shell come interprete dei comandi:
esegue uno dopo l'altro i comandi forniti

loop forever
<**accetta** comando da console>
<**esegui** comando>
end loop;

Lo shell è un **processore comandi**, che **accetta comandi** da terminale o da un file comandi e li **esegue** fino alla fine del file
(si può usare <CTRL><D>)

loop forever
< LOGIN >
repeat
<accetta comando da console>
<esegui comando>
until <fine file>
< LOGOUT >
end loop

Maggiori capacità rispetto a un semplice esecutore di un comando alla volta.

COMANDI RELATIVI AL FILE SYSTEM

Sintassi:

comando [**-opzioni**] [**argomenti**] <INVIO>

Un comando termina con un INVIO, oppure è separato con **;** da altri comandi nella stessa linea

CREAZIONE / GESTIONE DI DIRETTORI

mkdir	<nomedir>	*creazione di un nuovo direttorio*
rmdir	<nomedir>	*cancellazione di un direttorio*
cd	<nomedir>	*cambio di direttorio*
pwd		*stampa il direttorio corrente*
ls	[<nomedir>]	*visualizz. contenuto del direttorio*

TRATTAMENTO FILE

ln	<vecchionome> <nuovonome>	*link*
cp	<filesorgente> <filedestinazione>	*copia*
mv	<vecchionome> <nuovonome>	*rinom. / spost.*
rm	<nomefile>	*cancellazione*
cat	<nomefile>	*visualizzazione*

Esempi di comandi:

cd /nome2/nome6
cat file
ls /nome1
rm nome2.txt

Il comando ls per listare il contenuto di una directory

Varie opzioni:

ls -a [<nomedir>]
visualizza file "nascosti" (nome che inizia con '**.**')

ls -l [<nomedir>]
mostra tutte le informazioni per i file (tipo del file, permessi, numero link, proprietario...)

ls -la [<nomedir>]
è l'unione delle precedenti

ls -F
tutte le informazioni dei file visualizzando i file normali con il loro nome, gli eseguibili con nome e suffisso, i direttori con nome e suffisso **/**

ls -d
i direttori sono visualizzati come i file, senza entrare nel contenuto e listarlo

ls -R <nomedir>
esame ricorsivo della gerarchia a partire da nomedir

ls -l [<nomedir>]
tutte le informazioni per i file, permessi, tipo del file, numero link, proprietario, ora modifica, nome

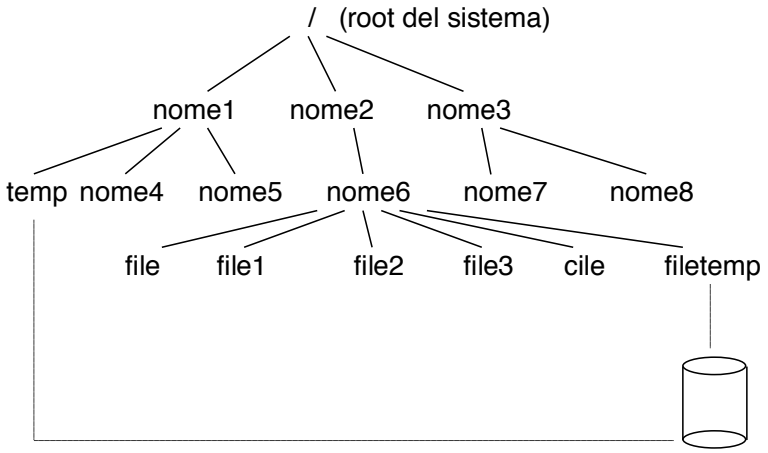
ls -i
lista gli i-number dei file con le altre informazioni

ls -r
lista i file in ordine opposto al normale ordine alfabetico

ls -t
lista i file in ordine di ultima modifica, dai più recenti, fino ai meno recenti

Comando ln, link

Le informazioni contenute in uno stesso file possono essere **visibili come file diversi**, tramite “riferimenti” (link) allo stesso file fisico.



Il sistema considera e tratta il tutto:

- se un file viene cancellato, le informazioni sono veramente eliminate solo se non ci sono altri link a esso

comando: ln [-s]

ln /nome2/nome6/filetemp /nome1/temp

Si vedano i link di ogni direttorio

PROTEZIONE e DIRITTI SUI FILE

Per variare i bit di protezione:

chmod [**u g o**] [**+ -**] [**rwx**] <nomefile>

(u sta per user, il proprietario del file)

I permessi possono essere concessi o negati dal ***proprietario del file***

Esempio di variazione dei bit di protezione:

chmod 0755 /usr/dir/file

0	0	0	l	1	1	1	l	1	0	1	l	1	0	1
SUID	SGID	sticky	l	R	W	X	l	R	W	X	l	R	W	X
				User				**G**roup				**O**thers		

Ad esempio:

chmod u-w fileimportante

Altri comandi:

chown <nomeutente> <nomefile>

chgrp <nomegruppo> <nomefile>

cat

Il comando **cat** stampa a video il contenuto di un file:

```
cstefanelli@linuxdid:~$ cat /etc/passwd
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
......
```

more (e less)

il cat effettua una semplicissima stampa del contenuto a video di un file. Per una visualizzazione a pagine, certamente più comoda per file di larghe dimensioni, è necessario utilizzare un comando di paginazione (come more).

Il comando **more** permette di visualizzare un file una pagina per volta.

Esistono diverse alternative più recenti e potenti al comando more, tra cui va sicuramente segnalato il comando less (che permette anche di far scorrere la visualizzazione all'indietro).

echo

Il comando **echo** stampa a video la stringa passatagli come parametro:

```
cstefanelli@linuxdid:~$ echo Hello world!
Hello world!
```

Il comando echo rappresenta la primitiva fondamentale che la shell mette a disposizione per stampare informazioni a video.

sort

Il comando sort riordina le righe di un file in ordine alfabetico

Esempi e opzioni:
sort nomefile.txt *# elenca le righe del file nomefile.txt in # ordine alfabetico crescente*
sort –r nomefile.txt *# elenca le righe del file nomefile.txt # in ordine alfabetico decrescente*

Il comando sort ha molte opzioni:

-o <nomefileout> stampa su file

-n interpreta le righe del file come numeri

-k <n> ordina il file secondo il contenuto della n-esima colonna

diff

Il comando diff stampa la differenza tra il contenuto di due file.

Esempio:
diff <file1> <file2> *mostra (solo) le righe diverse*

Il comando diff è molto utilizzato nella gestione del codice sorgente.

wc

Il comando wc stampa il numero di righe, parole, o caratteri contenuti in un file.

Esempi e opzioni:

wc -l nomefile.txt *# conta le linee (opzione **l**) contenute*
 # nel file nomefile.txt
wc -w nomefile.txt *# conta le parole (opzione **w**)*
 # contenute nel file nomefile.txt
wc -c nomefile.txt *# conta i caratteri (opzione **c**) contenuti*
 # nel file nomefile.txt

grep

Il comando grep seleziona le righe di un file che contengono la stringa passata come parametro e le stampa a video.

Esempi:

grep stringa nomefile.txt *# seleziona le righe del file*
 # nomefile.txt che contengono
il testo “stringa” e le stampa
 # a video

grep –c stringa nomefile.txt *# conta le righe del file*
 # nomefile.txt che contengono
il testo *“stringa” e stampa il*
 # numero a video

grep –r stringa dir *# seleziona le righe di tutti i file*
nella directory dir che contengono il
testo “stringa” e le stampa a video.
ricorsivo sulle sotto directory.

Il comando grep ha numerosissime opzioni, ed è utilizzatissimo, specialmente nella gestione e manipolazione di file di configurazione e di codice sorgente.

head

Il comando head mostra le prime righe di un file.

Esempi e opzioni:

head –n 15 nomefile.txt # mostra le prime 15 righe
del file

head –c 30 nomefile.txt # mostra i primi 30 caratteri
del file

tail

Il comando tail mostra le ultime righe di un file.

Esempi e opzioni:

tail –n 15 nomefile.txt # mostra le ultime 15 righe
del file

head –c 30 nomefile.txt # mostra gli ultimi 30 caratteri
del file

time

Il comando time cronometra il tempo di esecuzione di un comando.

Esempi:

```
cstefanelli@linuxdid:~$ time ls foto.jpeg
foto.jpeg
ls foto.jpeg  0,00s user 0,00s system 0%
cpu 0,002 total
```

who

Il comando who mostra gli utenti attualmente collegati al sistema (ovverosia che hanno eseguito il login sulla macchina).

Esempio:

```
cstefanelli@linuxdid:~$ who
cstefanelli :0         2013-12-15 11:01
cstefanelli pts/0      2013-12-15 11:02 (:0)
cstefanelli pts/1      2013-12-15 12:14 (:0)
cstefanelli pts/3      2013-12-16 18:19 (:0)
```


man

In ambiente Unix, il comando man rappresenta l'help di sistema.

Esempio:
man grep *# mostra l'help (guida) del comando grep*

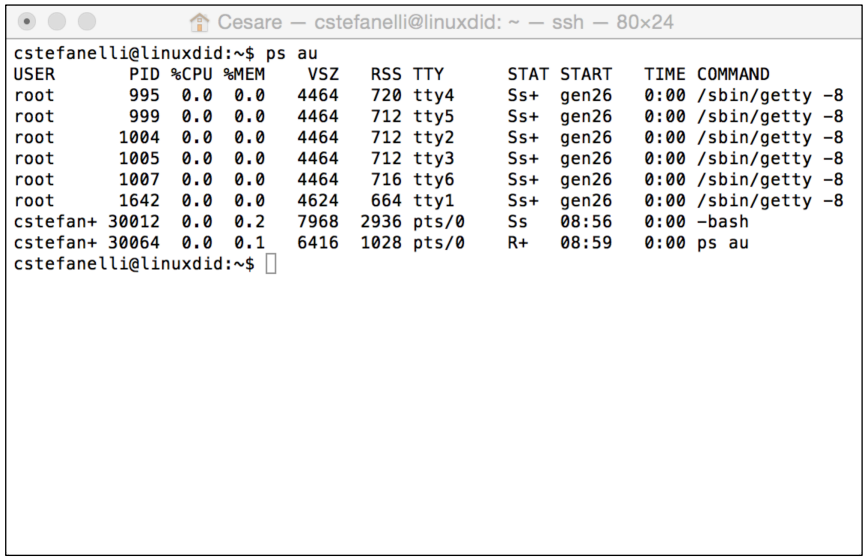
Oltre al comando man, le moderne distribuzioni di Linux mettono a disposizione anche il comando **info**.

ps

Il comando ps mostra la lista dei processi attualmente in esecuzione.

Ecco l'output del comando:

```
cstefanelli@linuxdid:~$ ps au
```



Cosa è il PID?

ps

Alcune opzioni importanti:

a	Mostra anche i processi degli altri utenti
u	Fornisci nome dell'utente che ha lanciato il processo e l'ora in cui il processo è stato lanciato
x	Mostra anche i processi senza terminale di controllo (processi demoni)

Attenzione: si ricordi di non usare il trattino (-) per specificare le opzioni del comando ps, in quanto ps adotta la sintassi di tipo "extended BSD" che non prevede l'uso di trattini.

Se si vuole un aggiornamento continuo e periodico dello stato dei processi correnti, si usi il comando top.

top

Il comando top fornisce una visione dinamica in real-time del sistema corrente. Esso visualizza continuamente informazioni sull'utilizzo del sistema (memoria fisica e virtuale, CPU, ecc.) e sui processi che usano la maggiore share di CPU.

Esiste anche una versione più avanzata e moderna di top, chiamata ***htop***. Al contrario di top, di solito htop non è incluso tra le utility di base disponibili sulle macchine Linux e va esplicitamente installato.

pgrep e pkill

pgrep restituisce i pid (process id) dei processi che corrispondono alle caratteristiche richieste (nome processo, utente, ecc.).

Ad esempio, per ottenere il pid del processo Skype:

```
cstefanelli@linuxdid:~$ pgrep skype
9280
```

Per ottenere invece il pid del processo demone sshd appartenente all'utente root:

```
cstefanelli@linuxdid:~$ pgrep -u root sshd
3488
```

pkill presenta la stessa interfaccia comandi di ***pgrep***, ma anziché stampare a video il pid dei processi, li uccide (in realtà invia loro un segnale SIGTERM).

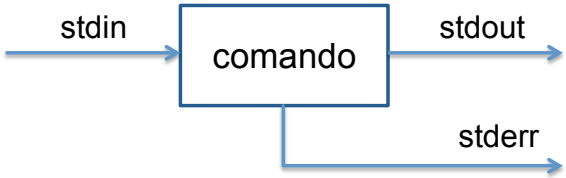
Ridirezione
Piping
Sostituzione

RIDIREZIONE DELL'I/O

I comandi UNIX si comportano come "**filtri**".

Un **filtro** è un programma che riceve in ingresso da un input e produce risultati su output (uno o più). Di default tutti i programmi (processi!) hanno alcuni collegamenti predefiniti:

standard input (stdin) ➔ tastiera
standard output (stdout) ➔ video
standard error (stderr) ➔ video



I filtri operano sull'input considerandolo a linee.

RIDIREZIONE DELL'I/O

L'associazione di default tra stdin-tastiera, stdout-video, stderr-video può essere modificata, ridirigendo quindi da/verso un qualunque file, **senza cambiare il comando.**

ridirezione dell'input (dello stdin)

<comando> **<** <fileinput>

ridirezione output (dello stdout)

<comando> **>** <fileoutput> *riscrive il fileoutput*

<comando> **>>** <fileoutput> *appende a fileoutput*

Alcuni esempi di ridirezione:

ls -lga > file
sort < file > filetemp
sort > filetemp
more < filetemp file

Si ricordi la completa OMOGENEITÀ fra dispositivi e file in Unix:
echo invio messaggio > /dev/ttys000

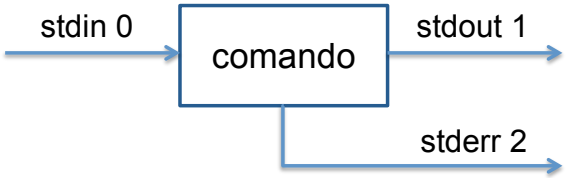
comando **>** file1 **<** file2 **>** file3 **<** file 4 **>** file5
Il comando esegue con stdin da file4 e stdout su file5
EFFETTI COLLATERALI distruzione dei file file1 e file3

RIDIREZIONE DELL'I/O

Ogni file in Unix è identificato da un file descriptor (una specie di puntatore a un file), costituito da un numero intero maggiore di zero (li studieremo in dettaglio nella parte di programmazione di sistema).

In particolare si noti che:

Standard input (stdin) identificato da file descriptor **0**.
Standard output (stdout) è identificato da file descriptor **1**.
Standard error (stderr) è identificato da file descriptor **2**.



Si noti che scrivere <comando> **<** <fileinput>
è la versione sintetica di <comando> **0<** <fileinput>

Si noti che scrivere <comando> **>** <fileoutput>
è la versione sintetica di <comando> **1>** <fileoutput>

E se voglio ridirigere lo stderr? <comando> **2>** <logfile>

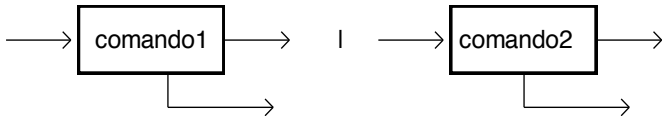
E se voglio ridirigere lo stderr assieme allo stdout? <comando> **2>&1**

ESEMPIO: ls > fileout 2>&1

PIPE

COLLEGAMENTO AUTOMATICO di comandi

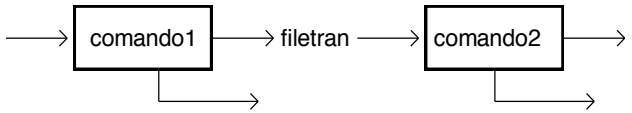
La PIPE *(tubo)* **collega**
l'***output*** di un comando con l'***input*** del successivo



<comando1> **|** <comando2>

In **UNIX** ➔ PIPE come **costrutto concorrente** (ogni comando è mappato in un **processo** separato). Mentre un comando produce l'output, l'altro lo consuma.

In **DOS** ➔ PIPE erano realizzate con **file temporanei**



<comando1> > <filetran> ; <comando2> < <filetran>

Esempi

who | wc -l *conta gli utenti collegati*

ls -R | more

cat file | grep stringa | sort > uscita

RIENTRANZA DELLA SHELL

Una shell è un programma che esegue i comandi, forniti da terminale o da file.

Si invocano le shell come i normali comandi eseguibili con il loro nome

sh [<filecomandi>]
csh [<filecomandi>]

Le invocazioni attivano un processo che esegue la shell.

Le shell sono RIENTRANTI:

più processi possono condividere il codice senza errori e interferenze

sh
sh
csh
ps # quanti processi si vedono?

METACARATTERI

La SHELL riconosce caratteri speciali, detti metacaratteri o wildcard.
I metacaratteri permettono alla Shell di eseguire il pattern matching tra una stringa e i nomi di file presenti nel direttorio corrente.

***** una qualunque stringa di zero o più caratteri in un nome di file

? un qualunque carattere in un nome di file

[*ccc*]
un qualunque carattere, in un nome di file, compreso tra quelli nell'insieme. Anche **range** di valori: **[*c-c*]**

Per esempio **ls [q-s]*** lista i file con nomi che iniziano con un carattere compreso tra q e s

commento fino alla fine della linea

**** escape (segnala di *non* interpretare il carattere successivo come speciale)

echo * *stampa tutti i nomi di file del direttorio corrente*

echo * *stampa il carattere asterisco '*'*

ls [a-p,1-7]*[c,f,d]?
elenca i file i cui nomi hanno come iniziale un carattere compreso tra 'a e 'p' oppure tra 1 e 7, e il cui penultimo carattere sia 'c', 'f', o 'd'

ls *[!0-9]* lista i file del direttorio corrente i cui nomi non contengono alcun carattere numerico

ESECUZIONE di COMANDI in SHELL

In UNIX alcuni comandi sono eseguiti direttamente (comandi built-in), molti comandi sono invece eseguiti da una nuova shell

La **shell attiva** mette in esecuzione una **seconda shell che:**

- esegue le sostituzioni dei metacaratteri e dei parametri
- cerca il comando
- esegue il comando

La **shell padre** attende il completamento dell'esecuzione della sotto-shell (comportamento **sincrono**)

SCHEMA di un PROCESSORE COMANDI

procedure **shell** (ambiente, filecomandi);
< eredita ambiente (esportato) del padre, via copia>
begin
repeat
<leggi comando da filecomandi>
if < è **comando built-in**> *then*
<modifica direttamente ambiente>;
else if <è **comandoeseguibile**> *then*
<esecuzione del comando via nuova shell>
else if <è **nuovofilecomandi**> *then*
shell (ambiente, nuovofilecomandi);
else < errore>;
endif
until <fine file>
end shell;

Per abortire il comando corrente: CTRL-C

Non creare un file comandi ricorsivo senza una condizione di terminazione !!

Variabili nella shell

Ogni shell definisce:

- un insieme di variabili (trattate come stringhe) con **nome** e **valore**
- i riferimenti ai valori delle variabili si fanno con **$nomevariabile**
- si possono fare **assegnamenti**

nomevariabile=$nomevariabile
left-value *right-value*

Esempi:

```
x=123abc        # non si devono usare blank
echo $x         # visualizza 123abc
```

AMBIENTE DI SHELL (environment):
*insieme di **variabili di shell** (scritte in maiuscolo),* per esempio:

- una variabile registra il **direttorio corrente (PWD)**
- ogni utente specifica come fare la *ricerca dei comandi* nei vari direttori del file system: variabile **PATH** indica i direttori in cui cercare
- la variabile **HOME** indica il direttorio di accesso iniziale.

Sostituzioni della shell (parsing)

Prima della esecuzione, il comando viene scandito (*parsing*), alla ricerca di caratteri speciali (*, ?, $, >, <, l, etc.)

Come prima cosa, lo shell prepara i comandi come filtri:
ridirezione e piping di ingresso uscita
(su file o dispositivo)

Nelle successive scansioni, se la shell trova altri caratteri speciali, produce delle *sostituzioni*

1) **Sostituzione dei comandi**
I comandi contenuti tra ` ` (backquote) sono ***eseguiti*** e ne viene prodotto il risultato

```
echo `pwd`    # stampa il direttorio corrente
`pwd`         # tenta di eseguire /home/cesare
`pwd`/fileexe.exe    # e così ?
```

2) **Sostituzione delle variabili e dei parametri**
I nomi delle variabili (**$**nome) sono ***espansi*** nei valori corrispondenti

```
x=alfa          # non si devono usare blank
echo $x         # produce alfa
```

3) **Sostituzione dei nomi di file**
I metacaratteri ***, ?, [ ]** sono ***espansi nei nomi di file*** secondo un meccanismo di *pattern matching*

Sostituzioni della shell

Come si è visto, la shell opera con sostituzioni testuali sul comando e prepara l'ambiente di esecuzione per il comando stesso

Riassunto fasi

ridirezione e piping e le fasi di sostituzioni:
1) sostituzione comandi
2) sostituzione variabili e parametri
3) espansione dei nomi di file

Comandi relativi al controllo dell'espansione:

' (quote) nessuna espansione (né 1, né 2, né 3)

" (doublequote) solo sostituzioni 1 e 2 (non la 3)

```
y=3
echo ' * e $y '      # produce * e $y
echo " * e $y "      # produce * e 3
echo "`pwd`"         # stampa nome dir corrente
```

sia " che ' impediscono allo shell di interpretare i caratteri speciali per la ridirezione (< > >>) e per il piping (l)

Sostituzioni della shell: Esempi

Riassunto fasi
ridirezione e piping e le fasi di sostituzioni:
1) sostituzione comandi
2) sostituzione variabili e parametri
3) espansione dei nomi di file

Scansione della linea di comando con piu' passate successive (una per ciascuna fase).

Esempi:

```
$ es='??'
$ $es
tt: execute permission denied
$
```

shell esegue fasi 1, 2 (sostituzione di es con ??), 3 (sostituzione di ?? con un ipotetico file tt presente nel dir corrente) e prova quindi ad eseguire tt che non ha però i diritti di esecuzione

```
$ rr='`pwd`'
$ echo $rr
`pwd`
```

shell esegue fasi 1, 2 (sostituzione rr), 3, ed esegue quindi echo `pwd`

ATTENZIONE:

La shell esegue **una sola espansione di ciascun tipo** (comandi, variabili, metacaratteri nomi file)

```
y=3
x='$y'
echo $x    # stampa $y (perché x vale $y)
```

Per ottenere una ulteriore sostituzione, se richiesta, bisogna **FORZARLA ➔ eval**

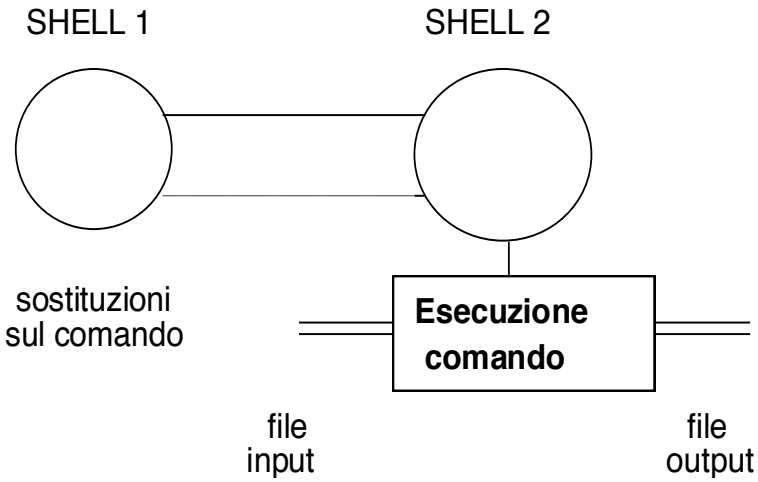
```
eval echo $x     # stampa 3 (perché valuta $y)
```

eval esegue il comando passato come argomento (dopo che la shell ne ha fatto il parsing). **eval** consente quindi una ulteriore fase di sostituzione

Esempio:

```
$ p='ls|more'
$ $p
ls|more: execute permission denied
$
$ eval $p
5_6_98client
5_6_98server
Ascii_pic
Attr
….
```

Esecuzione di un COMANDO



Il comando **esegue** avendo collegato:

- il proprio input al *file di input*
- il proprio output al *file di output*

specificati dalla **shell di lancio**

ESECUZIONE IN BACKGROUND (&)

La shell padre aspetta il completamento del figlio
esecuzione in ***foreground*** **(sincrona)**

È possibile non aspettare il figlio, ma proseguire
esecuzione in ***background*** **(asincrona)**

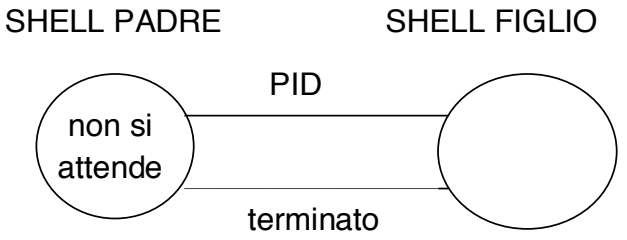
La shell invocante è immediatamente attivo

<comando> [<argomenti>] **&**
Process ID: <number>
Identificatore del processo in background
Quando termina lo shell padre viene avvisato

I processi in background si eliminano con
kill <PID>

I processi in background mandano l'output sulla console
si possono mescolare i messaggi di diversi processi

Devono prendere l'input da file (***ridirezione***)
altrimenti ci sarebbe confusione sull'input
(INTERFERENZA)



Qualche altro comando per una elaborazione avanzata

cut

cut è un altro comando utilissimo per la manipolazione di testo, che consente di selezionare parti di una stringa (**o di ciascuna riga di un file**).

cut permette di selezionare i caratteri della stringa che si trovano nelle posizioni specificate (opzione **-c**).

cut può dividere una stringa in più campi (opzione **-f**) usando uno specifico delimitatore (opzione **-d**) e può selezionare i campi desiderati.

Esempio 1: Selezione caratteri

```
cstefanelli@linuxdid:~$ s="pippo,pluto,paperino"

# stampa dal secondo al nono carattere
cstefanelli@linuxdid:~$ echo $s | cut -c 2-9
ippo,plu

# stampa dal secondo carattere a fine riga
cstefanelli@linuxdid:~$ echo $s | cut -c 2-
ippo,pluto,paperino

# stampa il primo e il quinto carattere
cstefanelli@linuxdid:~$ echo $s | cut -c 1,5
po
```


Esempio 2: Selezione campi

cstefanelli@linuxdid:~$ s="pippo,pluto,paperino"

```
# seleziona il secondo campo della stringa dove
# i campi sono divisi da virgole
cstefanelli@linuxdid:~$ echo $s|cut -f 2 -d ‘,’
pluto
```

Esempio 3: Stampa il nome e la home directory di ciascun utente del sistema

```
cesare@linuxdid:~$ cut -d : -f 1,6 /etc/passwd
root:/root
cesare:/home/cesare
…
```

Si ricordi il formato del file /etc/passwd:

username:password:UID:GID:commento:directory:shell

tr

tr è un comando che permette di fare semplici trasformazioni di caratteri all'interno di una stringa (**o di ciascuna riga di un file**), ad esempio sostituendo tutte le virgole con degli spazi.

ESEMPI:

```
cstefanelli@linuxdid:~$ str="pluto,plutone"

# sostituisce tutte le occorrenze del carattere
# "," con ":" in tutta la stringa
cstefanelli@linuxdid:~$ echo $str | tr , :
pluto:plutone

# elimina (opzione -d) il carattere "," dalla
# stringa
cstefanelli@linuxdid:~$ echo $str | tr -d ,
plutoplutone
```

tr esegue anche sostituzioni tra set di caratteri.

```
# sostituisce caratteri minuscoli con maiuscoli
cs@host:~$ echo $str|tr "[:lower:]" "[:upper:]"
PLUTO,PLUTONE

cs@linuxdid:~$ str="qui,quo,qua:pluto:pippo"
# sostituisce tutte le occorrenze dei caratteri
# "," e ":" con " " e "," (rispettivamente)
# in tutta la stringa
cs@linuxdid:~$ echo $str | tr ",:" " ,"
qui quo qua,pluto,pippo
```


seq

seq è un comando che stampa sequenze di numeri. Può essere molto utile per la realizzazione di cicli negli script.

ESEMPI:

```
# stampa numeri da 1 a 5
cstefanelli@linuxdid:~$ seq 1 5
1
2
3
4
5
```

find

find permette di cercare file all'interno del file system che soddisfano i requisiti richiesti dall'utente, ed eventualmente di manipolarli.

Esempio: stampa tutti i nomi delle sottodirectory contenute in $HOME/code

```
cstefanelli@linuxdid:~$ find code –type d
```

Esempio: trova tutti i file *.bkp nella directory mydir ed eliminali

```
cstefanelli@linuxdid:~$ find mydir –name "*.bkp" –type f -delete
```


tee

Il comando ***tee*** copia lo standard input sia nello standard output sia in un file passato come parametro.

È molto utile quando si scrivono comandi con molte pipe, per monitorare il funzionamento di uno stadio della pipe.

Esempio e opzioni:

cat nomefile.txt l grep stringa l tee altrofile.txt

cat nomefile.txt l grep stringa l tee –a altrofile.txt # append

Programmare in Shell

Shell: lo strumento RAD definitivo

I sistemi Unix propongono numerosissimi comandi di sistema. **Attraverso l'uso in combinazione di questi comandi è possibile realizzare in modo rapido applicazioni anche molto complesse.**

RAD: Rapid Application Development

Niente evidenzia il valore della shell più del seguente aneddoto di rilevante importanza storica nel mondo dell'informatica.

Nel 1986, Jon Bentley, columnist dell'importantissima rivista *Communications of the ACM* chiese a Donald Knuth, il guru mondiale degli algoritmi, di scrivere un programma per leggere un file di testo, determinare le N parole più ricorrenti e stampare una lista ordinata di quelle parole insieme alla rispettiva frequenza. Knuth sviluppò il programma richiesto utilizzando il linguaggio di programmazione Pascal e lo stile *literate programming*. Il programma scritto da Knuth era, ovviamente, un capolavoro: più di 10 pagine di Pascal perfettamente commentate che facevano un uso estremamente intelligente di algoritmi e strutture dati appositamente progettate. Knuth ovviamente aveva dedicato moltissimo tempo ed energie alla scrittura del programma.

Doug McIlroy, invitato da Bentley a commentare pubblicamente il lavoro di Knuth, rispose che la stessa cosa si poteva fare con 6 righe di shell!!!

http://www.leancrew.com/all-this/2011/12/more-shell-less-egg/

Una Moltitudine di Shell

La shell non è unica. Nei moderni sistemi Linux (e Unix) sono disponibili diversi tipi di shell.

- sh: Bourne Shell
- bash: Bourne Again Shell (versione avanzata di sh)
- zsh: Z Shell (versione molto avanzata di sh)
- ksh: Korn Shell
- csh: C Shell (sintassi simile al C)
- tcsh: Turbo C Shell (versione avanzata di csh)
- rush: Ruby Shell (shell basata su Ruby)
- hotwire: (propone un interessante e innovativo modello integrato di terminale e shell)

Ogni utente può **specificare quale shell desidera usare**.

La shell più usata è sicuramente Bash, che è molto simile alla shell di Bourne (/bin/sh).

La shell di Bourne, creata da Stephen Bourne dei laboratori AT&T negli anni ’70, ha sostituto la shell di Thompson, che non consentiva lo sviluppo di file comandi.

La shell di Bourne è stata poi affiancata da altre shell come Bash, Korn shell e Z shell, che ne riprendono la sintassi, ampliandola significativamente.

Si ricordi che, come i nomi di file nel filesystem, in ambiente Unix anche la shell è **case sensitive**.

PROGRAMMAZIONE NELLA SHELL

Il PROCESSORE COMANDI è in grado di elaborare comandi prendendoli da un file ➔ ***file comandi***

Linguaggio Comandi

Un ***file comandi*** può comprendere
- statement per il controllo di flusso
- variabili
- passaggio dei parametri

N.B. *quali* statement sono disponibili dipende da *quale shell* usiamo

ESECUZIONE FILE COMANDI

Per eseguire un file comandi (myFile):

• si rende eseguibile il file e poi lo si lancia:

```
chmod +x  myFile
myFile
```

ATTENZIONE: la shell cerca i comandi da eseguire solo all'interno delle directory appartenenti al percorso di sistema, specificato dalla variabile PATH.

Se vogliamo lanciare un comando che non risiede in una directory del percorso di sistema, dobbiamo necessariamente specificare il nome completo (relativo o assoluto) del file da invocare.

Uso di ./ per l'esecuzione di un file che risiede nella directory corrente, se questa non appartiene al percorso di sistema:

```
./myFile
```

oppure si può invocare direttamente una shell che lo esegua:

```
sh myFile
```

In questa modalità di invocazione la shell non cerca il file comandi myFile nel percorso di sistema, ma si limita ad aprire il file il cui nome è fornito a riga di comando. Quindi, non è necessario anteporre il prefisso ./ al nome di file comandi che risiedono nella directory corrente.

VARIABILI

Come già visto, le **variabili** contengono stringhe
NON è necessaria la definizione delle variabili

Il nome delle variabili è libero (alcune predefinite)
Il contenuto delle variabili è indicato dal metacarattere **$**

```
echo $HOME          # stampa il direttorio di default

echo PATH $PATH    # stampa PATH   /:/bin:$HOME:.
```

(il carattere ':' è il separatore dei vari campi in PATH)

Assegnamento

<variabile>=<valore> ***# niente spazi!***

```
i=12
echo i $i
j=$i+1
echo $j
```

La prima echo fornisce la stringa i 12
la seconda echo fornisce la stringa 12+1

Le variabili sono trattate come stringhe di caratteri

```
v="ls -F" ; $v
v2="ls -lga a*"
$v2     # al momento dell'invocazione, viene espanso in
        # "tutti i file del dir. corrente che iniziano per a"
echo $v  ;  echo v  ;  echo `$v`
```

PASSAGGIO PARAMETRI

comando argomento1 argomento2 ... **argomentoN**

Gli argomenti sono **variabili posizionali** nella linea di invocazione

- **$0** rappresenta il nome del file comandi in esecuzione
- **$1** rappresenta il primo argomento
- ....

DIR /usr/utente1 (il file DIR contiene *ls $1*)
l'argomento $0 è DIR
l'argomento $1 è /usr/utente1

DIR1 /usr/utente1 "*" (DIR1 contiene *cd $1; ls $2)*
il direttorio è cambiato solo per la sotto-shell
produce la lista dei file del direttorio specificato

NB: * deve essere passato a ls senza essere sostituito ➔ virgolette! (cosa produrrebbe **DIR1 /usr/utente1 *** ?)

COMANDI CORRELATI
È possibile far scorrere tutti gli argomenti ➔ **shift**

- $0 non va perso, solo gli altri sono spostati ($1 perso)

	$0	$1	$2
prima di **shift**	DIR	-w	/usr/bin
dopo **shift**	DIR	/usr/bin	

È possibile riassegnare gli argomenti ➔ **set**

set exp1 exp2 exp3 ...

- gli argomenti sono assegnati secondo la posizione

ALTRE VARIABILI

Oltre agli argomenti di invocazione del comando

$* l'insieme di tutte le variabili posizionali, che corrispondono agli argomenti del comando: $1, $2, ecc.
$# il numero di argomenti passati (***$0 escluso***)
$? il valore (int) restituito dall'ultimo comando eseguito
$$ l'identificatore numerico del processo in esecuzione (pid, proces identifier)

Altre variabili predefinite sono:

$PATH percorso di sistema
$HOME direttorio dell'utente
$PWD direttorio corrente
$UID id dell'utente
$IFS lista di caratteri da considerare come separatori di input. Il valore di default è spazio, tab, e nuova riga (\n).

Il contenuto di IFS viene considerato per tutte le operazioni di parsing dell'input (es. read, liste di stringhe, ecc.), e può essere modificato dall'utente – ad esempio nel caso in cui si debba processare un input un formato di dati in cui i caratteri spazio e tab assumono un significato ben preciso.

INPUT/OUTPUT

Per ricevere l'input da tastiera si usa il comando read, e per stampare a video si usa il comando echo:

```
read  var1 var2                          #input
echo  var1 vale $var1 e var2 $var2       #output
```

read la stringa in ingresso viene attribuita alla/e variabile/i secondo corrispondenza posizionale

ATTENZIONE: read legge una sola riga di input e ne assegna il contenuto alle variabili passate come parametro, considerando come separatori dell'input i caratteri contenuti nella variabile IFS – ovverosia spazio, tab e fine riga (\n).

ESEMPIO DI READ

Esempio 1. Input insufficiente

```
cstefanelli@linuxdid:~>  read pippo pluto
2
cstefanelli@linuxdid:~>  echo $pippo
2
cstefanelli@linuxdid:~>  echo $pluto
```

Esempio 2. Input corretto

```
cstefanelli@linuxdid:~> read pippo pluto
2 3
cstefanelli@linuxdid:~> echo $pippo
2
cstefanelli@linuxdid:~> echo $pluto
3
```

Esempio 3. Input eccedente

```
cstefanelli@linuxdid:~> read pippo pluto
2 3 4
cstefanelli@linuxdid:~> echo $pippo
2
cstefanelli@linuxdid:~> echo $pluto
3 4
```

Per evitare errori nel parsing dell'input, è bene effettuare una read separata per ciascuna variabile.

ARRAY

La shell di Bourne **NON supporta direttamente array, ma consente di simularli** "in qualche modo":

INPUT:

```
i=3;    read array$i
eval    array$i=espressione
eval    array`echo $i`=espressione
```

OUTPUT:

```
eval    echo \$array$i
```

Si noti l'uso di eval per ottenere il nome di una cella dell'array.

CODICI DI RITORNO DEI COMANDI

Ogni statement in uscita restituisce un **valore di stato,** che indica **il completamento o meno del comando**

Tale valore di uscita è posto nella variabile ?

$? può essere riutilizzato in espressioni o per il controllo di flusso successivo

Stato **vale zero** ➔ comando OK
valore positivo ➔ errore

ESEMPIO 1

cp a.com b.com

se il comando non è riuscito (es: il file a.com *non* esiste) **allora** *errore* (valore > 0) **altrimenti** *successo* (valore 0)

$ **cp a.com b.com**

cp: cannot access a.com

$ **echo $?**

2

ESEMPIO 2

```
ls file
grep "stringa"  file            # stato OK  se trovato
echo stato di ritorno $?
```

---

test

Comando per la **valutazione di una espressione**

test -<opzioni> <nomefile>

Restituisce uno stato uguale o diverso da zero
- valore **zero ➔ true**
- valore **non-zero ➔ false**

ATTENZIONE: convenzione opposta rispetto al C!

Motivo: i codici di errore possono essere più di uno e avere significati diversi

TIPI DI TEST POSSIBILI

test **-f** <nomefile> esistenza di file
-d <nomefile> esistenza di direttori
-r <nomefile> diritto di lettura sul file (**-w** e **-x**)

test <stringa1> **=** <stringa2> # uguaglianza stringhe
test <stringa1> **!=** <stringa2> # diversità stringhe

ATTENZIONE:
- gli **spazi intorno all' =** (o al !=) sono ***necessari***
- stringa1 e stringa2 possono contenere metacaratteri (attenzione alle espansioni, può essere necessario usare le virgolette ' oppure " a seconda dei casi)

test **-z** <stringa> # vero se stringa è nulla
test <stringa> # vero se stringa *non* è nulla

TIPI DI TEST / segue

test <numero1> [**-eq -ne -gt -ge -lt -le**] <numero2>
confronta tra loro due stringhe numeriche, usando uno degli operatori relazionali indicati

Espressioni booleane

! not
-a and
-o or

---

expr

Per le **espressioni aritmetiche** può essere utile l'uso del comando **expr**

L'operatore expr valuta l'espressione aritmetica e *produce un valore*

ESEMPIO:

expr 24 - 12 + 65 *# fornisce 77*

i=12; j="$i + 1"; echo $j ➔ 12 + 1 # attenzione a spazi
expr $j; ➔ 13
expr $i + 1 -16 ➔ -3

STRUTTURE DI CONTROLLO

ALTERNATIVA

if <lista-comandi>
  then
    <comandi>
  [**else** <comandi>]
fi

ATTENZIONE:

- le parole chiave (do, then, fi, etc.) devono essere o **a capo** o **dopo il separatore ;**
- if controlla il valore in uscita dall'ultimo comando di <lista-comandi>

Esempio

```
# fileinutile
# risponde “sì” se invocato con “si” e un numero < 24
if test $1 = si -a $2 -le 24
    then echo si
    else echo no
fi
```

Esempio di invocazione:

fileinutile si 12 ➔ *stampa* si

Esempio (controllo numero argomenti)

```
if  test  $1; then echo OK
          else echo Almeno un argomento
fi
```

Esempio

file leggiemostra (uso: leggiemostra filename)

```
read var1
if test $var1 = si
then
  if test -f $1
  then ls -lga $1; cat $1
  fi
else echo niente stampa $1
fi
```

ALTERNATIVA MULTIPLA

case <var> **in** **#** alternativa multipla dip. da var

<pattern-1>**)** <comandi> **;;**
...
<pattern-i> **|** <pattern-j> **|** <pattern-k>**)** <comandi>;;
...
<pattern-n>**)** <comandi> **;;**
esac

ESEMPI:

```
read  risposta
case  $risposta in
    S* | s* | Y* | y* )  < OK >;;
    * )                  <problema>;;
esac
```

```
# append: invocazione  append [dadove] sucosa
case $# in
  1) cat >> $1;;
  2) cat < $1 >> $2;;
  *) echo  uso: append [dadove] adove; exit 1;;
esac
```

Attenzione: il **case** è l'unico caso in cui i metacaratteri * e ? non vengono espansi sui nomi di file del direttorio corrente, ma servono solo per eseguire un pattern matching con <var>

RIPETIZIONI ENUMERATIVE

for <var> [**in** <lista di stringhe>]
do
  <comandi>
done

scansione della lista di stringhe e ripetizione del ciclo per ogni elemento della lista. Se non si specifica una lista di stringhe, la shell assume $* come valore di default.

ESEMPI

```
for i in * #esegue per tutti i file nel dir corrente
do
  <comandi>
done

for i  #cioè in $*  esegue per tutti i par  di invocazione
do
  <comandi>
done

for i in `cat file1`# esegue per ogni parola contenuta
                    # nel file file1
do    <comandi>
done

# stampa numeri da 1 a 15 a intervalli di 3
for i in `seq 1 3 15`
do
 echo -n "$i,"
done

# risultato: 1,4,7,10,13,
```

RIPETIZIONI NON ENUMERATIVE

while <lista-comandi>
do
<comandi>
done

Si ripete per tutto il tempo che il valore di stato dell'ultimo comando della lista è zero (successo); si termina quando tale valore diventa diverso da zero.

```
# file esiste (cicla fino a comparsa file di nome $1)
# invocazione   esiste nomefile
while test ! -f $1      # ls non ritorna 1 se non trova il file
  do  sleep 10; echo file assente
  done
```

until <lista-comandi>
do
<comandi>
done

Come while, ma inverte la condizione

```
# file esisteutente
# invocazione esisteutente nome
until who l grep $1
  do sleep 10
  done
```

Uscite anomale
- vedi C: **continue**, **break** e **return**
- **exit** [**status**]: funzione primitiva di UNIX

MENU TESTUALI

Il comando *select* può essere usato per realizzare semplici menù testuali.

Esempio:

```
select risposta in pippo pluto paperino;
do
     echo "Hai scelto $risposta"
done
```

All'esecuzione dello script di cui sopra, la shell stampa il menù:

```
1) pippo
2) pluto
3) paperino
#?
```

Nel caso l'utente digiti 1 la shell stampa:

```
Hai scelto pippo.
```

Esempi di uso di select

Il comando select è particolarmente utile per la selezione di file:

```
#!/bin/bash
echo "Scegli quale file cancellare"
select i in `ls *.bak`
do
    echo "Selezionato: $i"
    rm -f $i
done
```

Un'altra utile applicazione è costituita dalla selezione tra un insieme di operazioni:

```
#!/bin/bash
select i in "stampa" "cancella"
do
    echo "Selezionato: $i"
done
if test $i = "stampa"
then
  ...
fi
...
```


AMBIENTE di un FILE COMANDI

L'*ambiente* dei file comandi è composto da:
- ***variabili predefinite***
- ***direttorio corrente***
- *insieme di variabili usate e variate dall'utente*

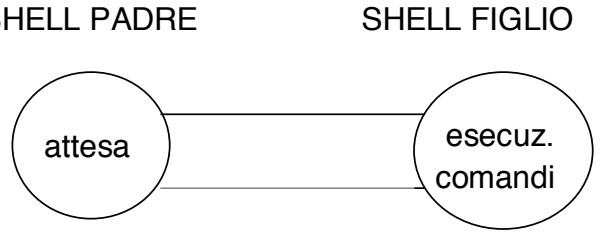
L'ambiente è accessibile agli ***shell figli***, che possono accrescere il tutto e aggiungere nuove variabili

- **viene passato solo l'ambiente predefinito iniziale**
- **sono aggiunti i valori delle sole variabili esplicitamente esportate (istruzione export)**

NB: in caso di modifica di variabili predefinite, i nuovi valori devono essere esplicitamente esportati

L'ambiente padre è preservato

OGNI COMANDO è ESEGUITO da una NUOVA SHELL, distinta dalla shell-padre
La shell-figlio ottiene una copia (parziale) dell'ambiente del padre *ma le eventuali modifiche NON sono viste dalla shell padre*



ESEMPI

```
# file view filename [filename]
case $# in
  0) echo Usage is: $0 filename [filename]
     exit 1;;
  *) ;;
esac
for i in $*        # in $* poteva essere omesso
do
  if test -f $i          # se il file esiste
    then
      echo $i        # visualizza il nome del file
      cat $i         # visualizza il contenuto del file
    else    echo file $i non presente
  fi
done


# file lista  <filenames>
for i
do
  echo -n "$i ?" > /dev/tty  # terminale attual. connesso
    # -n = non emettere newline;
    #  ? non è sostituito (ci sono le virgolette)
  read answer
  case  $answer in
     y*I Y*I s* I S* ) echo $i; cat $i ;;
  esac
done
```

NOTA BENE: I comandi utente sono trattati in modo OMOGENEO ai comandi di sistema

```
lista f* > temp
lista  ?p* I grep  <stringa>
```



```
# file cercadir [direttorioassoluto] file
case $# in
  0) echo errore. Usa  $0 [direttorio] file
     exit 2 ;;
  1) d=`pwd`;  f=$1;;
  2) d=$1;      f=$2;;
esac

PATH= ....    # quali direttori bisogna considerare?
              # PATH deve comprendere il direttorio in cui
              # si trova cercadir stesso
cd $d
if test -f $f
 then echo il file $f è in $d
fi
for i in *
do
   if test -d $i
     then echo direttorio $d/$i
          cercadir `pwd`/$i $f          # ricorsione
   fi
done
```

NB: questo esempio sottolinea il tipico problema che c'è spostandosi nei direttori: i comandi non inclusi nel path *non* verranno trovati
In alternativa si possono considerare i **nomi assoluti** dei comandi che non sono nel path

File comandi ricorsivi tendono ad agire su un sottodirettorio per volta e a lanciare una invocazione per ogni sottodirettorio trovato: *non si devono prevedere così le profondità dell'albero dei direttori*

Controllo degli argomenti

Il controllo degli argomenti nei FILE COMANDI è fondamentale

E' necessario **verificare gli argomenti**

- devono essere **innanzitutto** ***nel numero giusto***
- **poi** ***del tipo richiesto***

```
# invocazione  di comando per ...
case $# in
0|1|2|3|4)    echo Errore. Almeno 4 argomenti ... >& 2
              exit 1;;
esac

# in alternativa:
if test $# -lt 4
     echo Errore …
fi

# in caso di argomenti corretti:
# primo argomento: dev'essere un direttorio (o file)
if test ! -d $1
      then echo argomento sbagliato: $1 direttorio >& 2; exit 2
fi

# primo argomento: se è un nome di file assoluto
case $1 in
     /*)  if test ! -f $1
               then echo argomento sbagliato: $1 file >& 2; exit 2
          fi;;
     * ) echo argomento sbagliato: $1 assoluto; exit 3;;
 esac
```



```
# primo argomento: se è un nome di file relativo
case $1 in
    */*) echo argomento sbagliato: $1 nome relativo >& 2; exit 3;;
     * ) ;;
esac

# secondo argomento: dev'essere stringa numerica
# si potrebbe usare un case con match  [0-9] ?
# e case con match  [0-9] | [0-9][0-9] ?

# si veda il comando cut che consente di selezionare
# le colonne delle righe di un file
# si metta in pipe l'argomento e lo si selezioni in un ciclo
# una lettera alla volta:  echo $2  |  cut -d$i
# il singolo carattere può essere estratto ed esaminato

# il comando expr può verificare una espressione
# numerica (è necessaria la operazione)
expr $i + 0 >  /dev/null 2> /dev/null
if  test $? -ne 0
     then echo errore in argomento numerico: $i &> 2; exit 4
fi
# qual è la ragione delle ridirezioni su /dev/null?

Si possono eliminare alcuni argomenti per comodità di scansione:
si usi lo shift, dopo avere salvato gli argomenti che vengono
eliminati
  salva1=$1  # salvataggio di $1
  shift
  salva2=$1  # salvataggio di $2
  shift
  # cosa vale adesso $*?
  for i
   do
     # il ciclo è fatto per tutti argomenti esclusi quelli tolti
   done
# gli argomenti iniziali sono dati da $salva1 $salva2 $*
```

EDITOR STANDARD DI UNIX: VI

vi (**VI**sual display editor)

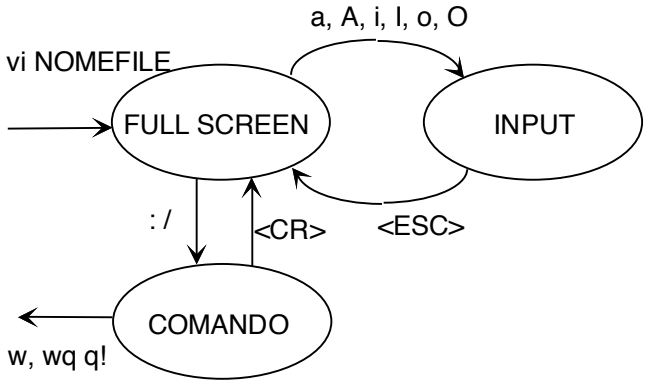
- è l'editor standard di Unix
- non basato sul concetto di WINDOW (finestra)
- ma usabile da un qualunque tipo di terminale

➔ È un Editor **a linee, *non* full-screen**

Vi ragiona a *stati*:

1) lo stato comandi display (*full screen*)
2) lo stato riga comandi (*comando*)
3) lo stato di input (*input*)

Diagramma a stati di VI:



legenda: Maiuscole prima del cursore, minuscole dopo il cursore

Per invocarlo: **vi <nomefile>**

il file di nome <nomefile> viene aperto se esiste, o creato altrimenti. Ha inizio l'editing e l'accettazione dei comandi.

CASO NUOVO FILE ➔ vi prova.c

```
<===  cursore
~
~
~
~
~
~
~
~
 "prova.c" [New file]                                    <===
```

RIGA COMANDI

CASO FILE ESISTENTE ➔ vi prova.c

```
# include <stdio.h>
main(argc, argv)
int argc;
char ** argv;
{int i;
   for (i = 0; i < argc; i++)
        printf("argv[%d] = %s\n", argv[i]);
}
~
 "prova.c"   9 lines, 138 characters                     <===
```

RIGA COMANDI

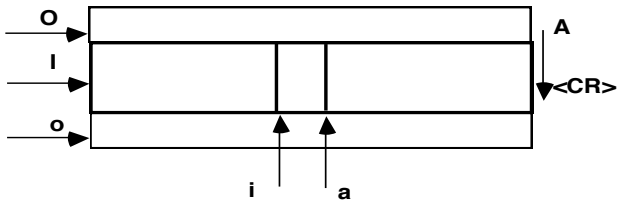
Nel caso di file lungo, viene visualizzata la parte iniziale

Il file è considerato composto da un insieme di linee, ciascuna completata dal terminatore di linea (<CR>)

Ogni linea è formata da parole e da caratteri, e ci sono comandi adatti per ogni possibile azione.

COMANDI di INPUT

Si passa da stato comandi a stato input nelle diverse posizioni :



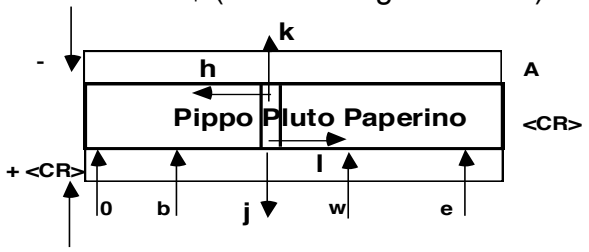
COMANDI DI SCHERMO

MOVIMENTO

di un carattere ← h ↓ j ↑ k → i
<blank> carattere successivo
<backspace> carattere precedente

di una parola **w** (dopo), **b** (prima)
e (fine della parola corrente)

di una riga **<CR>** o **+** (prossima)
- (inizio della riga precedente)
$ (fine della riga corrente)



*Alcuni comandi possono essere preceduti da un numero che indica **quante volte** il comando deve essere **ripetuto***

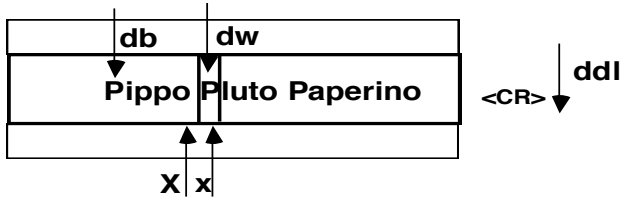
Ad esempio: **3w** sposta il cursore in avanti di 3 parole

VISUALIZZAZIONE

visualizza videata seguente **<CTRL> F** (indicato **^F**)
visualizza videata precedente **<CTRL> B** (indicato **^B**)
rinfresca il buffer (corrente) **<CTRL> L** (indicato **^L**)

CANCELLAZIONE

di un carattere **x** (corrente), **X** (prima)
di una parola **dw** (dopo), **db** (prima)
di una riga **dd**
di tutto il resto della riga corrente **d$** o **D**



SOSTITUZIONE

di un carattere	**r** <ch>
di un testo al posto del carattere	**R** testo **<ESC>**
di un testo al posto della parola corrente	**cw** testo **<ESC>**

AZIONI VARIE

ripete l'ultima azione	**.**
annulla l'ultima azione eseguita **(undo)**	**u**
annulla tutte le azioni sulla linea	**U**
unisce la linea corrente con la seguente	**J**
ripete la ricerca in avanti	**n**
ripete la ricerca all'indietro	**N**

COMANDI con uso di BUFFERIZZAZIONE

Esiste un buffer corrente senza nome, in cui sono anche memorizzate le ultime parti distrutte

copia la riga corrente in un buffer	***n*Y *n*yy**
copia il buffer sopra la riga corrente	**P**
copia il buffer dopo la riga corrente	**p**
(maiuscolo prima, minuscolo dopo)	
altri buffer (nomi da **a** a **z**, p.e. **f**)	**"fnyy**
salvataggio e ripristino	**"fnp "fnP**

Esempi:

a) COPIA DI UN INSIEME DI RIGHE
- posizionare il cursore sulla prima riga da copiare
- **<n>Y** copia <n> righe nel buffer
- spostare il cursore sulla riga sotto la quale copiare
- **p** inserisce le righe prelevate dal buffer

b) SPOSTAMENTO DI UN INSIEME DI RIGHE
- posizionare il cursore sulla prima riga da spostare
- **<n>dd** muove <n> righe nel buffer
- spostare il cursore sulla riga sotto la quale copiare
- **p** inserisce le righe prelevate dal buffer

COMANDI ULTERIORI

COMANDI dalla RIGA di COMANDO

RICERCA

di una stringa specificata (avanti)	**/**stringa
di una stringa (indietro)	**?**stringa

MOVIMENTO del cursore

su una linea specificata	**:**numerolinea
ultima linea del file	**:$**

ELIMINAZIONE

di linee del file	**:n,md** *linee da n a m*

INTERAZIONE con il FILE SYSTEM

lettura ed inserimento di file alla posizione corrente	**:r** nomefile
lettura ed inserimento di file dopo la linea n	**:nr** nomefile
uscita da vi	**:q**
senza salvare	**:q!**
scrittura su file	**:w** nomefile
parti su file	**:n,mw** nomefile
aggiunte su file	**:n,mw >>** nomefile
forzata su file	**:w!** nomefile
superando i normali controlli (se possibile)	
specifica di opzioni	**:set opzione**
visualizza i numeri di riga	**:set nu**mber
elimina i numeri di riga	**:set nonu**mber

ESECUZIONE di un qualunque comando di SHELL
escape ad uno shell **:!**<comando di shell>

Esempi di comandi:
a) SALVA ED ESCE DALL'EDITOR
- **<ESC>** esce dallo stato input
- **:wq** salva ed esce dall'editor

ATTENZIONE:
Se il file non consente la scrittura da parte dell'utente che sta usando il vi, l'editor riportera' un errore quando si tenta di scrivere sul file

b) ESCE DALL'EDITOR SENZA SALVARE
- **<ESC>** esce dallo stato input
- **:q!** esce dall'editor senza salvare

ATTENZIONE:
Se si usa :q senza aver salvato il file, l'editor avvisa
No write since last change (:quit! overrides)

Esistono moltissimi altri comandi: nonostante la scarsa leggibilità, **vi** è l'unico editor presente in tutti i sistemi UNIX. Si consiglia di approfondire la conoscenza del vi.

Attualmente, la versione più moderna e potente di vi è sicuramente vim, http://www.vim.org. Vim supporta syntax highlighting, spell checking, etc.