SISTEMA OPERATIVO

Sistema Operativo come gestore risorse
rende efficiente l'uso delle risorse sia hardware che software

Sistema Operativo come macchina virtuale
semplifica l'uso del sistema di calcolo da parte degli utenti e processi

Gestore della attribuzione delle risorse del sistema ai processi di utente in base al tipo di richiesta e agli obiettivi da raggiungere ***(uso efficiente delle risorse)***

Risorse Hw e Sw: tempo di CPU, memoria, I/O, ...
* risoluzione di **conflitti** nell'uso delle risorse
* scelta dei **criteri** con cui assegnare una risorsa

Macchina virtuale astratta per semplificare l'uso delle risorse e nascondere i dettagli implementativi.

Esempio**: Controllore di un floppy disk**
* numerosi comandi *(lettura, scrittura, movimento del braccio, formattamento delle tracce, etc..)*
* ogni comando ha più parametri *(indirizzo del blocco, numero di settori per traccia, etc..)*
* numerose condizioni di stato e di errore al completamento del comando

Necessità di nascondere all'utente i dettagli hardware legati al particolare dispositivo.
=> operazioni del **S.O.** ***(file system)***

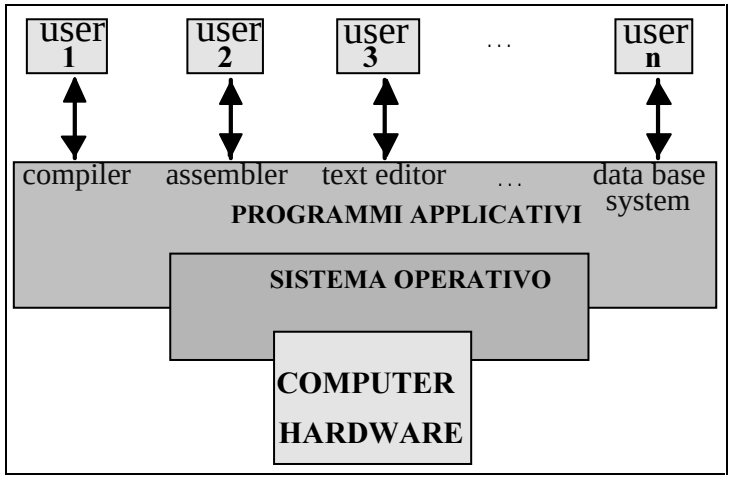
SISTEMA OPERATIVO UNIX

Bell Laboratories dell'AT&T

- **sistema multiutente e multitasking**
- **memoria virtuale**

Versioni recenti del sistema
System V, sviluppato dall'AT&T
BSD 4.x (**B**erkeley **S**oftware **D**istribution)

UNIX
multiutente e multithreaded



Ogni processo vede una risorsa
memoria virtualmente infinita

UNIX

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

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

Gli strumenti di uso sono:

processore comandi (**shell**)

nucleo (**primitive di sistema**)

linguaggio di sistema (**C)**

UNIX livelli

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

FILE SYSTEM

- FILE COME **STREAM DI BYTE**
  NON previste organizzazioni logiche o accessi a record

- FILE SYSTEM **gerarchico**
  ALBERO di sottodirettori come SISTEMA di NOMI come puntatori ai file corrispondenti raggruppati logicamente

- 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

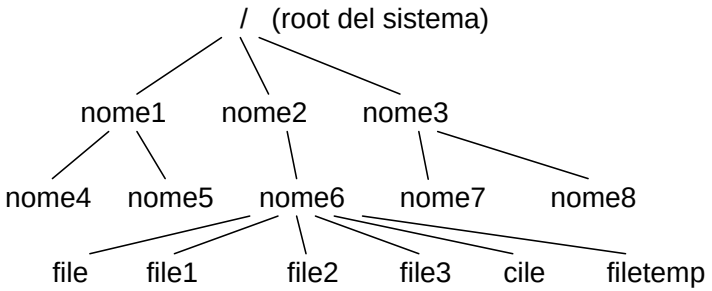
ORGANIZZAZIONE del FILE SYSTEM

NOMI di file **per arrivare alle informazioni**
- **ASSOLUTI**: dalla radice */nome2/nome6/file*
- **RELATIVI**: dal direttorio corrente (o nello stesso)
  nome6/file *file*
  relativi in modo **stretto**, se il nome è relativo al **direttorio di appartenenza**

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

Ogni utente ha un direttorio a **default**
direttorio in cui l'utente lavora all'ingresso nel sistema

I nomi (e la sintassi in generale) sono **case-sensitive**
Si usano abbreviazioni nei nomi dei file (**wild card**)
***** per una qualunque stringa
? per un qualunque carattere



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

Ingresso **LOGIN**
Username: INF54
Password: XXXXXXX

L'accesso viene verificato al login
Uscita **LOGOUT**

L'uscita avviene solo al logout
exit, logout, o <ctrl>D (fine file)

In una sessione si fanno i comandi di una **shell** specificata all'interno di **/etc/passwd**

politiche di gestione

Necessità di politiche di gestione corrette e **giuste**
Necessità di politiche di uso corrette e **giuste**

Ogni utente deve mantenere segreta la propria password per evitare abusi a danno anche di altri utenti e dell'intero sistema

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
N.B.: è memorizzata in forma crittografata

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

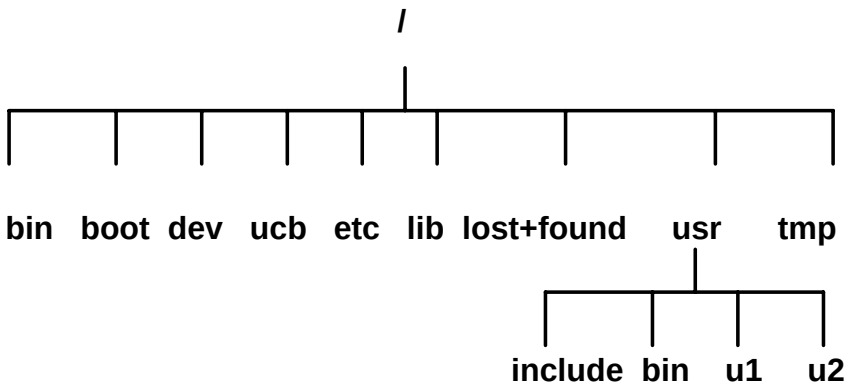
direttorio ==> direttorio iniziale in cui si trova l'utente al login (home directory)

comando ==> comando che viene eseguito automaticamente all'atto del login
N. B.: in genere è uno shell (anche un comando)

ESEMPIO:

root:.XPc4HKe0nPQA:0:1:Operator:/:/bin/sh
antonio:rTIW65BOuQ9ng:120:20:Antonio Corradi:/usr/antonio:/bin/sh

STRUTTURA DEL FILE SYSTEM



bin comandi principali di sistema

dev i dispositivi

etc, file significativi per il sistema,
p. e. **passwd** che memorizza gli utenti autorizzati

lib, le librerie di sistema

tmp, file temporanei

usr, sottodirettorio per utenti

Esiste un utente, **root**, privilegiato rispetto agli altri:
il gestore del sistema che sfugge alle regole di sicurezza

PROTEZIONE dei File

Necessità di regolare gli ***ACCESSI*** alle informazioni

Per un **file**, esistono 3 tipi di **utilizzatori (processi)**:
- il proprietario, **user**
- il gruppo del proprietario, **group**
- tutti gli altri utenti, **others**

Per ogni tipo di utilizzatore, i modi di accesso al file sono:
lettura (r), scrittura (w) ed esecuzione (x)

Ogni utente ha
identificatore (**user ID**) e **gruppo** (**group GID**)
UID e GID passano ai **processi** generati dall'utente

Ogni file è marcato dal proprietario
- user-ID del **proprietario**;
- group-ID del **gruppo**;
- un insieme di 12 bit di protezione

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

i primi 9 ==> triple di permessi
lettura, scrittura ed esecuzione
per proprietario, utenti del gruppo e altri

Ogni operazione confronta i diritti del processo in accesso con quelli del file

Ulteriori bit di diritti

dodicesimo bit
SUID bit set-user-id bit

Se a 1 in un file di **programma eseguibile**,
l'utente in esecuzione (il processo) è assimilato al creatore di quel file, solo per la durata della esecuzione
Un accesso (ad es. in lettura) ad un file del creatore è consentito durante la esecuzione, ma non in modo diretto (e non previsto)

Senza suid, un programma potrebbe causare errori, durante l'esecuzione per operazioni di lettura/scrittura su altri file di cui l'utente non ha i diritti relativi

Per esempio, **passwd**, che cambia la password di un utente, modifica /etc/passwd del superutente, che non può essere modificato direttamente. Solo il suid lo rende possibile

undicesimo bit
SGID bit

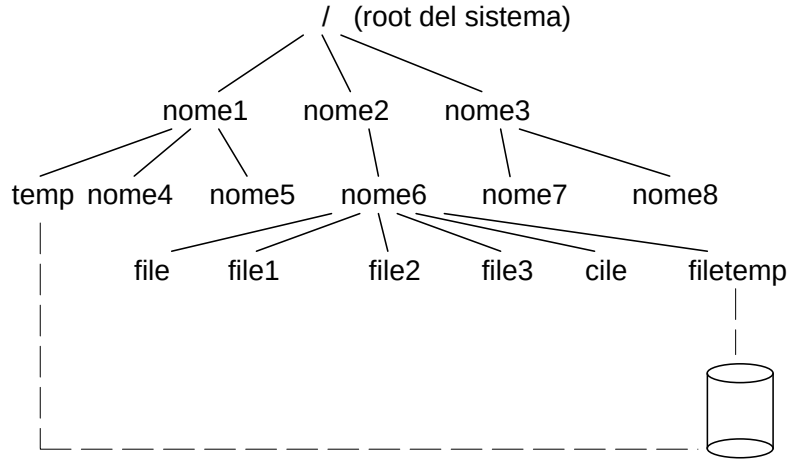
come SUID bit, per il gruppo (set-group-id)

decimo bit
sticky bit

il sistema tende a mantenere in memoria principale la immagine del programma, anche se non è in esecuzione
(indicazione trascurata dal SO)

Linking

Le stesse informazioni contenute in un file possono essere **visibili come file diversi** con nomi di file diversi



ln /nome2/nome6/filetemp /nome1/temp

Il sistema considera e tratta il tutto:

- se un file viene cancellato, le informazioni sono vera mente eliminate solo se non ci sono altri link a esso
- Il link cambia i diritti? ➔ Meglio di no

Due tipi di link:
link fisici (si collegano le strutture del file system)
link simbolici (si collegano solo i nomi)

comando: ln [-s]

Si vedano i link di ogni direttorio
Cosa succede se elimino un file linkato?
Dove è registrata l'informazione necessaria?

SHELL ovvero
il **PROCESSORE COMANDI**

Lo shell interprete dei comandi:
lo shell mette in esecuzione i comandi forniti uno dopo l'altro

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 e riconosci comando da console>
<esegui comando>
until <fine file>
< LOGOUT >
end loop

Maggiori capacità rispetto ad un semplice esecutore di un comando alla volta

COMANDI RELATIVI AL FILE SYSTEM

Sintassi:

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

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

DIRETTORI

mkdir <nomedir>
rmdir <nomedir>
cd <nomedir>
pwd
ls [<nomedir>/<nomefile>]
differenze?

TRATTAMENTO FILE

ln <oldfile> <newfile>
cp <filesorg> <filedest>
mv <vecchionomefile> <nuovonomefile>
rm <nomefile>
cat <nomefile> o **cat** <file1> <file2> <filen>
file <nomefile>
pr <file1> <file2> <filen>

Esempi di comandi:

cd /nome2/nome6
cat file*
ls /nome1
rm *

PROTEZIONE

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

I permessi possono essere concessi o negati
dal ***proprietario del file***
Esempio di variazione dei bit di protezione:

chmod 0755 /usr/dir/file # i diritti di file

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

chmod u-w fileimportante

Altre informazioni sul file mantenute in i-node
ownerID groupID numero link

chown <nomeutente> <nomefile>
chgrp <nomegruppo> <nomefile>

Opzioni
ls -a [<nomedir>]
visualizza file "nascosti", il cui nome inizia con '.' che non vengono mostrati normalmente
(vedi file .profile per ogni utente)

ls -A
come in -a, escludendo **.** e **..**

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, dimensione, nome
ls -i
lista gli i-number dei file con le altre informazioni
ls -1
lista i file una linea alla volta (default in caso di non terminale)
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

Le opzioni si possono combinare

Così per ogni comando

pr +n <pag.> **-h**<heading>
-l <pagel> **-w** <#col> <nomefile>
mostra il contenuto del file a pagine dalla pagina specificata, con l'heading specificato, e con il numero di righe e colonne specificate

ALTRI COMANDI DI SISTEMA

I file sono considerati come insiemi di linee, composti di parole, composte di caratteri, e separate da blank

more <nomefile>
sort <nomefile>
sort [**-r**] <nomefile> [<nomefile>]
r sta per reverse. Molte opzioni:
solo il merge (opzione **-m**)
uscita su file opzione **-o** <nomefileout>).
diff <file1> <file2>
find <direttorio> **-name** <nomefile> **-print**
wc [**-lwc**] [<nomefile>]
si contano le linee (opzione **l**), le parole (opzione **w**) ed i caratteri (opzione **c**) dello standard input o del file specificato.

COMANDI di STATO del sistema

date
time
who
elenca gli utenti collegati
ps
elenca i processi correnti nel sistema
sleep
sospensione del processo quando specificato
man <comando>
help del comando

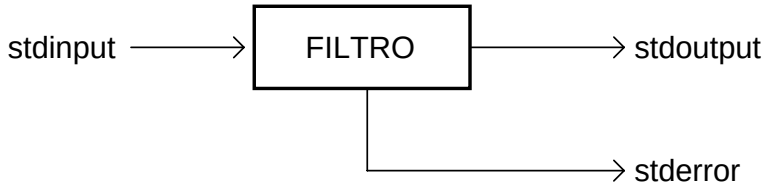
ESEMPIO: date; time; who; ps

RIDIREZIONE DELL'I/O

Tutti i comandi di UNIX sono **filtri**

Un **filtro** è un programma che riceve in ingresso da un input e produce risultati su output (uno o più)

standard input --> console
standard output --> console
standard error --> console



I filtri possono operare sull'input considerandolo a linee

more
sort
grep <stringa>
rev
rovescia le linee (ogni linea viene rovesciata)
tee <file>
filtra da input a output e mette anche nel file

head [-numerolinee]
tail [-numerolinee]
filtrano le prime/ultime linee del file
awk
gestore di trasformazioni di stringhe per un file

Ciascuno dei comandi può essere ridiretto
su un file diverso
senza cambiare il comando

OMOGENEITÀ dispositivi e files

ridirezione dell'input
<comando> **<** <fileinput>

ridirezione output
<comando> **>** <fileoutput>

<comando> **>>** <fileoutput>

Alcuni esempi di ridirezione:

ls -lga > file

ls > /tmp/file
cd /tmp
sort < file > filetemp
rev < filetemp > file
more < filetemp file
rm file filetemp

who > temp
wc -l temp

ps > file

RIDIREZIONE (estensioni)

Ogni comando trova aperti dallo shell

stdin, stdout, stderr

RIDIREZIONE MULTIPLA

In caso di ridirezione, il file di ingresso è aperto in lettura, il file di output aperto in scrittura

(cioè creato o con il contenuto cancellato)

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

Cosa produce il comando seguente?

> filequalunque

ALTRI FILE

In Bourne Shell è possibile anche usare altri file in ridirezione, specificandoli alla invocazione del comando o agganciare più output ad uno stesso file

comando **2>** file2 **3>** file3 **5>** file5

Si richiede la apertura del file2 con chiave 2, del file 3 con chiave 3, etc.

comando **>& 2**

L'output del comando viene forzata sul canale di chiave 2, cioè l'assegnamento corrente di stderror

Si noti che la numerazione prosegue dai file standard:
stdin 0, stdout 1, stderr 2, ...

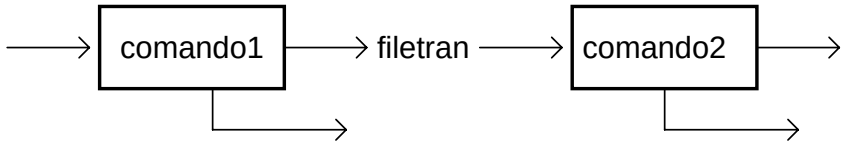
PIPE

COLLEGAMENTO AUTOMATICO di comandi

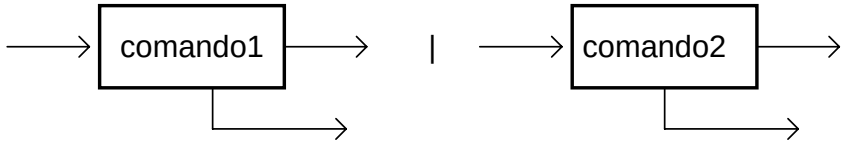
la PIPE **collega**

l'***output*** di un comando con
l'***input*** del comando successivo

SCHEMA IMPLEMENTATIVO



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



<comando1> **|** <comando2>

REALIZZAZIONE con FILE TEMPORANEI ==> DOS

in **UNIX** ==> PIPE come **COSTRUTTO PARALLELO**

le pipe sono svolte in modo parallelo
ogni comando è mappato in un processo

I comandi della pipe procedono in parallelo tra loro
==> NON si creano file temporanei

ESEMPI DI PIPING:

who | wc -l

cd /tmp
ls | sort | rev | more # nessuna remove

rev < file1 | sort | tee file | rev | more

Definizione di processo

Informalmente, 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, specificata in un eseguibile

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

Un sistema multitasking come UNIX mantiene **più** processi in esecuzione ***"concorrentemente"***

RIENTRANZA DELLO SHELL

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

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

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

Le invocazioni attivano un processo che esegue lo shell

Gli shell sono RIENTRANTI

RIENTRANZA ==>
più processi possono condividere il codice senza errori ed interferenze

sh
sh
csh
ps # quanti processi si vedono?

METACARATTERI o WILDCARD

Il comando prevede una sintassi ==>
lo SHELL riconosce **caratteri speciali** (WILD CARD) per formare espressioni regolari per il **pattern matching** in genere sempre sui nomi di file

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

? *un qualunque carattere* in un nome di file

[$c_1c_2c_3$] *un qualunque carattere di quelli nell'insieme* in un nome di file. Anche **range** di valori: **[c_0-c_n]**
Per esempio *ls [q-s]** lista i file con nomi che iniziano con almeno un carattere compreso tra q e s

[!$c_1c_2c_3$] *nessun carattere di quelli nell'insieme*
Esempio *ls *[!0-9]* lista i nomi dei file che terminano con caratteri non numerici

commento fino alla fine della linea

**** non interpretare il carattere successivo come speciale (**[]!*?**)

Esempio *ls *** lista i nomi dei file che contengono il carattere * in qualunque posizione

Il comando **echo** scrive la stringa successiva
echo * #lista tutti i nomi di file del direttorio corrente

ls * lista i file del direttorio corrente (nel caso di direttori questi vengono listati a loro volta)

ls [a-p,1-7]*[cfd]?
riporta i file i cui nomi hanno come iniziale un carattere compreso tra 'a e 'p' e tra 1 e 7
Il penultimo carattere deve essere c, f, oppure d.

echo * esegue l'echo del carattere '*', privato del suo significato

ls *[!*\?]* lista tutti i file del direttorio corrente che non contengono una wildcard * o ?

ls /*/*/* lista tutti i file dei direttori di secondo livello a partire dalla root (entra nei direttori a sua volta)

ls -d /*/*/* lista tutti i file dei direttori di secondo livello a partire dalla root (i direttori sono trattati come i file)

ls [a-z]*[0-9]*[A-Z] lista i file i cui nomi iniziano con una minuscola, terminano con una maiuscola e contengono almeno un carattere numerico

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

ls [a-z,A-Z,0-9][a-zA-Z0-9] lista i file con nomi di due caratteri alfanumerici (entrando nei direttorio)

echo [a-z,A-Z,0-9][a-zA-Z0-9] lista i file con nomi di due caratteri alfanumerici

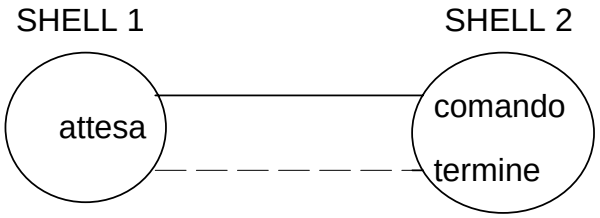
ESECUZIONE di COMANDI in SHELL

In UNIX *(quasi)* ogni comando viene eseguito da un nuovo shell

lo **shell attivo** mette in esecuzione un **secondo shell**

Il **secondo shell**
esegue le sostituzioni
ricerca il comando
esegue il comando

Lo **shell padre** attende il completamento dell'esecuzione del sottoshell (comportamento **sincrono**)



AMBIENTE DI SHELL:
insieme di variabili di shell
Per esempio, una variabile registra il **direttorio corrente**
Inoltre, ogni utente specifica come fare la *ricerca dei comandi* nei vari direttori del file system
la variabile **PATH** indica i direttori in cui ricercare ogni comando da eseguire
la variabile **HOME** indica il direttorio di accesso iniziale
etc.

SCHEMA di un PROCESSORE COMANDI

procedure **shell** (ambiente, filecomandi);
< eredita ambiente (esportato) del padre, via copia>
begin
repeat
<leggi comando da filecomandi>
if < è **comandoambiente**> *then*
<modifica direttamente ambiente>;
else if <è **comandoeseguibile**> *then*
<esecuzione del comando>
<attraverso un nuovo shell>
<che esegue il comando>
shell (ambiente, comandoeseguibile);
else if <è **nuovofilecomandi**> *then*
shell (ambiente, nuovofilecomandi);
else < errore>;
end if
until <fine file>
end shell;

Per abortire il comando corrente
<CTRL> <C>

Si noti la omogeneità di trattamento di comandi eseguibili e file comandi
*L'ambiente passato **per copia**: le **modifiche non** sono visibili allo **shell** che lo ha generato*

PROCESSORE VARIABILI <==> Ambiente
Lo shell è anche un **MACROPROCESSORE**

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

riferimento sinistro (no blank) riferimento destro
nomevariabile = $nomevariabile

Prima della esecuzione, il comando viene scandito (*parsing*), alla ricerca di caratteri speciali (*, ?, $, etc.)
Questi producono sostituzioni:

1) **Sostituzione delle variabili/parametri**
I nomi delle variabili (**$**nome) ==>
vengono espansi nei valori corrispondenti

```
x=12            # non si devono usare blank
echo $x         # produce 12
```

2) **Sostituzione dei comandi**
I comandi contenuti tra **‘ ‘** (backquote) ==>
sono eseguiti e ne viene prodotto il risultato.

```
echo ‘pwd‘      # stampa il direttorio corrente
‘pwd‘           # cosa si produce ?
‘pwd‘/fileexe.exe   # e così ?
```

3) **Sostituzione dei nomi di file**
I metacaratteri ***, ?, [ ]** ==>
sono espansi ai nomi di file nel file system secondo un pattern matching

Sostituzioni dello shell

Per ogni comando, oltre alle sostituzioni:
1) **delle variabili/parametri**
2) **dei comandi**
3) **dei nomi di file**

lo shell prepara i comandi come filtri
4) **ridirezione e piping di ingresso uscita (su file o dispositivo)**

Lo shell opera con sostituzioni testuali sul comando e prepara l’ambiente di esecuzione per il comando stesso

Comandi relativi all’espansione:

’ (quote) nessuna espansione (non 1, 2, 3)
" (doublequote) solo sostituzioni 1 e 2 (non la 3)

```
y=3
echo ’ * e $y ’   # produce * e $y
echo " * e $y "   # produce * e 3
```

Il processore esegue una sola espansione

```
y=3
x=’$y’
echo $x    # stampa $y
```

Per ottenere una ulteriore sostituzione,
bisogna **FORZARLA (uso di eval)**

```
eval echo $x     # stampa 3
```

eval consente una ulteriore fase di sostituzioni

Sostituzioni della shell: Esempi

1) sostituzione comandi
2) sostituzione variabili e parametri
3) espansione dei nomi di file

Scansione della linea di comando con più 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 il file del dir corrente tt) e prova quindi ad eseguire tt che non ha pero' i diritti di esecuzione

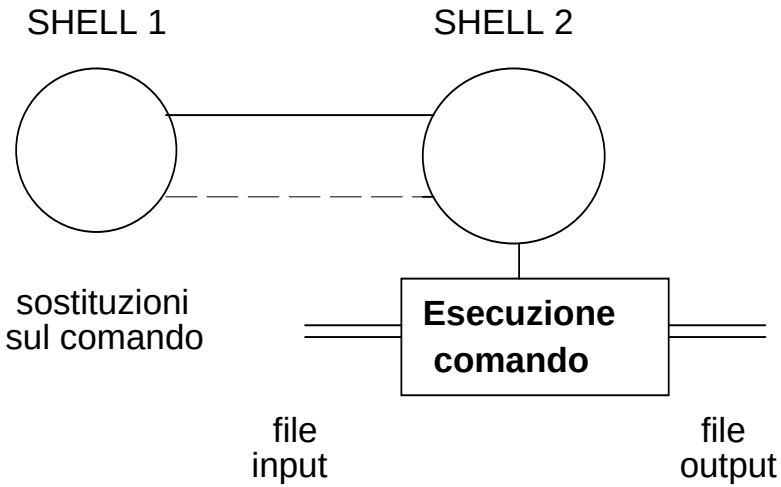
$ **rr='`pwd`'**
$ **echo $rr**
`pwd`

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

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

I metacaratteri per la ridirezione sono riconosciuti e trattati prima del lancio del comando

Esecuzione di un COMANDO



Il comando **esegue** avendo collegato
il proprio input al *file di input*
il proprio output al *file di output*
specificati dallo **shell di lancio**

ESECUZIONE IN PARALLELO &

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

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

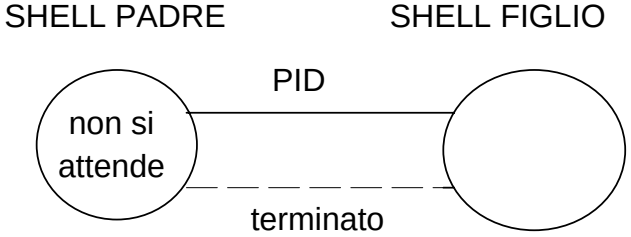
Lo shell invocante è immediatamente attivo

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

I processi in background si eliminano con
kill <PID> ; # o anche; **kill -9** <PID>

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

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



PROGRAMMAZIONE NELLO SHELL

PROCESSORE COMANDI
verso **linguaggi di programmazione**

Linguaggio Comandi

si possono scrivere ***file comandi,***
cioè statement per il controllo di flusso,
variabili
passaggio dei parametri

SHELL DI BOURNE
/bin/sh

usato come linguaggio prototipale di sistema
per il **rapido sviluppo** delle applicazioni

Lo shell è **case sensitive**

Per fare eseguire un file comandi (F):

- rendere eseguibile un file esistente e lanciarlo
  chmod +x F; F
- invocare direttamente uno shell che lo esegua
  sh F

VARIABILI

Sono disponibili **variabili** che contengono stringhe
NON è necessaria la definizione delle variabili
Il nome delle variabili è libero (alcune predefinite)
Il contenuto delle variabili indicato con metacarattere **$**

echo **$HOME** # stampa il direttorio di default
echo PATH **$PATH**
stampa PATH /:/bin:/usr/bin:$HOME:.
Il carattere ':' è il separatore dei vari campi.

Assegnamento

<variabile>**=**<valore> { niente bianchi, prego }

i=12
echo i $i
j=$i+1 # la sostituzione $i avviene sempre
echo $j
La prima echo fornisce la stringa: i 12; la seconda 12+1.
j="$i + 1" *# i " per avere l'assegnamento intero*

Le variabili sono trattate come stringhe di caratteri.

l="ls -F"; $l
l1="ls -lga a*"; $l1

l1 espande al momento della invocazione nel direttorio
corrente tutti i file che iniziano per a

echo $l; echo l; echo `$l`

PASSAGGIO PARAMETRI

comando argomento-1 argomento-2 ... **argomento-n**

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

variabile	**$0** il comando
primo argomento	**$1**
secondo argomento	**$2** etc

Si può assegnare ad un argomento?

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

DIR1 /usr/utente1 "*" (DIR1 contiene *cd $1 ; ls $2)*
il direttorio è cambiato solo per il sottoshell
lista dei file nel direttorio specificato
cosa produce **DIR1 /usr/utente1 ***

È possibile spostare gli argomenti verso il basso ➔ **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 al comando (***$0 escluso***)
$? il return code dell'ultimo comando eseguito
$$ il numero del processo in esecuzione

Si può fare anche dell'INPUT/OUTPUT

```
read   var1 var2 var3                  #input
echo   var1 vale $var1 e var2 $var2    #output
```

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

NON esistono variabili array
Gli shell non consentono questa possibilità direttamente

INPUT assegnamento di valore a variabile vettoriale

```
i=3;    read array$i
eval    array$i=espressione          # funziona!
eval    array\$i=espressione         # ed anche queste
eval    array‘echo $i`=espressione
```

OUTPUT: valore della variabile vettoriale

```
eval    var=\$array$i
eval    echo\$array$i
```


STRUTTURE DI CONTROLLO

principali statement di controllo di flusso

Ogni statement ==> in uscita restituisce in **?**
valore di stato di completamento del comando
$? può essere riutilizzato in espressioni
o per il controllo di flusso successivo

Stato **vale zero** ==> comando OK
valore positivo ==> errore

cp a.com b.com
se il comando non è riuscito
allora *stato errore* (valore > 0)
p.e. a.com non esiste ==> il ritorno codice di errore
altrimenti *stato successo* (valore 0)

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

expr $i + 1
 # stato OK (valore 0) se espressione aritmetica e != 0
echo stato di ritorno $?
cosa vale lo stato se i è uguale a *12, 1ab, 0, -1*

Si confronti con
expr $i + 1 > /dev/null
expr $i + 1 > /dev/null 2> /dev/null
expr $i + 1 > /dev/null 2>& 1

TEST

Comando per la valutazione di una espressione

forma generale di valutazione di una espressione
test -<opzioni> <nomefile>; **test <condizioni>**
ritorna uno stato uguale o diverso da zero

test **-f** <nomefile> esistenza di file
-d <nomefile> per direttori
-r <nomefile> diritto di lettura sul file (**-w** e **-x**)
...
test <stringa1> **=** <stringa2> #anche !=
valuta se due stringhe sono uguali
(qui i bianchi sono necessari: s1 **=** s2)
test **-z** <stringa1>
valuta se la stringa è nulla
test <stringa1>
valuta se la stringa non è nulla
test <numero1> [**-eq -ne -gt -ge -lt -le**] <numero2>
confronta tra loro due stringhe numeriche, usando uno degli operatori relazionali indicati

Espressioni booleane
! not monadico **-a** and **-o** or

Per le **espressioni aritmetiche ==> expr**
valuta l'espressione aritmetiche e si produce un valore
expr 24 - 12 + 65 fornisce 77.

```
i=12; j="$i + 1"; echo $j      ==> 12 + 1
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 ;**
la condizione è il valore in uscita dall'ultimo comando

```
  # fileinutile
  if test $1 = si -a $2 -le 24
      then echo si
      else echo no
  fi
```

Esempio di invocazione: fileinutile si 12 ==> *stampa* si

```
if  test $# -ge 1;   then echo OK
                     else echo Almeno un argomento
fi
if  test $variabile; then variabile=$var
fi  # per dare valore alla variabile una volta
```

Almeno un argomento
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
    <pattern-1>) <comandi> ;;
                    ...
    <pattern-i> | <pattern-j> | pattern-k>) <comandi>;;
                    ...
    <pattern-n>) <comandi> ;;
esac
```

alternativa multipla, secondo il valore in var

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

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

Ancora controllo dei parametri

```
case $# in
  0) echo Usage is: $0 file etc  >&2
     exit 2;; # si esce
  *) ;;
esac
```

...

RIPETIZIONI ENUMERATIVE

```
for <var> [in <list>]
    do
        <comandi>
    done
```

scansione della lista <list> e ripetizione del ciclo per ogni stringa nella lista

```
for i in *
  #esegue per tutti i file nel direttorio corrente

for i              # cioè in $*
  #esegue per tutti i parametri  di invocazione

for i in `ls s*`
 do <comandi>
done

 for i in `cat file1`
 do <comandi per ogni parola del file file1>
done

#file crea
for i          # cioè in $*
  do > $i;   # ridirezione di output su $i con chiusura
done
```

RIPETIZIONI NON ENUMERATIVE

while <lista-comandi>
do
<comandi>
done

Si ripete fintanto che il valore di **stato di ritorno** dell'ultimo comando della lista è **successo** (ossia uguale a zero)

```
#file ce; invocazione   ce nomefile
while test ! -f $1
   do sleep 10
done
```

until <lista-comandi>
do
<comandi>
done

```
#file ceutente; invocazione     ceutente nomeutente
until who | grep $1
  do sleep 10
  done
# mail all'utente
```

Uscite anomale
- vedi C ==> **continue**, **break** (e **return**, vedi man)
- **exit** [**status**] ==> (default 0) funzione primitiva di UNIX

Le parole chiave (do, then, fi, etc.) devono essere
o **a linea nuova** o **dopo il separatore ;**

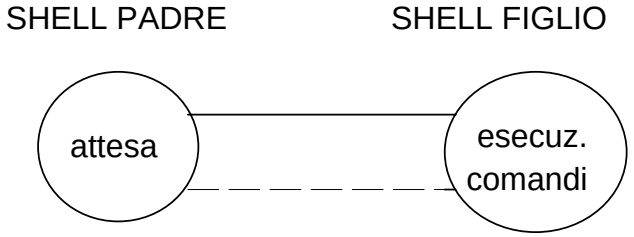
AMBIENTE di un FILE COMANDI

Si noti l'*ambiente* dei file comandi composto di:
- ***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
solo l'ambiente predefinito iniziale viene **copiato** ai figli **con i valori iniziali.**
In caso di aggiunta di **nuove variabili o di modifica di variabili predefinite**, i nuovi valori sono **copiati solo se** le **variabili** sono **esplicitamente esportate**

L'ambiente padre è preservato in **ogni caso**

OGNI COMANDO è ESEGUITO da un NUOVO e DISTINTO SHELL
che ottiene una copia (parziale) dell'ambiente del padre ma non può modificarla



export PATH, varnuova, varmia

Non esiste modo di **riportare variabili** (o altro) dal **figlio al padre** (UNIX suggerisce la **condivisione di file**)

IL TRATTAMENTO DEI SEGNALI

Un programma Shell tiene anche conto di
EVENTI ASICRONI

I segnali UNIX sono **eventi**
a cui si può **associare un gestore**

Lo statement shell
trap comandi numerosegnale

associa al segnale specificato il comando (o i comandi) specificati
All'arrivo del segnale si eseguono i comandi specificati

trap 'rm /tmp/*.*; exit' 2
associa al segnale 2 la ripulitura del direttorio tmp
vi si inseriscono i file temporanei

Funzioni **differenziate** associate ai **diversi segnali**
In genere:
- **0** fine del file
- **2** <CTRL><C>: interrupt da tastiera;
(HP: <Shift><Del>)
- **3** <CTRL><Z>: stop da tastiera

Si può associare a ciascuno una stampa differenziata in base alla causa di interruzione
o fornire una nuova azione

Le azioni conseguenti sono

ignorare
```
trap ’ ’ 1  # o anche:
```

azione utente
```
trap ’ls; exit’ 15 # da kill
trap ’ls’ 15
```

default: TERMINAZIONE
```
trap 2
```

```
# file scan: dal direttorio corrente per ogni direttorio

d=’pwd’
for i in *
do
  if test -d $d/$i
    then
      cd $d/$i
      while echo " $i: direttorio"
        trap exit 2 # si esce dal comando se interrupt
        read x
      do
        trap ’  ’ 2  # ignorati
        eval $x
      done
  fi
done
```

ESEMPI

```
case $# in
  0) echo Usage is: $0 file [file]
     exit 1;;
  *) ;;
esac
for i in $*
do                  # in $* poteva essere omesso
  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
  read answer
  case  $answer in
     y*| Y*| s* | 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* | grep  <stringa>
```



```
# file   cercadir; invocazionecercadir [nomeassoluto] 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?
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
   fi
done
```

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 ed a lanciare una invocazione per ogni sottodirettorio trovato: *non si devono prevedere così le profondità dell'albero dei direttori*

Comandi per lo sviluppo di un programma

CC: Compilatore C
cc -o nomefileeseguibile -g nomefile1 ... nomefilen
file oggetto (**.o**)
qualificatore **-g** le tavole per il debugger sorgente.

I nomefile possono essere anche file in forma oggetto

cb: C program Beautifier
migliora indentazione e paragrafazione

lint: fornisce indicazioni di errore
lint nomefile1 ... nomefilen

cflow: crea il grafo di chiamate dei vari file in ingresso
cflow nomfilesorgente1 ... nomefilesorgente.n

cxref: costruisce una tavola dei riferimenti
cxref nomefilesorgente

ctrace: controlla la esecuzione
(attenzione: modifica il file sorgente)

Uso suggerito:
ctrace sorgente.c > temp.c {un file per volta}
cc temp.c { uscita a default su a.out }
a.out { esecuzione controllata }

error: gestisce gli errori
accesso ad errori sui file sorgente
per esempio entrando in editor sul file

Uso suggerito:
cc ... |& error -q -v {gli errori in pipe su error}

dbx: Debugger Simbolico

I file eseguibili devono essere stati prodotti con l'opzione **-g** dal compilatore cc.

Questo debugger rende possibile l'accesso alla versione sorgente del programma, che può essere consultata, con comandi:

list lineainizio, lineafine
edit filename
edit nomefunzione
func nomefunzione {dichiara la funzione corrente}

Si possono analizzare e variare i dati:

display espressioneconvariabili
undisplay espressioneconvariabili
whatis identificatore {fornisce il tipo della variabile}
which identificatore {fornisce indicazioni sulla variabile correntemente specificata}
whereis identificatore {fornisce indicazioni relative a variabili o entità con quel nome}
dump funzione {stampa i valori ed i parametri della funzione stessa se attiva}

Esecuzione controllata:

run argomenti {inizia la esecuzione}
stop at linea
stop if condizione
stop in proc
{varie forme di breakpoint, per arrestare il programma ed iniziare il monitoraggio}
cont {continua l'esecuzione fino al prossimo breakpoint}
step n
{passo passo, senza n; altrimenti esegui n linee sorgente}
next n
{come sopra, saltando la esecuzione delle funzioni}

Si usino tutti gli strumenti per lo sviluppo di un programma di prova.

make: correlatore di strumenti

Il **make** gestisce in modo 'automatico' le correlazioni di più file che devono costituire un programma.
Lo sviluppo diventa **automatico** dopo aver specificato le dipendenze

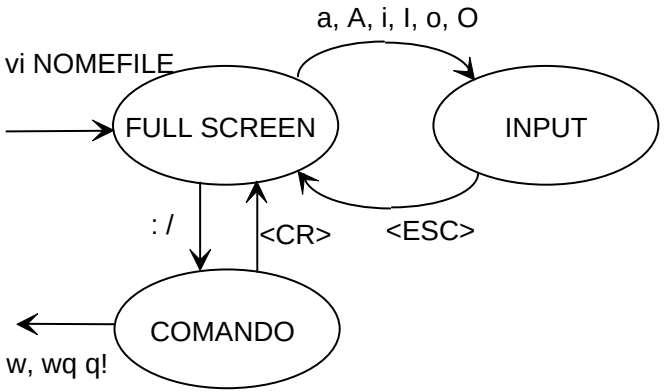
EDITOR STANDARD DI UNIX: VI

vi (**VI**sual display editor)
Editor standard di UNIX

non basato sul concetto di WINDOW (finestra)
ma usabile da un qualunque tipo di terminale
Un Editor **Full-screen** ed **a linee** come **comandi**

3 stati di VI:
1) lo stato comandi display (*full screen*);
2) lo stato riga comandi (*comando*);
3) lo stato di input (*input*).

Diagramma a stati del VI:



legenda: Maiuscole prima del cursore, minuscole dopo il cursore

comando per invocarlo
vi <nomefile>
il file di nome <nomefile> viene aperto se esiste, creato altrimenti. Inizio editing ed accettazione comandi

CASO NUOVO FILE ===> vi prova.c

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

RIGA COMANDI

CASO FILE ESISTENTE===> vi prova.c

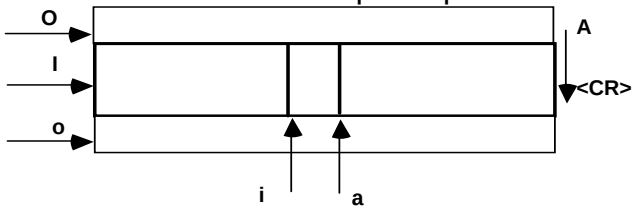
```
# include <stdio.h>

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 composta 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

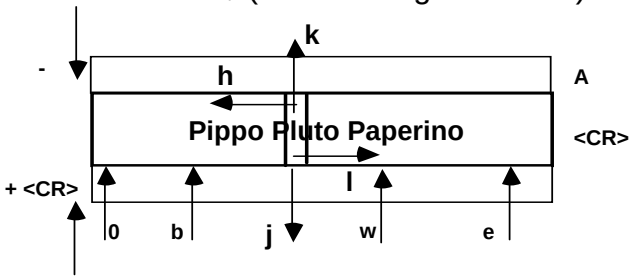
Si passa da stato comandi a input in posizioni diverse



COMANDI FULL SCREEN

MOVIMENTO

di un carattere	quattro tasti sicuri (h,j,k,l) ← h ↓ j ↑ k → l **<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 il numero di volte che il comando deve essere ripetuto

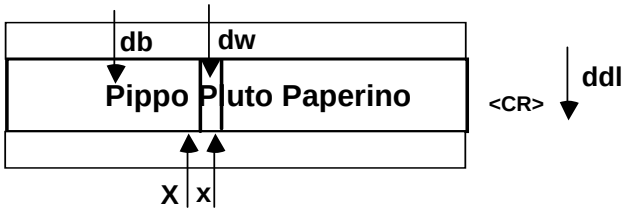
Ad esempio: **3w** sposta il cursore in avanti di 3 parole

VISUALIZZAZIONE

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

DISTRUZIONE

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 **nY nyy**
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) MOVIMENTO 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 della RIGA COMANDO: (stato 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 **:$**

DISTRUZIONE

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
scrittura su file superando i normali controlli (se possibile)
:w! nomefile
specifica di opzioni **:se**t **opzione**
visualizza i numeri di riga **:se**t **nu**mber
elimina i numeri di riga **:se**t **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 riporterà 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 approfondirne la conoscenza.

Inoltre si usino le invocazioni di **shell** per il **debug**:

sh -v e **sh -x**

shell in forma verbosa (-v) e
con espansione dei comandi prima della esecuzione (-x)