IL SISTEMA UNIX

Prima Parte

Visibilità utente e comandi

Roberto Polillo

Corso di Sistemi Operativi
Corso di Laurea in Informatica
Università di Milano

Ultimo aggiornamento: febbraio 1998

INDICE

1. INTRODUZIONE A UNIX

CHE COS'È UNIX

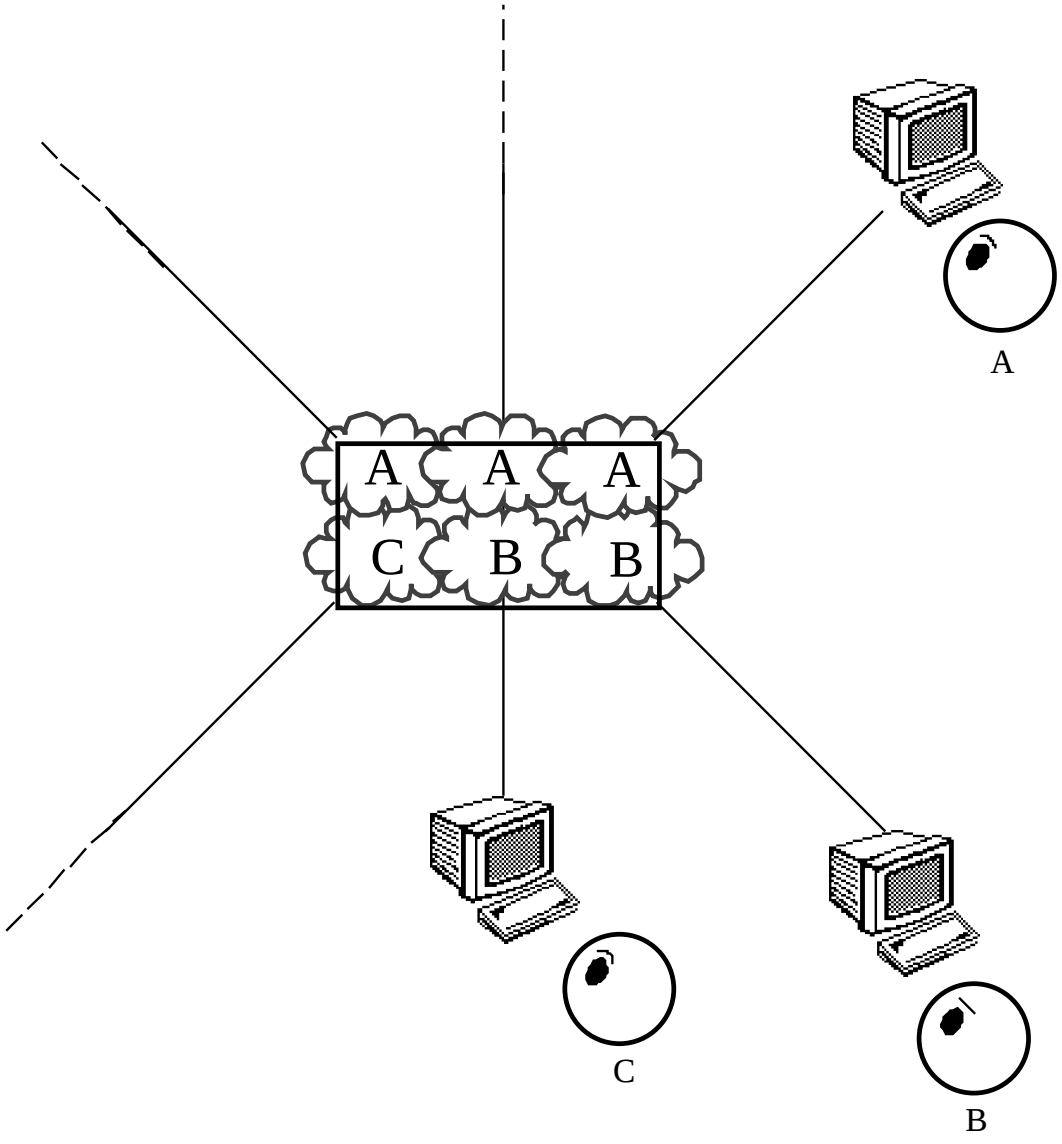
Unix è il nome di una **famiglia** di sistemi operativi, disponibili su molti elaboratori

Nome del sistema	**Fornitore**
AIX	IBM
A/UX	Apple
BSD Berkeley	Univ.California
HP-UX	Hewlett-Packard
Linux	public-domain
OSF/1	DEC
SCO Unix Operation	Santa Cruz
SunOS	SUN
Solaris	SUN
Ultrix	DEC
System V	vari
.....	

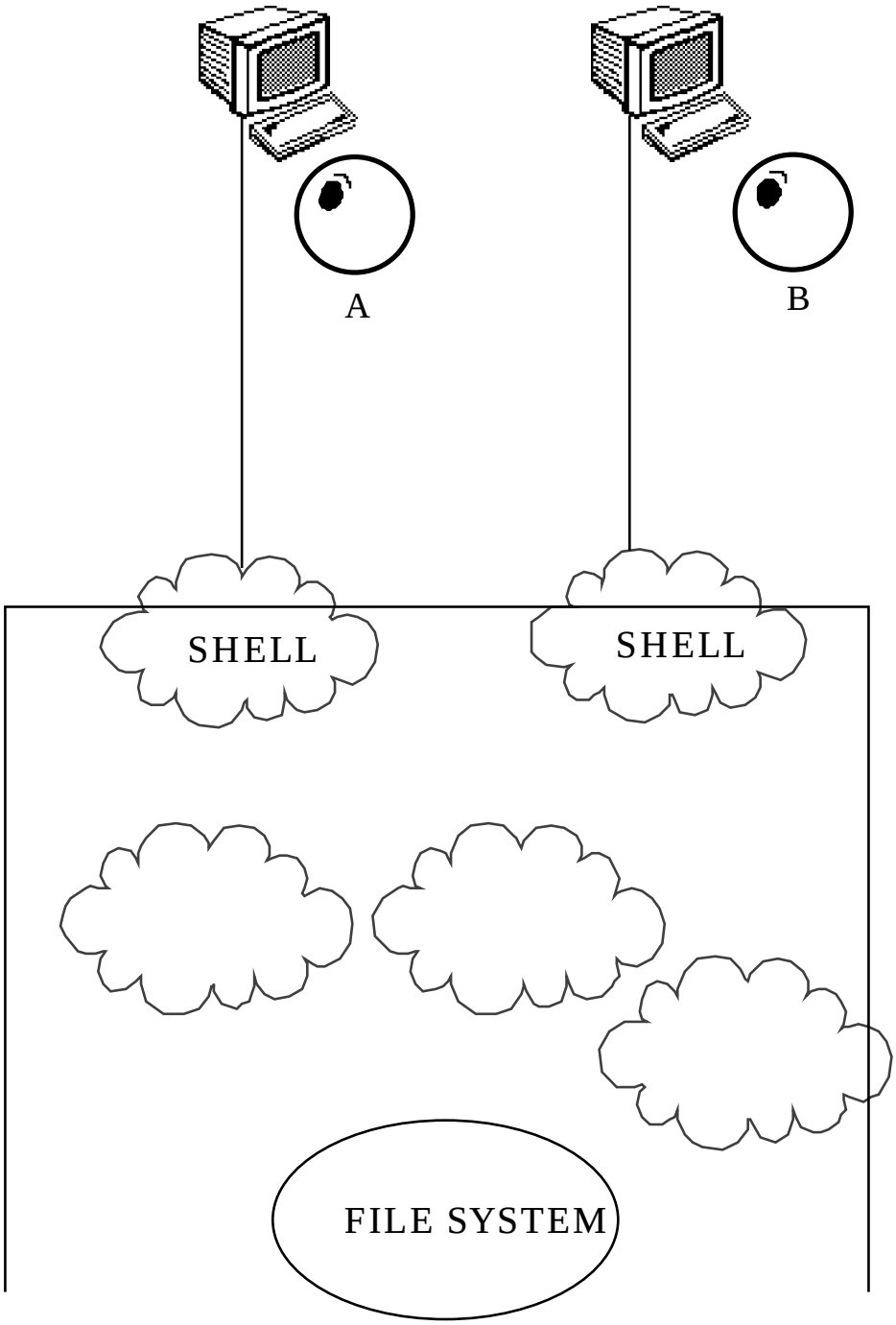
UNIX: CARATTERISTICHE GENERALI

È un sistema:

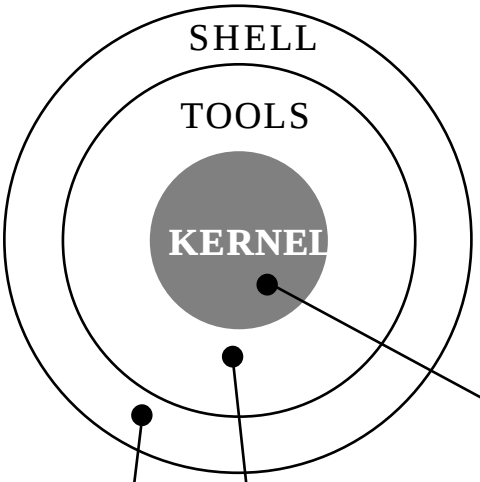
- Multi-user
- Multi-processing
- Time-sharing



ARCHITETTURA DI UNIX: GENERALITÀ



ARCHITETTURA DI UNIX: VISIONE A LIVELLI



Realizza una macchina virtuale che fornisce una interfaccia di programmazione semplice ed elegante

È l'insieme dei programmi applicativi che realizzano tutti i "comandi" del sistema, utilizzando le primitive del Kernel

È un programma applicativo come gli altri, che realizza l'interfaccia utente
Sono disponibili varie shell:
ogni singolo utente può utilizzare la shell che desidera

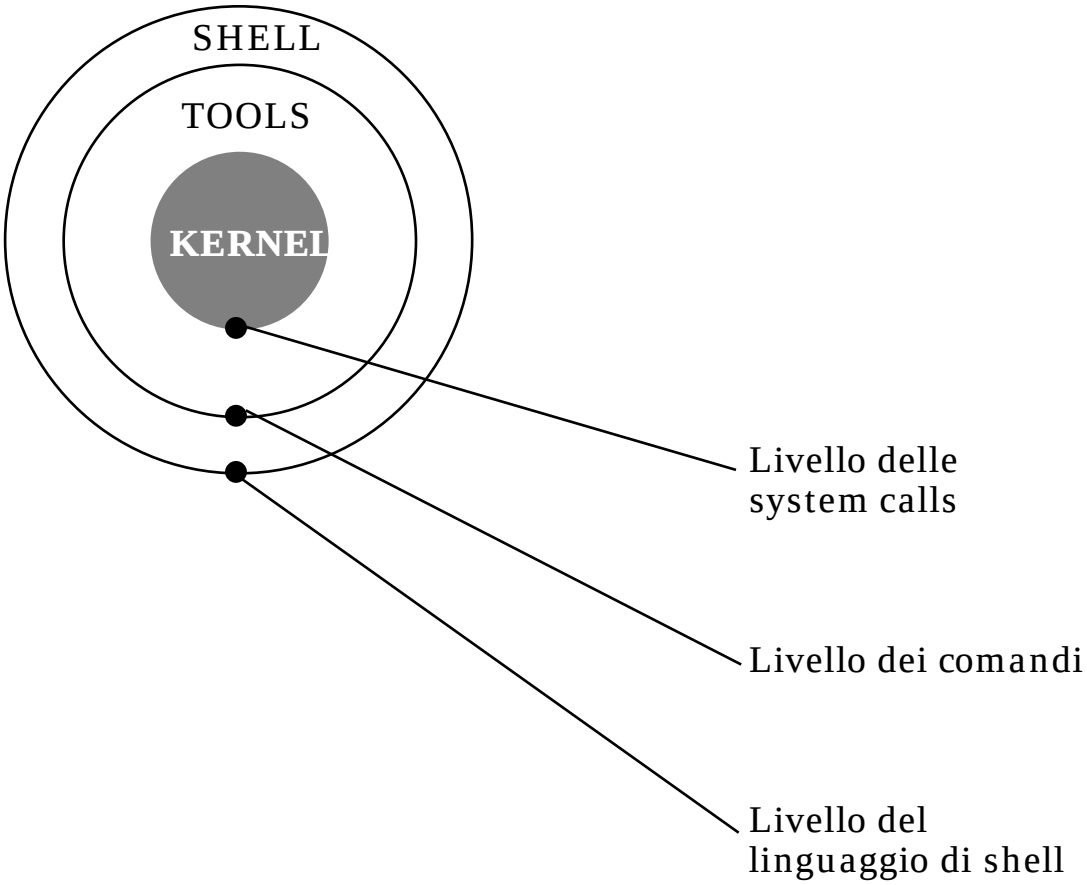
IL KERNEL UNIX: FUNZIONI

- Gestione degli interrupt e degli errori
- Schedulazione e gestione dei processi
- Gestione della memoria
- Gestione dei files
- Servizi di input/output
- Servizi di gestione data e ora
- Accounting di sistema

UNIX: CLASSI DI COMANDI

- Amministrazione del sistema
- Gestione dei files
- Elaborazione testi
- Sviluppo software
- Comunicazione
- ...

LIVELLI DI VISIBILITÀ E INTERFACCE



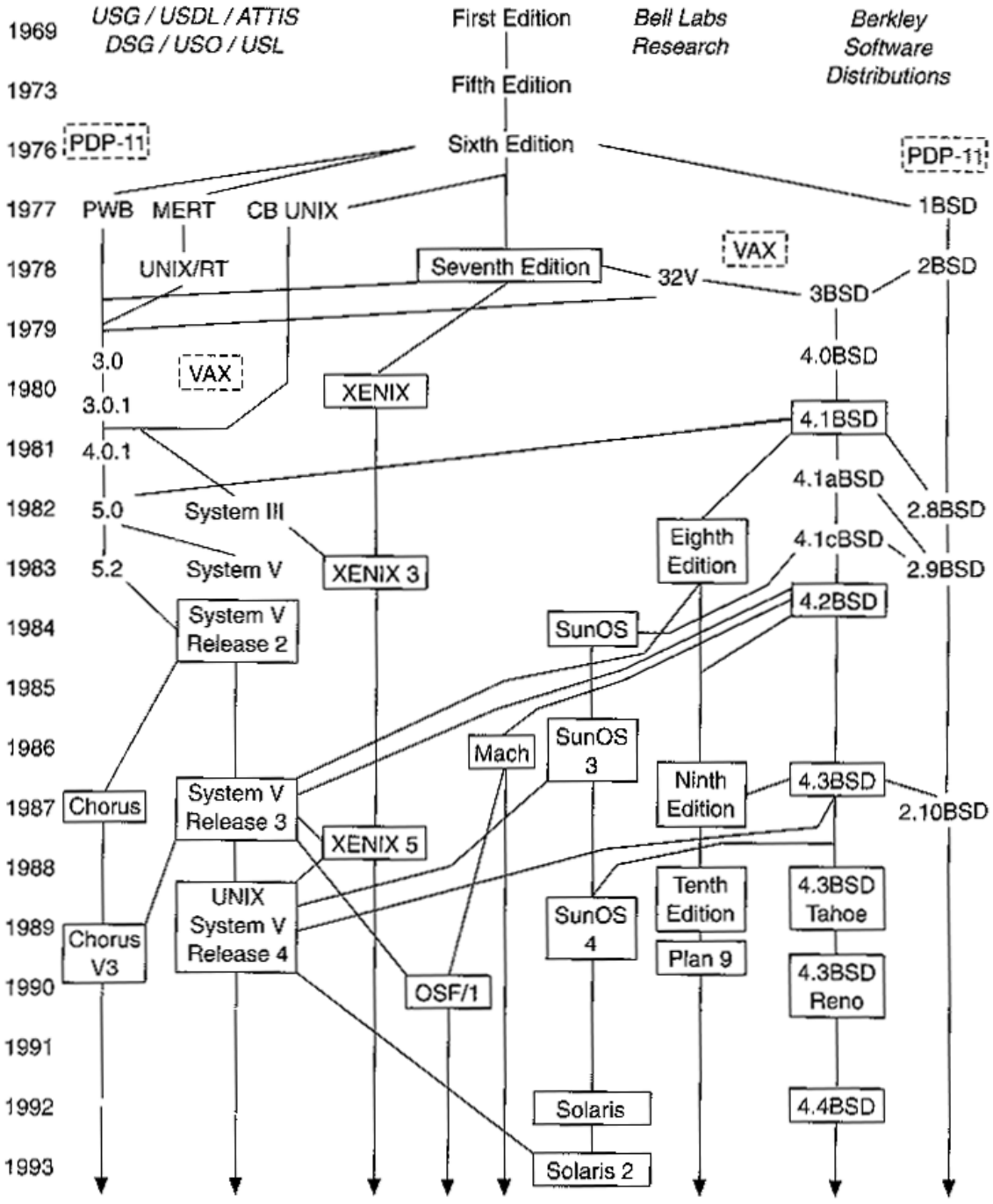
STORIA DI UNIX

- La prima versione di Unix fu sviluppata da Ken Thompson e Dennis Ritchie, ai Bell Laboratories, nel 1969

- La Sixth Edition fu la prima versione in commercio, nel 1976

- Nel 1978 venne rilasciata la Seventh edition, che chiude la fase iniziale di progettazione di Unix: inizia la proliferazione di dialetti

I DIALETTI DI UNIX: ALBERO GENEALOGICO



da: Silberschatz, Galvin, "Operating System Concepts", 1994

LA STANDARDIZZAZIONE DI UNIX

Dalla fine degli anni ’80 sono stati fatti vari sforzi per "standardizzare" Unix

L'obbiettivo è la **portabilità delle applicazioni** da un sistema a un altro, a livello sorgente (prevalen-temente):

- programmi C (prioritariamente)
- script di shell
- programmi in altri linguaggi

La competizione dei vari costruttori per il controllo dello Unix "standard" ha creato negli anni una situazione molto complessa

STANDARDS PRINCIPALI

- **C** (ANSI nel 1989, poi ISO)

  Definisce anche la C Standard Library

- **POSIX** (IEEE dal 1988, poi ISO)

  "**P**ortable **O**perating **S**ystem **I**nterface for Uni**x**"

  poi: "Portable Operating System Interface for Computer Environments"

- **XPG** (X/Open, dal 1989)

  "X/Open Portability Guide"

- **SVID** (AT&T, 1989)

  "System V Interface Definition"

LA FAMIGLIA POSIX (APR 97)

Codice	Titolo
1003.0	The Guide to Posix Open Systems Environments
1003.1	System Application Programming Interface (API)
1003.2	Shell and Utility Application Interface
1003.3	Test Methods - Measuring conformance to POSIX
1003.5	ADA Bindings
1003.9	Fortran Bindings
1003.10	Supercomputing Application Environment Profile (AEP)
1003.11	Transaction Processing Application Environment Profile (AEP)
1003.13	Real-time Application Profile
1003.14	Multiprocessing Application Environment Profile (AEP)
1003.16	Vis Bindings
1003.17	Directory Services and Name Space
1003.18	Posix Profile

POSIX: GLI STANDARD PRINCIPALI

1003.1(POSIX.1): System Application Programming Interface (API)

- dal 1990
- definisce l'interfaccia fra programmi applicativi e s.o., nei termini di una libreria di funzioni
- obbiettivo: portabilità dei programmi fra sistemi conformi a POSIX (a livello sorgente)

1003.2 (POSIX.2): Shell and Utility Application Interface

- dal 1992
- definisce il linguaggio di shell e i principali comandi
- obbiettivo: portabilità degli script di shell fra sistemi conformi a POSIX

LA "CULTURA" UNIX

Unix è molto più che una famiglia di sistemi operativi:

- è un enorme insieme di programmi
- ed una **cultura** basata su di essi

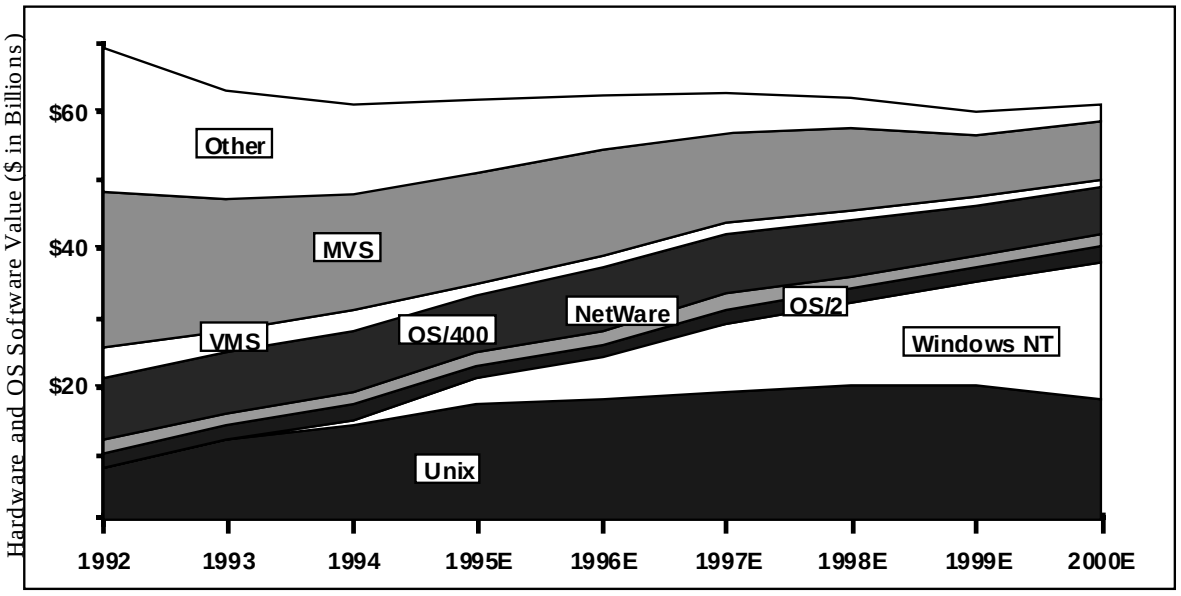
Lo scopo di questo corso è quello di fornire una introduzione a questa cultura, e **non** di spiegare l'uso di alcuni comandi, per i quali esistono ottimi manuali

Pertanto, nel seguito **non ci si riferisce ad alcuna specifica versione di Unix**, ma si introducono concetti e comandi in funzione del loro interesse generale

Gli esempi utilizzano il sistema SunOS 5.3

EVOLUZIONE DI UNIX

Worldwide Server/Host Spending by Operating System



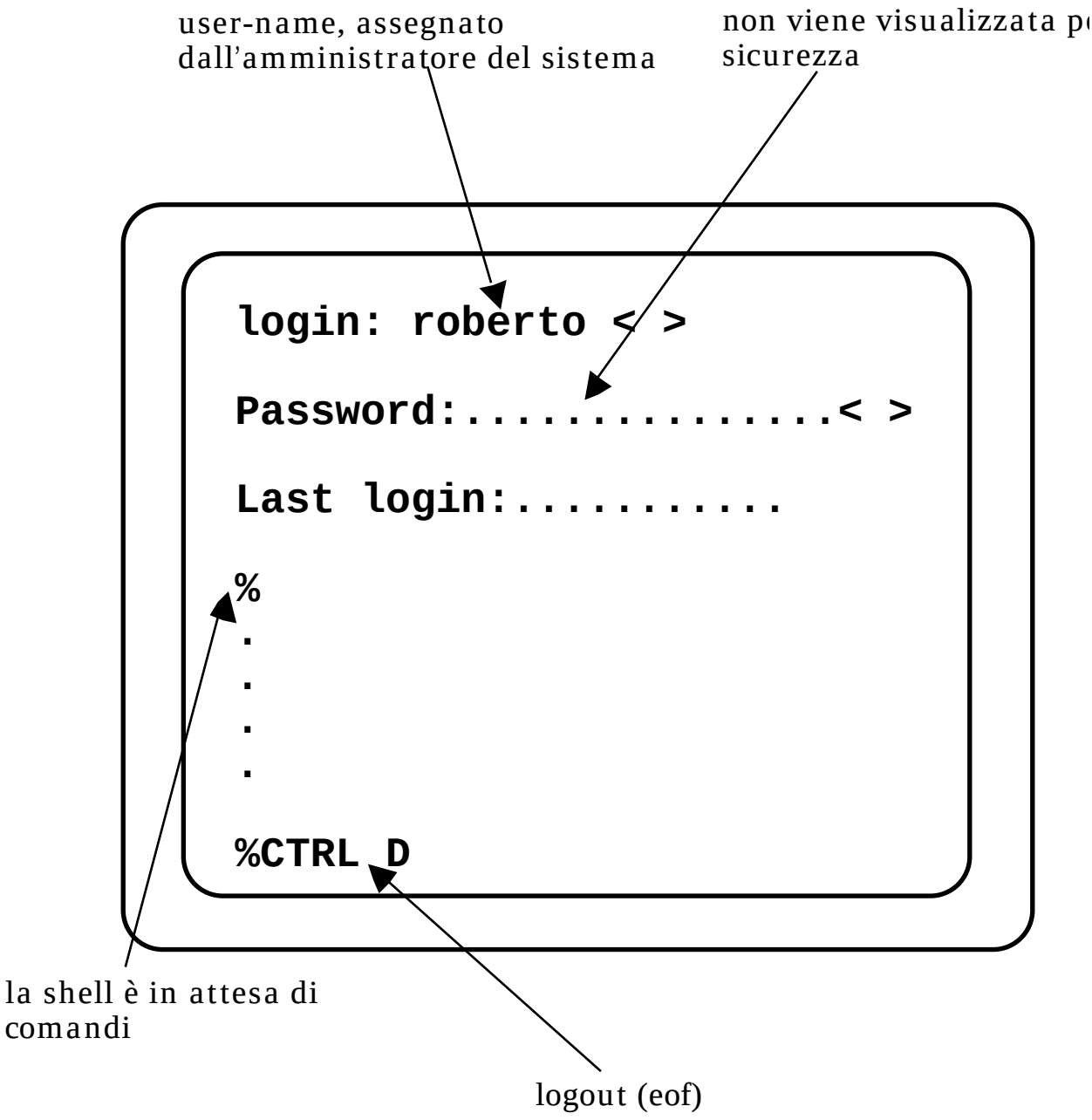
Source: Gartner Group, dicembre 1995

"Unix is simple and coherent, but it takes a genius (or, at any rate, a programmer) to understand and appreciate its simplicity"

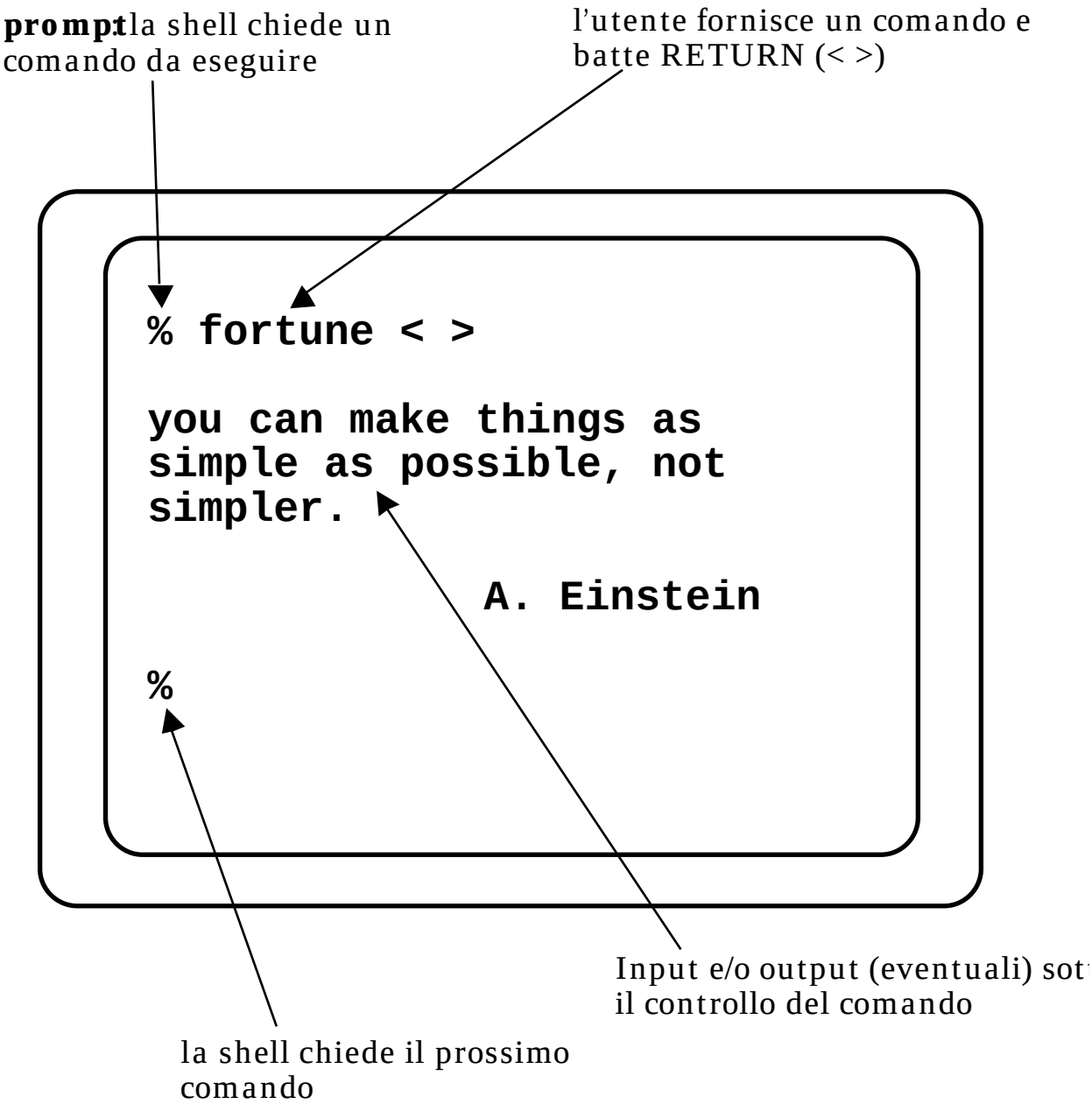
D. Ritchie

2. SESSIONE DI LAVORO IN UNIX

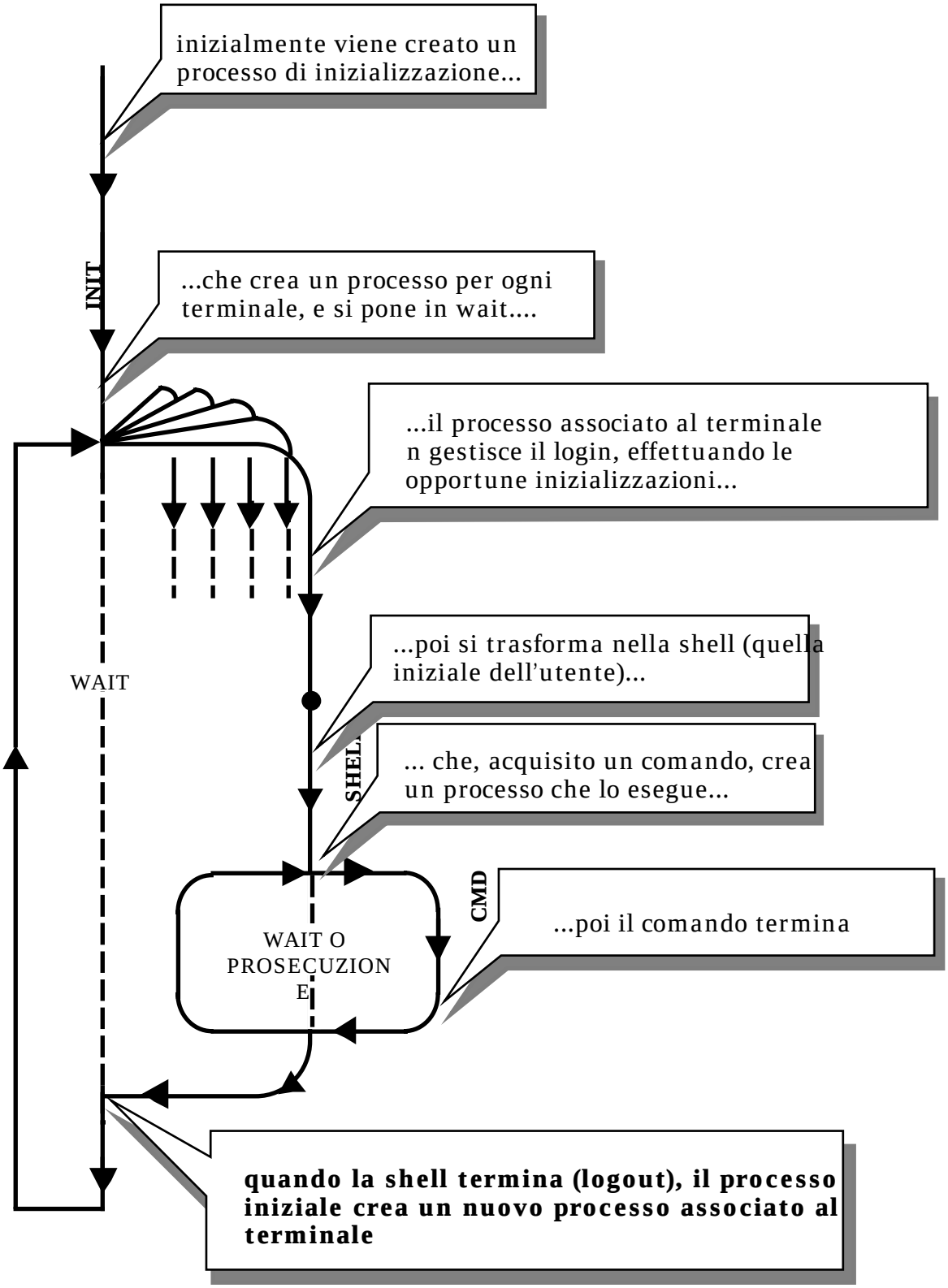
UN ESEMPIO DI SESSIONE



INTERFACCIA UTENTE



PROCESSI: VISIONE DINAMICA



FORMATO DEI COMANDI

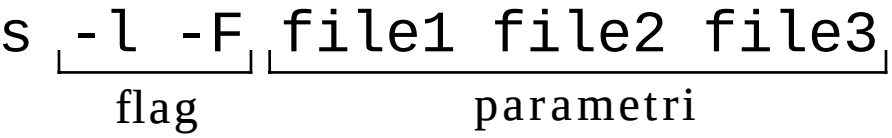
comando [*argomento ...*]

Gli *argomenti* possono essere:

- opzioni o flag (**-**)
- parametri

separati da almeno un *separatore*

Esempio:



Forme equivalenti:

```
ls -F -l file1 file2 file3
ls -lF file1 file2 file3
ls -Fl file1 file2 file3
```

CONVENZIONI NOTAZIONALI PER I COMANDI

termini in neretto	devono essere scritti esattamente come appaiono
termini in corsivo	devono essere rimpiazzati da valori opportuni
[ ]	significa “opzionale”
...	significa “una o più volte”

Esempio:

`ls` [**`-aAcCdfFgilLqrRstu1`**] [*`filename`*`...`]

IL COMANDO who

who[....][**am I**]

- **who**

  lista il nome, terminale, e data/ora di login di tutti gli utenti correnti

- **who am I**

  come sopra per l'utente che lo esegue

Esempio:

```
% who
marco    pts/0   Mar 4 20:24
(venere.inet.it)
roberto  pts/1   Mar 5 16:24  (194.20.15.99)
marco    pts/2   Mar 4 20:23
(venere.inet.it)
%

% who am I
roberto  pts/1   Mar  5 16:24
(194.20.15.99)
%
```

ESEMPIO: IL COMANDO `date`

`date``[…]`

Modifica o stampa (nel formato specificato) la data corrente

Esempio:

```
% date
Sun Mar  5 16:41:05 MET 1995
%
```

IL COMANDO `man`

Unix ha un manuale di riferimento, accessibile “in linea”, mediante il comando **`man`**

Il manuale è organizzato in **sezioni** e **sottosezio-ni**; ogni sezione è composta di **pagine** (logiche)

Ogni pagina descrive **un singolo argomento** (es.: un comando)

Esempio:

Sezione	**Contenuti**
1	Commands
2	System Calls
3	Library Functions
4	Administrative Files
5	Miscellaneous Information
6	Games
7	I/O and Special Files
8	Maintenance Commands

IL COMANDO `man`

man [*opzione...*] *titolo...*

Visualizza le pagine del manuale specificate mediante i suoi parametri

`-s` permette di specificare la sezione

Esempi:

```
%man who
....
%man -s 2 kill
....
%man -s 2 intro
```

Note:

- Se il numero di sezione non è specificato, viene selezionata la prima occorrenza
- Ogni sezione o sottosezione inizia con una pagina chiamata `intro`

ESEMPIO

```
% man man
man(1)     User Commands          man(1)

NAME
  man-find and display reference manual pages

SYNOPSIS
  man [ - ] [ -adFlrt ] [ -M path ] [ -T macro-package ]
     [ [ -s section ] title ... ] ...
  man [ -M path ] -k keyword ...
  man [ -M path ] -f filename ...

AVAILABILITY
  SUNWdoc

DESCRIPTION
  man displays information from  the  reference  manuals.
It
  displays  complete manual pages that you select by title,
or
  one-line summaries selected either by keyword (- k),  or
by
  the name of an associated file (-f).
<omissis>

OPTIONS
  -a Show all manual pages matching title within  the  MAN-
  PATH  search  path.   Manual pages are displayed in the
  order found.
<omissis>

USAGE
 Sections
  Entries in the reference manuals  are  organized  into
sec-
  tions. A section  name  consists of a major section name,
  typically a single digit, optionally followed by  a
subsec-
  tion  name,  typically  one  or  more letters.  An
unadorned
<omissis>

ENVIRONMENT
 MANPATH  A colon-separated list of  directories;  each
     directory   can   be   followed   by   a   comma-
     separated list  of  sections.   If  set,  its
     value overrides /usr/share/man as the default
<omissis>
```

ESEMPIO (SEGUE)

```
FILES
  /usr/share/man         root of  the  standard  manual
                       page directory subtree
  /usr/share/man/man?/*     unformatted manual entries
  /usr/share/man/cat?/*     nroffed manual entries
  /usr/share/man/fmt?/*     troffed manual entries
<omissis>

SEE ALSO
  apropos(1),  cat(1),  col(1),  eqn(1),  more(1),
nroff(1),
  refer(1),    tbl(1),    troff(1),    vgrind(1),
whatis(1),
  catman(1M), eqnchar(5), man(5)

NOTES
  Because troff is not 8-bit clean, man has not been  made
8-
  bit clean.
<omissis>

BUGS
  The manual is supposed to be reproducible either on a
photo-
  typesetter  or on an ASCII terminal.  However, on a
terminal
  some information (indicated by font changes,  for
instance)
  is lost.
<omissis>


Sun Microsystems    Last change: 14 Sep 1992
5


%
```

IL COMANDO `whatis`

whatis [*comando...*]

Mostra solo la sezione NAME della pagina del manuale (equivale a `man -f`)

Esempio

```
% whatis time date
time   time (1)    - time a command
time   time (2)    - get time
date   date (1)    - print and set the
date
%
```

IL COMANDO apropos

apropos [*word...*]

Cerca i comandi la cui descrizione nel manuale (NAME) contiene le parole specificate (equivale a man-k)

Il comando non distingue fra maiuscole e minuscole

Esempio

```
% apropos manual
catman   catman (1m)     - create the cat files for the
                           manual
man      man (1)         - find and display reference manual
                           pages
man      man (5)         - macros to format Reference Manual
                           pages
mansun   mansun (5)      - macros to format Reference Manual
                           pages
route    route (1m)      - manually manipulate the routing
                           tables
whereis  whereis (1b)    - locate the binary, source, and
                           manual page files for a command
%
```

3. IL FILE SYSTEM: GENERALITÀ

FILE SYSTEM UNIX: CARATTERISTICHE GENERALI

- Struttura gerarchica
- Files senza struttura ("byte stream")
- Protezione da accessi non autorizzati
- File & device independence
- Semplicità

FILES UNIX

I tipi (principali) di files sono tre:

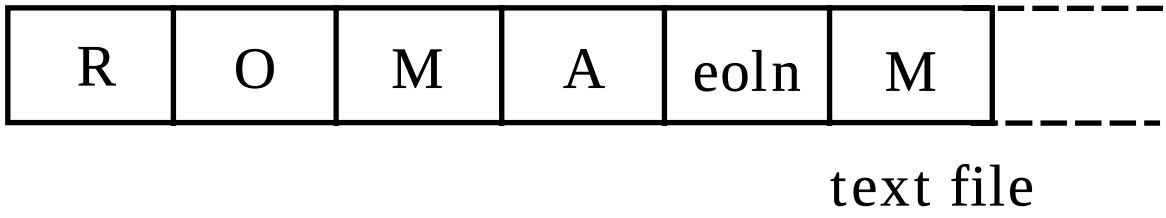
- **Files ordinari**
- **Directories**
- **Files speciali**

Il sistema assegna biunivocamente a ciascun file un identificatore numerico, detto **i-number** ("index-number"), che gli permette di rintracciarlo nel file system.

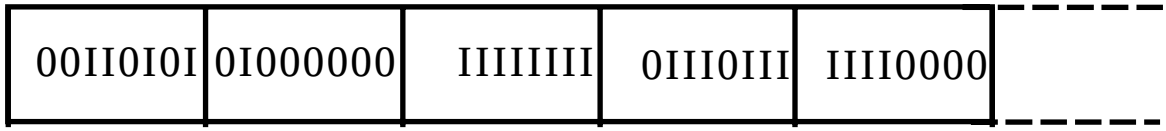
FILES ORDINARI

- Sono sequenze di byte ("byte stream")
- Possono contenere informazioni qualsiasi (dati, programmi sorgente, programmi oggetto,...)
- Il sistema non impone nessuna struttura

Esempi:



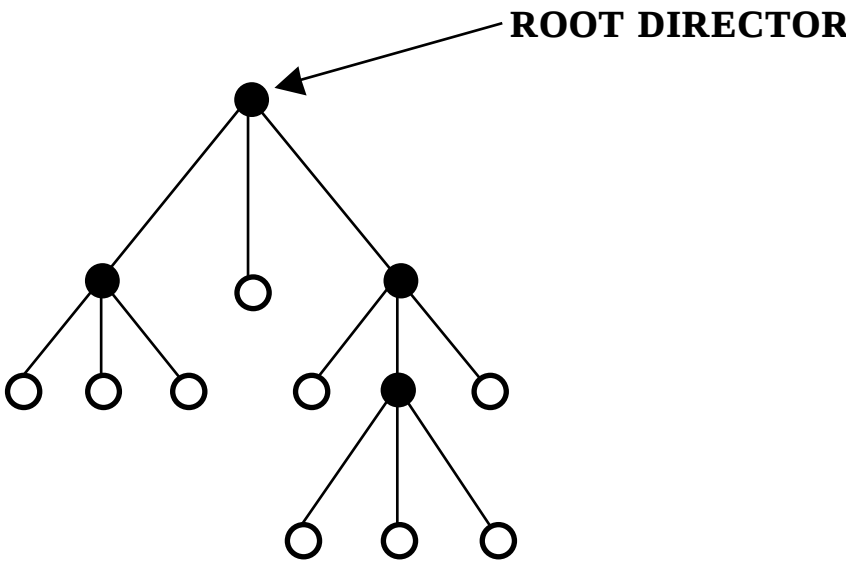
text file



binary file

ORGANIZZAZIONE DEI FILE

Per consentire all'utente di rintracciare facilmente i propri files, Unix permette di raggrupparli in **directories**, organizzate in una (unica) struttura gerarchica:

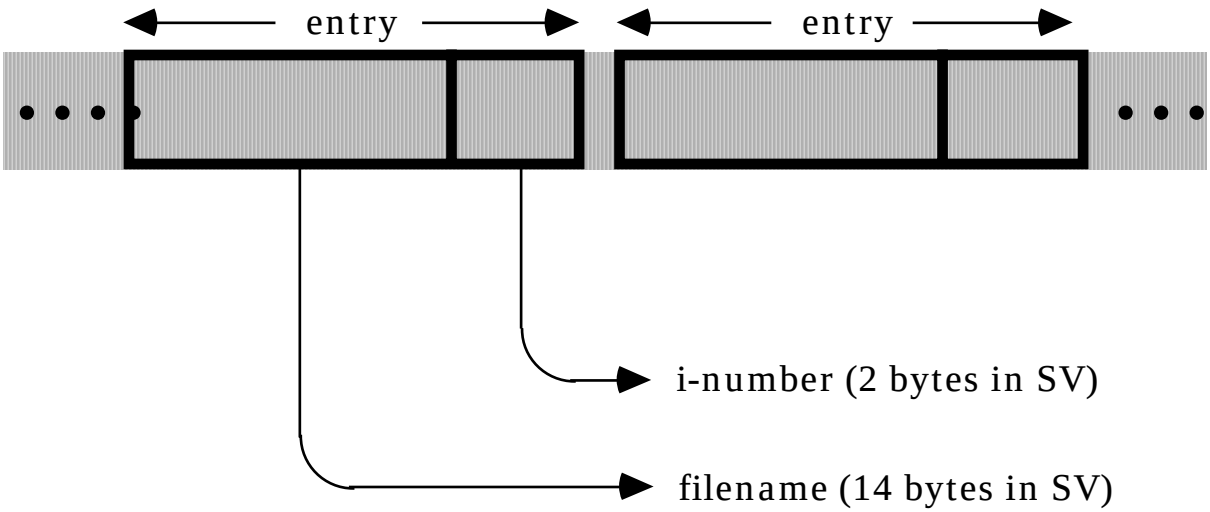


● : directory

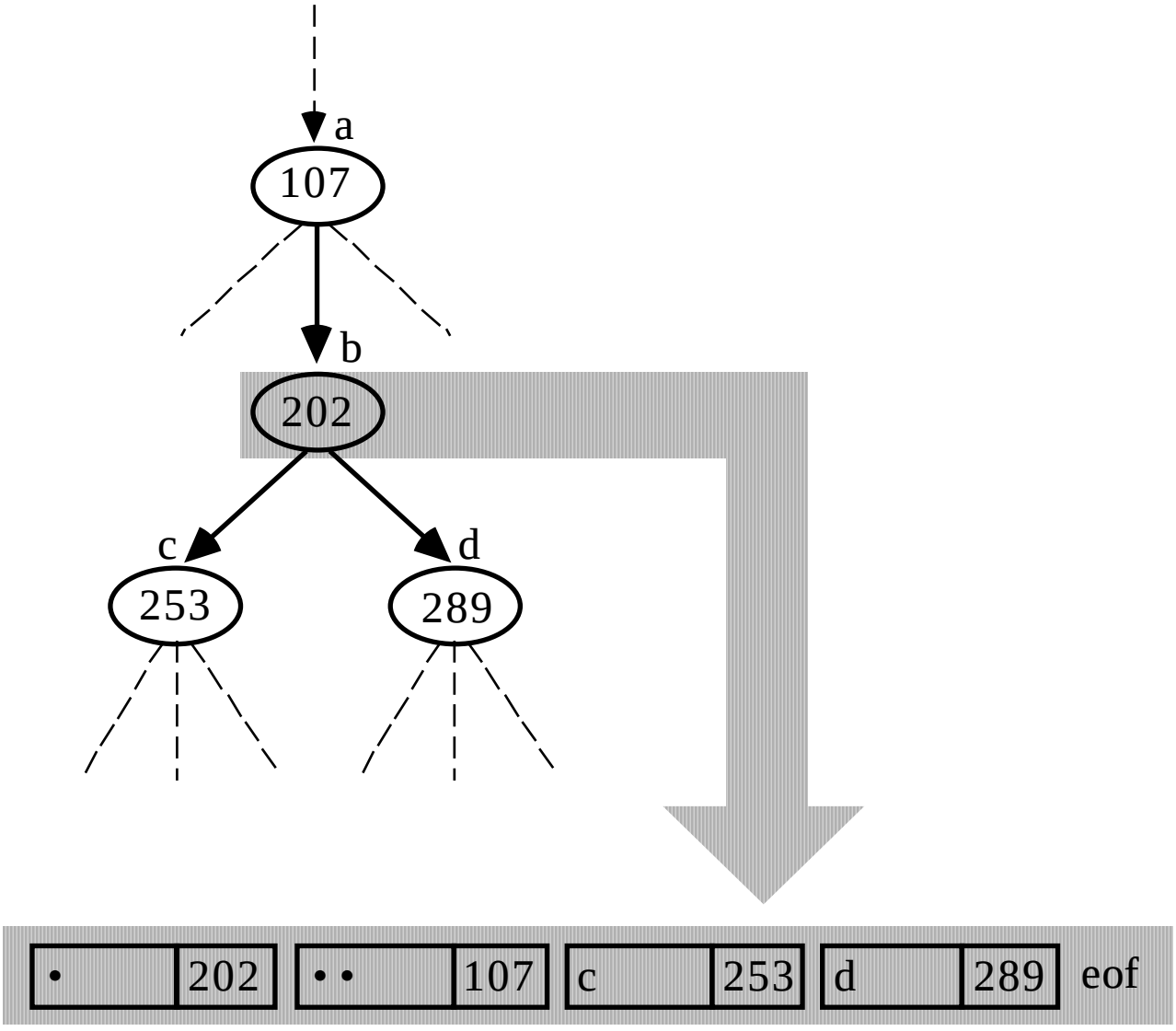
○ : file ordinario
directory (vuota)
file speciale

DIRECTORIES

- Sono sequenze di bytes come i files ordinari
- Differiscono dai files ordinari solo perché non possono essere scritte da programmi ordinari
- Il loro contenuto è una serie di **directory entries:**
- ... che definiscono l'associazione fra gli i-number (usati dal sistema) e i **filename** mnemonici (usati dall'utente):



ESEMPIO



Come si vede, ogni directory ha sempre almeno 2 entries:

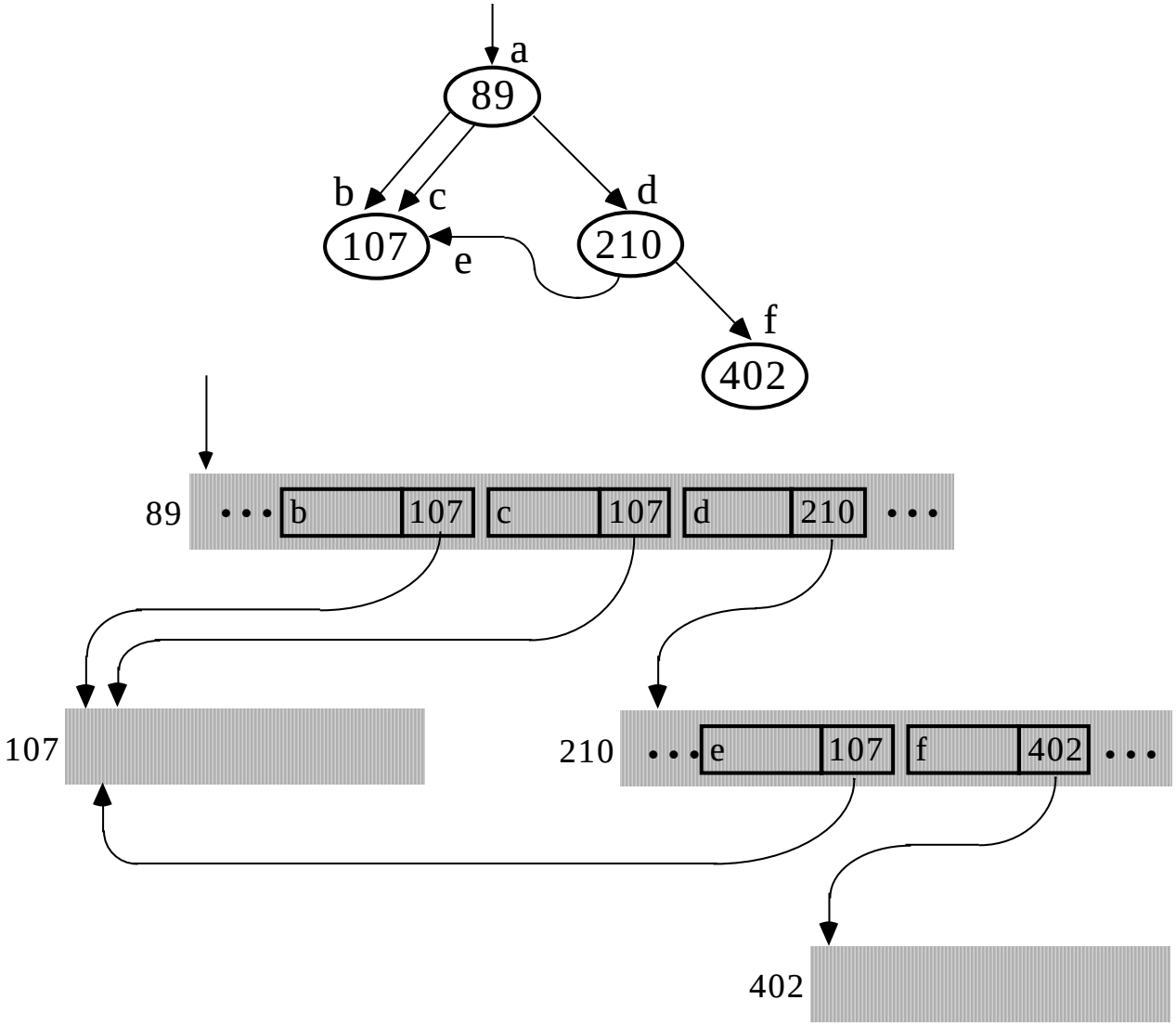
"•" la directory stessa

"••" la directory padre

FILENAME SINONIMI

Un file può avere più filename (ma sempre un solo i-number)

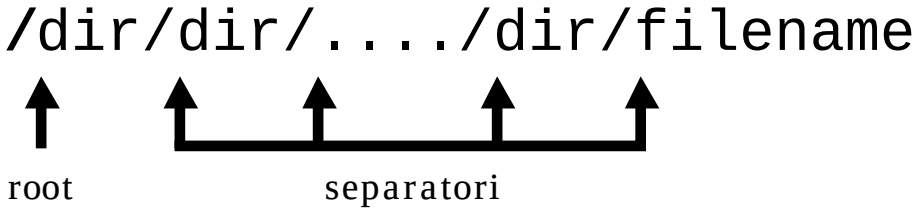
Esempio:



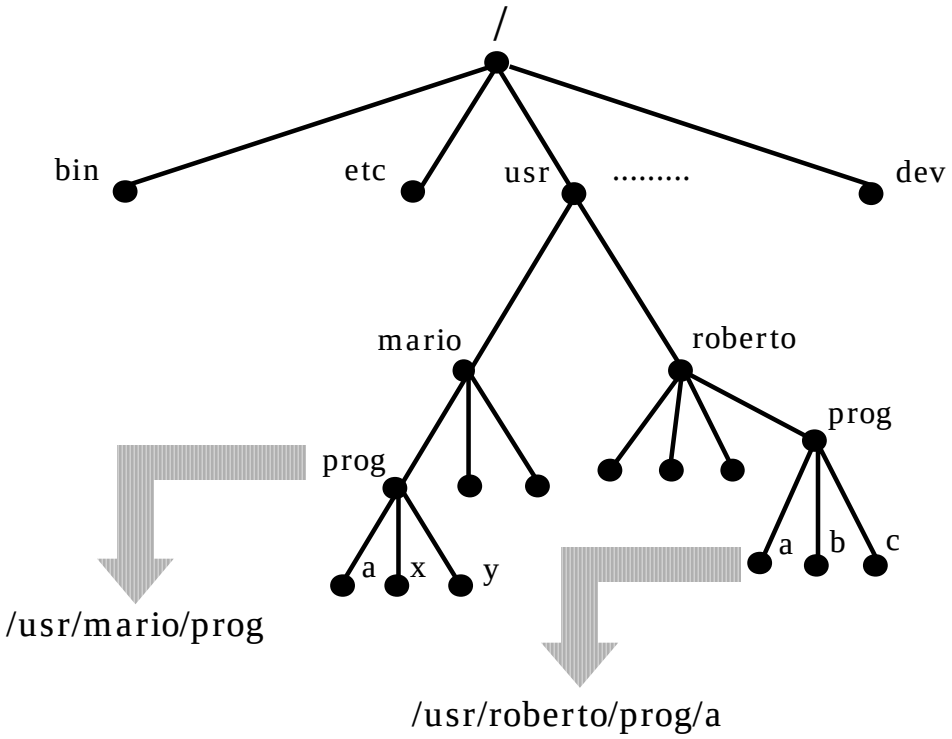
Il file 107 **ha 3 links**

PATHNAMES

Ogni file viene identificato univocamente specificando il suo **pathname**, che individua il cammino dalla root-directory al file



Esempio:



TIPICHE DIRECTORIES DI SISTEMA

`/bin`	comandi eseguibili
`/dev`	files speciali (I/O devices)

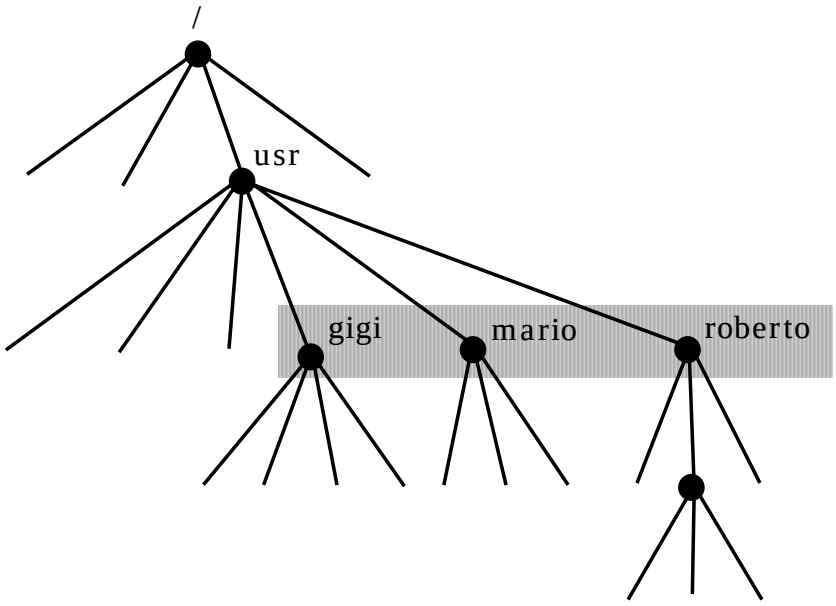
`/etc`	files per l'amministrazione del sistema, ad esempio:
	`/etc/passwd`
	`/etc/termcap`

`/lib`	librerie di programmi
`/lost+found`	
`/tmp`	area temporanea
`/usr`	home directories

N.B. La struttura varia da versione a versione

HOME DIRECTORIES

- Ad ogni utente viene assegnata, ad opera del system administrator, una directory di sua proprietà (**home directory**) che ha come nome lo username del suo proprietario
- Ad essa, l'utente potrà appendere tutti i files (o subdirectories) che desidera
- Spesso (ma non sempre) le home directories sono sotto `/usr`:



- Per denotare la propria home directory si può usare l'abbreviazione "**~**"

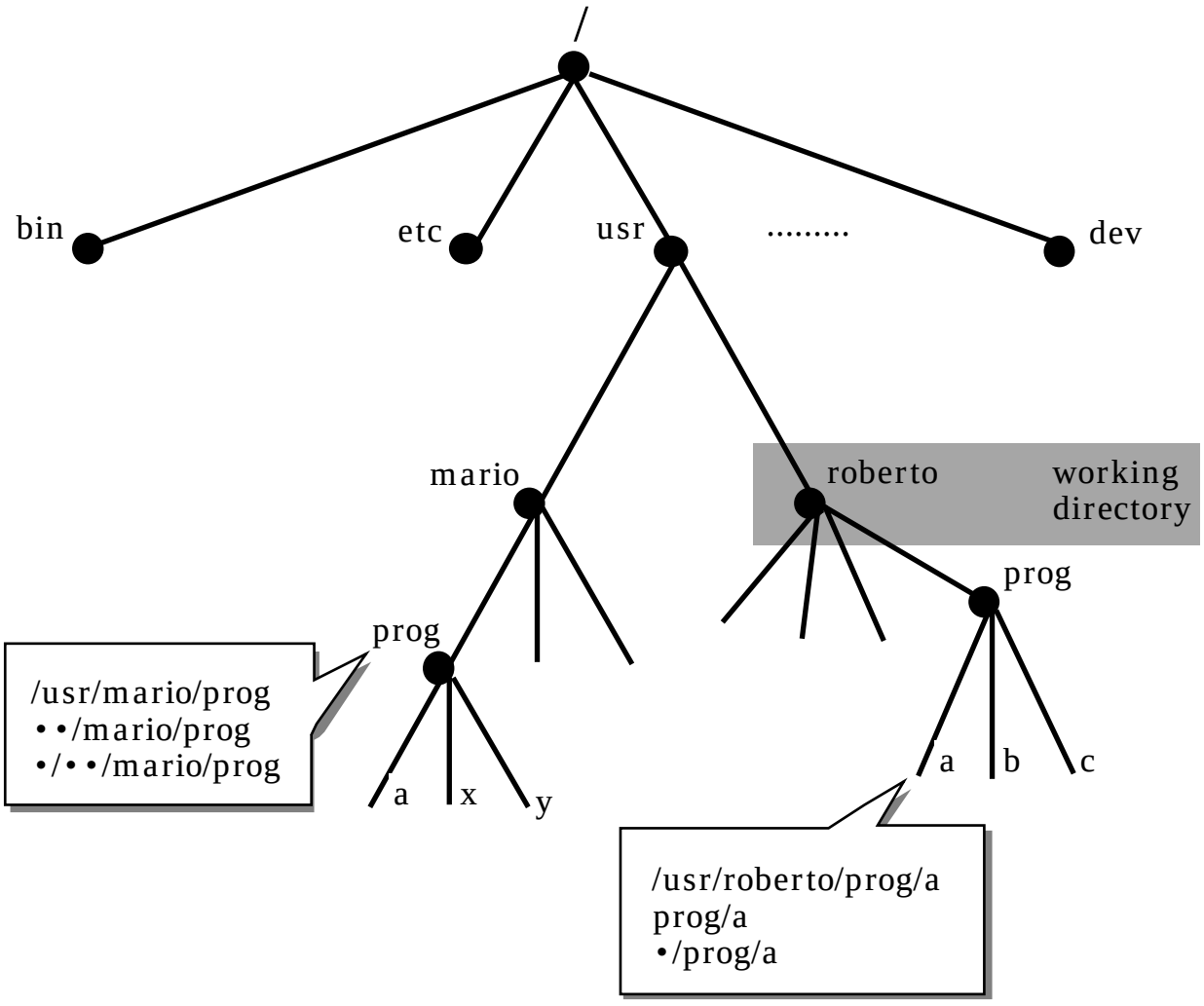
CURRENT WORKING DIRECTORY

- Ogni utente opera, ad ogni istante, su una directory corrente, o **working directory**
- Subito dopo il login, la working directory è la home directory dell'utente
- L'utente può cambiare la working directory con apposito comando (cd)

PATHNAMES RELATIVI

Ogni file può essere identificato univocamente specificando solamente il suo **pathname relativo alla working directory**

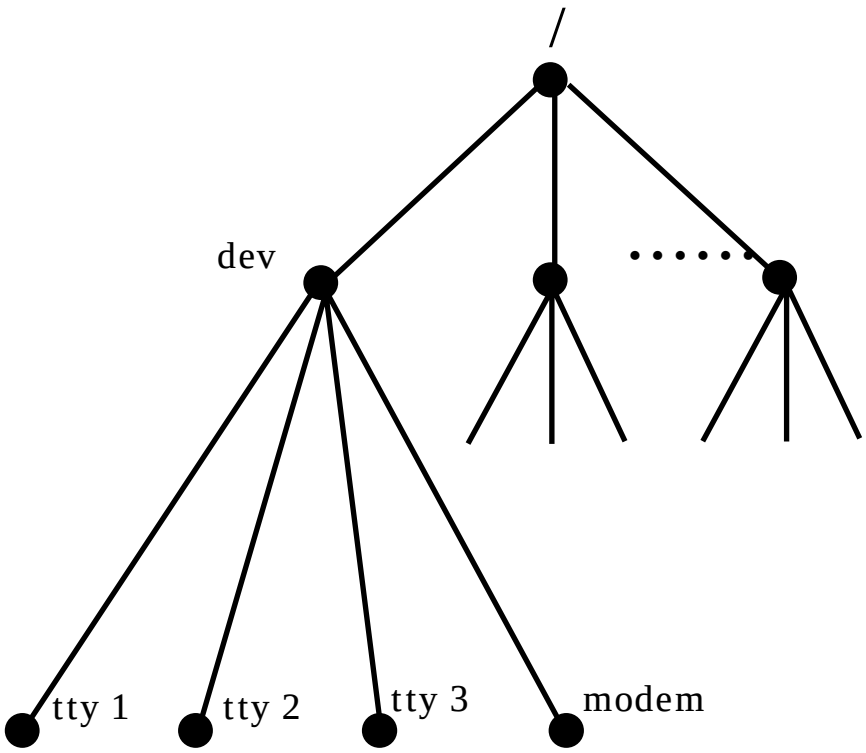
Esempio:



FILES SPECIALI

- Ogni device di I/O viene visto, a tutti gli effetti, come un file (**file speciale**)
- Richieste di lettura/scrittura da/a files speciali causano operazioni di input/output dai/ai devices associati

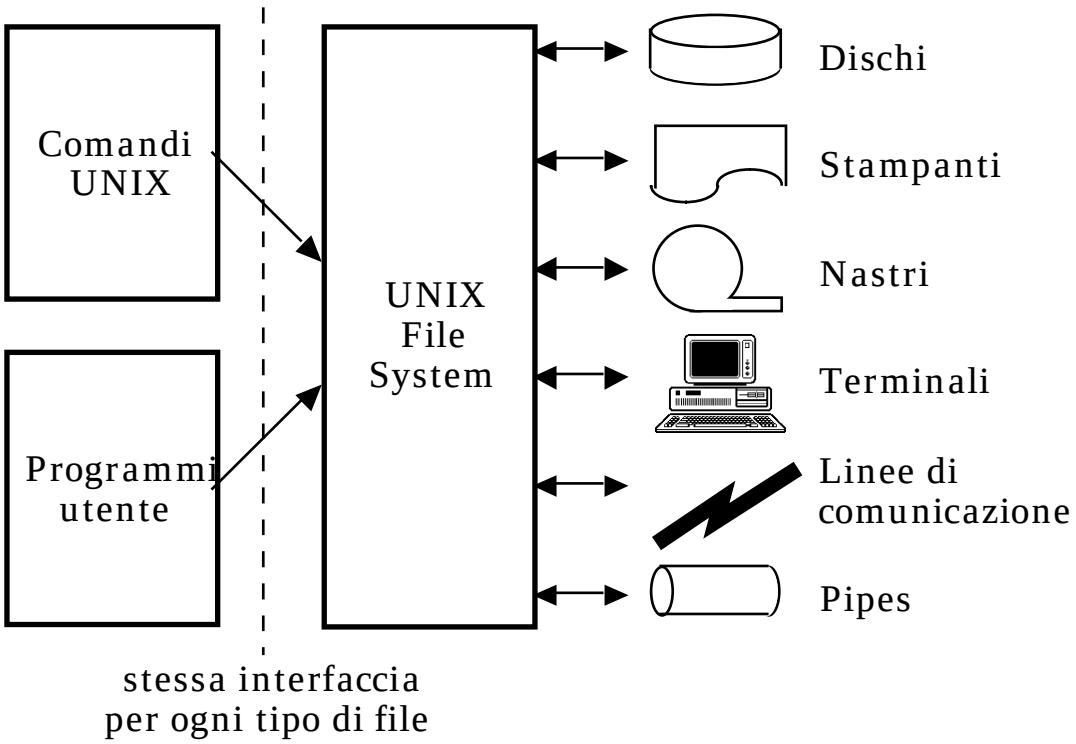
Esempio:



FILES SPECIALI: VANTAGGI

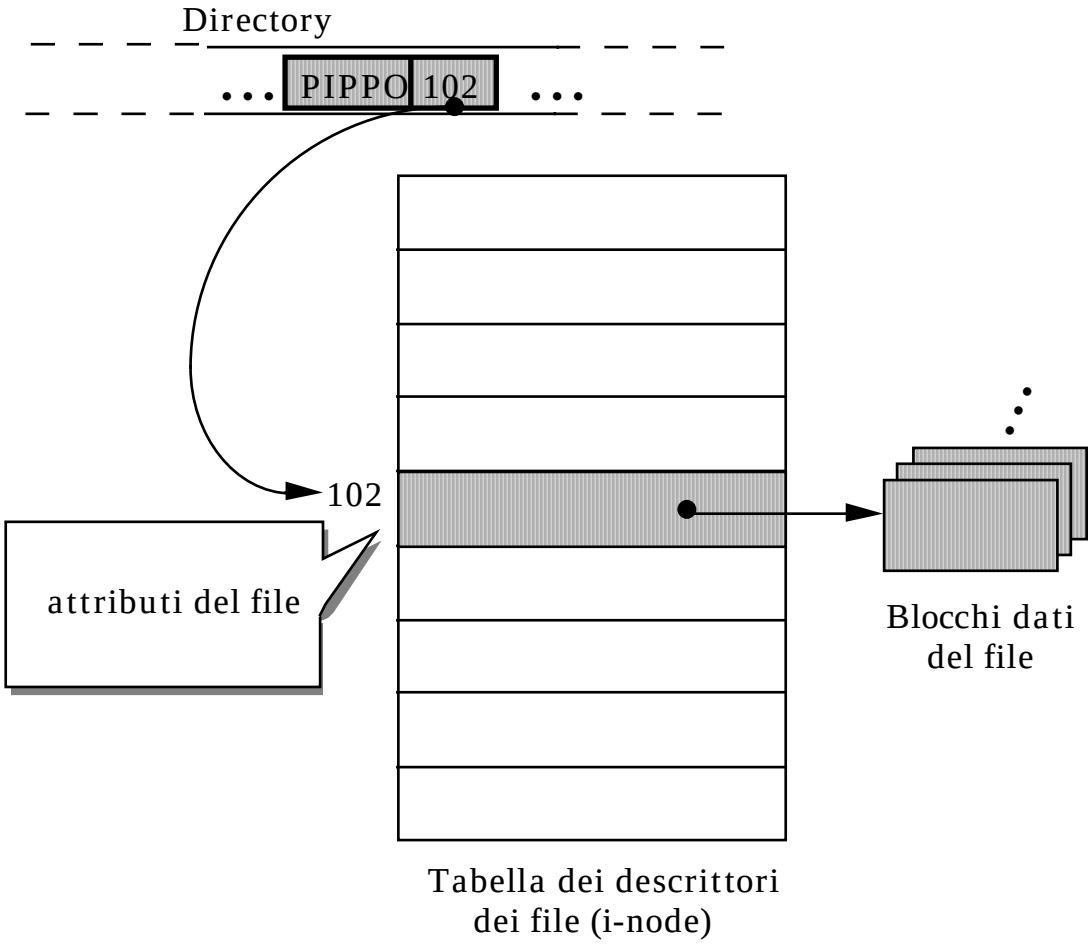
Trattamento uniforme di files e devices

In Unix i programmi non sanno se operano su un file o su un device



File & device independence

IMPLEMENTAZIONE DEI FILE (CENNI)



ATTRIBUTI DI UN FILE UNIX

Per ogni file (ordinario, directory, speciale) Unix mantiene le seguenti informazioni nel descrittore del file:

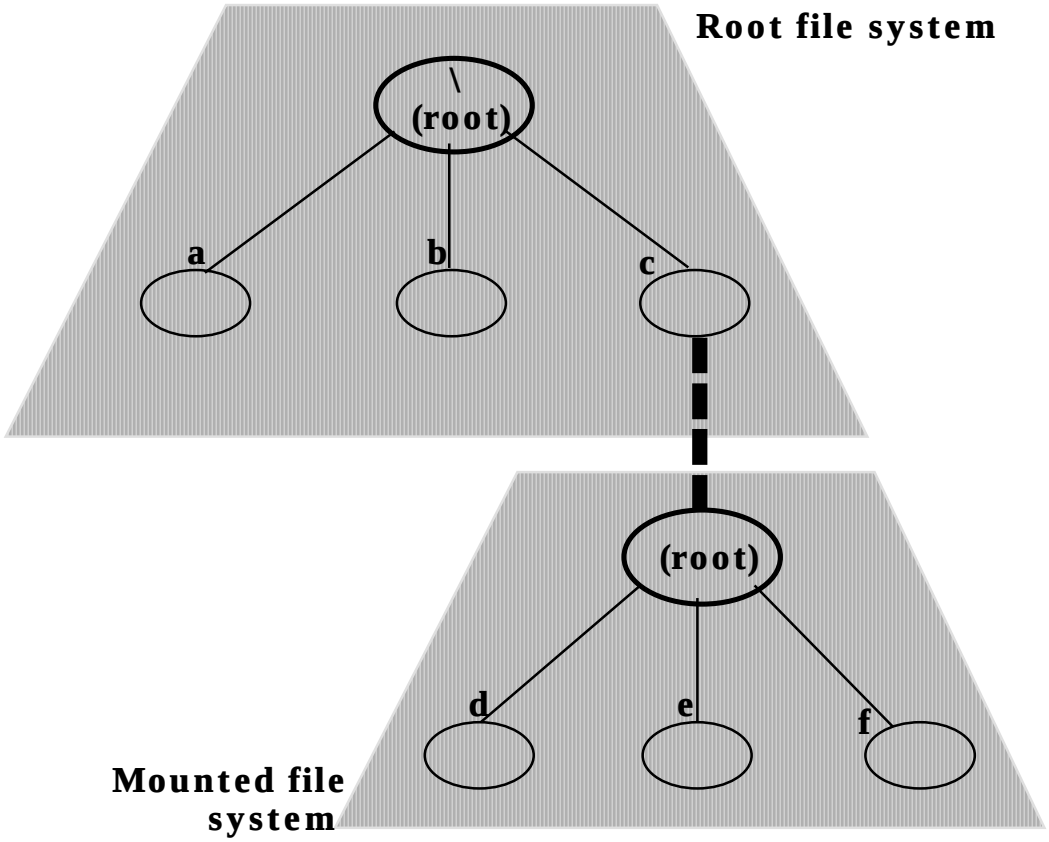
Tipo	ordinario, directory, speciale?
Posizione	dove si trova?
Dimensione	quanto è grande?
Numero di links	quanti nomi ha?
Proprietario	chi lo possiede?
Permessi	chi può usarlo e come?
Creazione	quando è stato creato?
Modifica	quando è stato modificato più di recente?
Accesso	quando è stato l'accesso più recente?

FILE SYSTEM "MONTABILE"

Un file system Unix è sempre **unico**, ma può avere parti residenti su device rimuovibili (es. dischetti)

Queste parti devono essere:

- "montate" prima di potervi accedere (`mount`)
- "smontate" prima di rimuovere il supporto (`umount`)



La stessa tecnica si usa per suddividere il file system fra diversi device, anche se non rimuovibili

4. COMANDI DI GESTIONE DELLE DIRECTORIES

GESTIONE DIRECTORIES: ALCUNI COMANDI

pwd	**p**rint **w**orking **d**irectory
cd	**c**hange **d**irectory
ls	**l**i**s**t directory
du	**d**isk **u**sage
mkdir	**m**a**k**e **dir**ectory
rmdir	r**m**ove **dir**ectory
ln	**l**i**n**k

…

IL COMANDO pwd

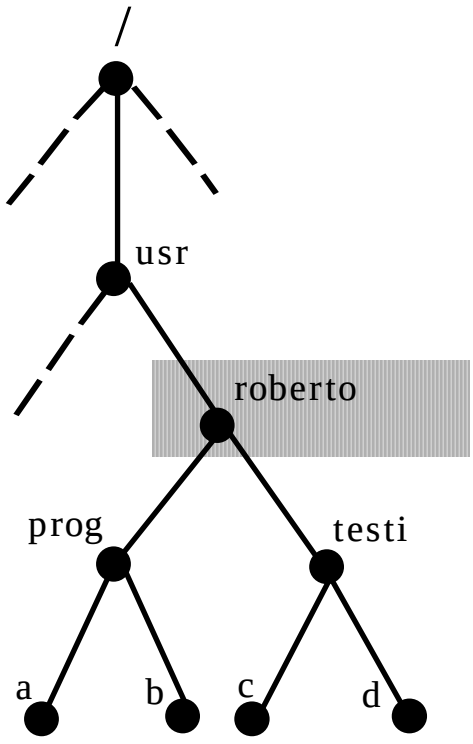
pwd

"print working directory"

- stampa il pathname della directory corrente

Esempio:

```
% pwd
/usr/roberto
%
```



IL COMANDO cd

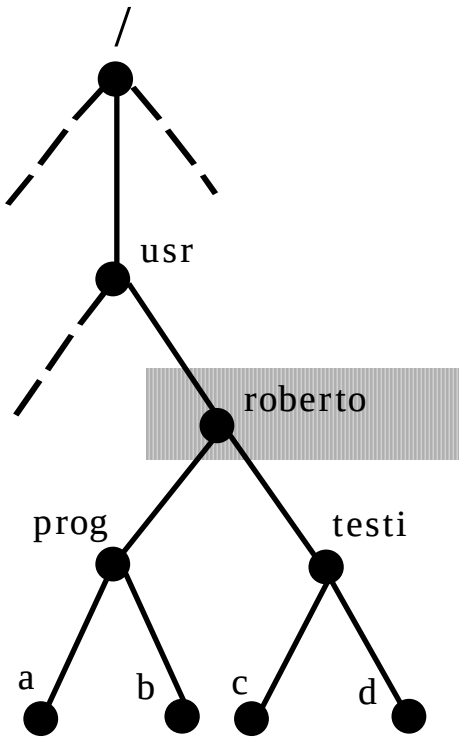
cd [*directory*]

"change directory"

- la directory specificata diviene la working directory
- se nessuna directory è specificata, si "ritorna" alla home directory

Esempio:

```
%cd /usr
%pwd
/usr
%cd
%pwd
/usr/roberto
%
```



IL COMANDO `ls`

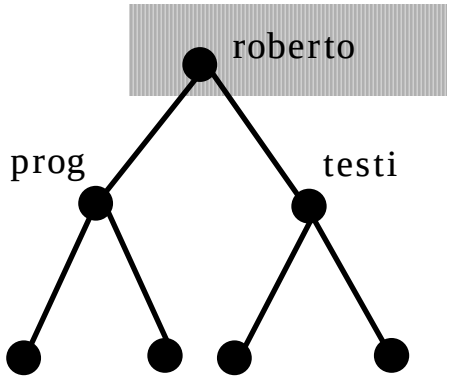
`ls` [*`options`*][*`directory...`*]

"list directory"

- lista (in ordine alfabetico) il contenuto della o delle directories indicate
- se nessuna directory è indicata, lista il contenuto della working directory
- possiede numerose opzioni

Esempio:

```
%ls
prog testi
%
```



`ls` - ALCUNE OPZIONI

`-s`	fornisce la dimensione in blocchi (**s**ize)
`-t`	lista nell'ordine di modifica (prima il file modificato per ultimo) (**t**ime)
`-1`	un nome per ogni riga
`-F`	aggiunge / al nome delle directory e * al nome dei files eseguibili
`-R`	si chiama ricorsivamente su ogni sottodirectory
`-i`	fornisce l'i-number del file

•••• e molte altre

ls - ESEMPI

```
% ls
dir1   file1
% ls -s
total 4       2 dir1    2 file1
% ls -t
file1  dir1
% ls -1
dir1
file1
% ls -F
dir1/  file1
% ls -R
dir1   file1

./dir1:
file1     file2     file3     file4
% ls -i
199742 dir1  51204 file1
%
```

FILES NASCOSTI ("DOTFILES")

I files il cui nome inizia con "." vengono listati solo specificando l'opzione `-a` ("all")

Esempio:

```
% ls -a
.   .cshrc    .mailrc          dir1
..     .login    .sh_history      file1
%
```

IL COMANDO du

du [*options*][*name...*]

"disk usage"

- stampa il numero di blocchi contenuti in tutti i files e (ricorsivamente) directories specificate
- se *name* non è specificato, si intende la directory corrente
- `-s`: solo il totale

Esempio:

```
% du
2       ./dir1
2       ./dir2
14      .
% du -s ..
198812  ..
%
```

IL COMANDO `mkdir`

`mkdir` *`directory`*`...`

"make directory"

- crea la/le directory indicata/e

Esempio:

```
% mkdir dir1 dir2
% ls
dir1  dir2
%
```

IL COMANDO `rmdir`

`rmdir` *`directory`*…

"remove directory"

- rimuove la/le directory indicata/e
- le directory devono essere vuote

Esempio:

```
% rmdir dir
rmdir: dir: Directory not empty
% ls dir
a
% rm dir/a
% rmdir dir
%
```

IL COMANDO `ln`

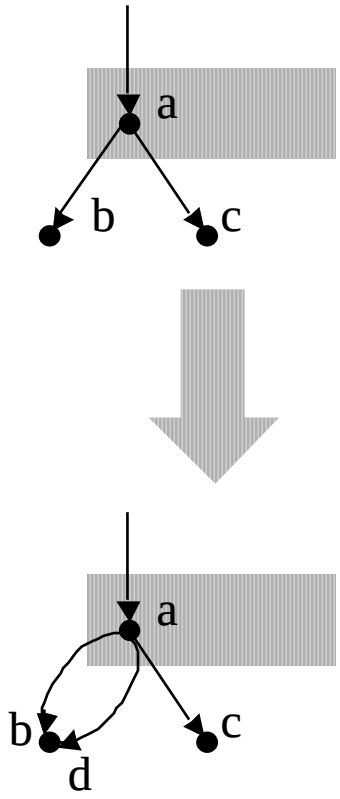
`ln` *`name1 name2`*

"link"

- associa il nuovo nome (link) *`name2`* al file (esistente) *`name1`*, che non può essere una directory

Esempio:

```
% ln b d
%
```



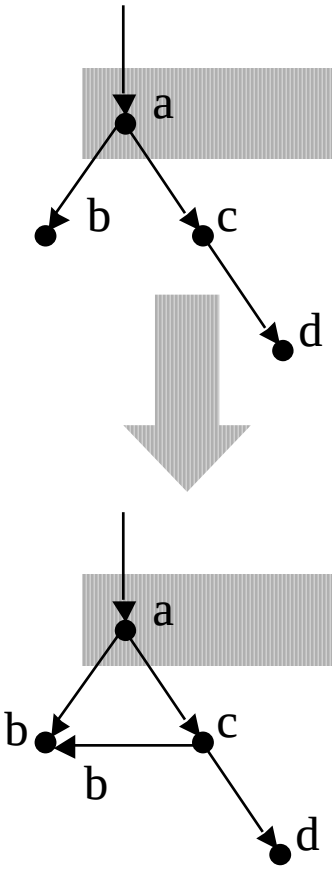
IL COMANDO `ln` (SEGUE)

`ln` *`name1 name2`*

- se *`name2`* è una directory, il nuovo nome è *`name2/name1`*

Esempio:

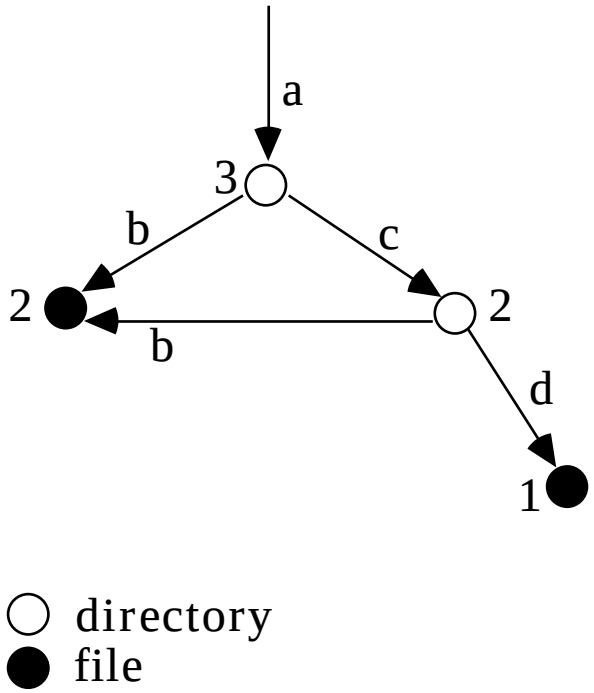
```
% ln b c
%
```



NUMERO DI LINKS

È uno degli attributi dei files gestiti dal sistema

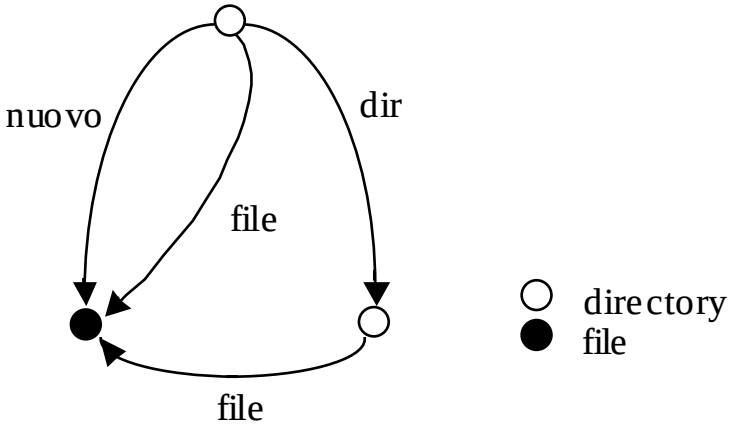
Esempio:



Per vedere il numero di links: `ls -l`

ESEMPI

```
% mkdir dir
% touch file
% ls -l
total 2
drwxr-sr-x   2 roberto  usrmail   512   Mar 11 19:40 dir
-rw-r--r--   1 roberto  usrmail     0   Mar 11 19:40
file
% ln file nuovo
% ls -i
199742 dir   51204 file   51204 nuovo
% ls -l
total 2
drwxr-sr-x   2 roberto  usrmail    512 Mar 11 19:40 dir
-rw-r--r--   2 roberto  usrmail      0 Mar 11 19:40 file
-rw-r--r--   2 roberto  usrmail      0 Mar 11 19:40
nuovo
% ln file dir
% ls -l dir
total 0
-rw-r--r--   3 roberto  usrmail      0 Mar 11 19:40 file
% ln dir nuovissimo
ln: dir is a directory
%
```



NOTE

- Tutti i link allo stesso file hanno identico status e caratteristiche
- Non è possibile distinguere la entry originaria dai nuovi link
- I link di questo tipo non possono essere fatti con file che stanno su file system diversi

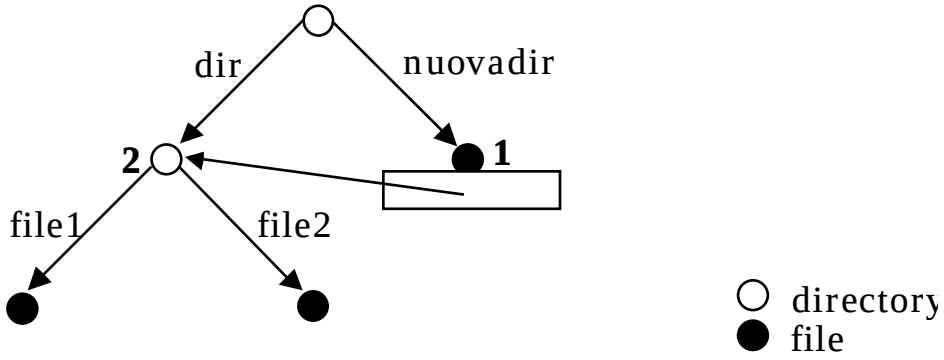
SYMBOLIC LINKS

`ln -s` *`name1 name2`*

- Permette di creare links a directories e links fra files o directories che stanno su file systems diversi
- Viene creato un file `name2` che contiene il link simbolico (pathname di `name1`)

Esempio:

```
% ls
dir
% ls dir
file1   file2
% ln -s dir nuovadir
% ls
dir     nuovadir
% ls nuovadir
file1   file2
% ls -l
total 4
drwxr-sr-x  2 roberto  usrmail 512 Mar 11 19:24
dir
lrwxrwxrwx  1 roberto  usrmail   3 Mar 11 19:24
nuovadir -> dir
%
```



5. COMANDI
DI GESTIONE DEI FILES

ALCUNI COMANDI PER GESTIRE I FILES

mv : **m**o**v**e file

cp : **c**o**p**y file

rm : **r**e**m**ove file

touch : modifica data e ora dell'ultimo accesso/modifica di un file

find : cerca file con specificati attributi

...

IL COMANDO mv

mv [*options*] *name...target*

"move"

- muove il file o directory *name* sotto la directory *target*

- se *name* e *target* non sono directories, il contenuto di *target* viene sostituito dal contenuto di *name*

mv: ESEMPI

- **Se *target* è una directory:**

```
% ls
file1       file2       targetdir
% mv file1 file2 targetdir
% ls
targetdir
% ls targetdir
file1       file2
% mv targetdir/file1 targetdir/file2 .
% ls
file1       file2       targetdir
%
```

- **Se *target* è un file:**

```
% ls
file1       file2       file3       targetfile
% mv file1 targetfile
% ls
file2       file3       targetfile
% mv file2 file3 targetfile
mv: Target targetfile must be a directory
Usage: mv [-f] [-i] f1 f2
       mv [-f] [-i] f1 ... fn d1
       mv [-f] [-i] d1 d2
%
```

- **Se *target* non esiste:**

```
% ls
file1       file2
% mv file1 file2 target
mv: target not found
% mv file1 target
% cat target
contenuto di file1
%
```

IL COMANDO cp

cp [*options*][*name...*] *target*

"copy"

come **mv**, ma *name* viene copiato

Esempi:

```
% ls
file1  file2  targetdir
% cp file1 file2 targetdir
% ls . targetdir
.:
file1  file2  targetdir

targetdir:
file1  file2
%
```

```
% ls
file1  targetfile
% cp file1 targetfile
% ls
file1  targetfile
%
```

IL COMANDO `rm`

`rm` [**`-r`**]*name…*

"remove"

- rimuove i files indicati
- se un file indicato è una directory: messaggio di errore, a meno che non sia specificata l'opzione `-r`
- … nel qual caso, rimuove ricorsivamente il contenuto della direttrice
- <altri parametri>

IL COMANDO `touch`

touch [*options*][*time*] *filename...*

- aggiorna la data e l'ora dell'ultimo accesso (opzione `-a`) o dell'ultima modifica (opzione `-m`) di *filename* (default: `-am`)
- se *time* non è specificato, usa la data e l'ora corrente
- se il file non esiste, lo crea

Esempio:

```
% touch 01281738 file1
% ls -l
total 0
----r--r--  1 roberto  usrmail  0  Jan 28 17:38
file1
%
```

FILENAME GENERATION

La shell fornisce un meccanismo che permette di specificare una lista di nomi di files mediante una singola espressione sintetica

Esempio:

```
% ls
gianni   giorgio  laura
mario
% rm g*
% ls
laura  mario
% rm *
%
```

Rimuove tutti i files presenti nella directory corrente, ad eccezione di quelli che iniziano con il "."

(.profile, .cshsc, .login, ...)

FILENAME GENERATION: METACARATTERI

? un carattere qualsiasi

Es. `rm file?`

***** una stringa di Ø o più caratteri qualsiasi

Es. `rm file*`

[...] uno qualsiasi dei (singoli) caratteri racchiusi fra [ ] (alternativa)

Es. `rm file[123]`

`rm file[a-z]` (subrange)

N.B.

- il "." a inizio nome file deve essere esplicitamente specificato
- per sopprimere il significato speciale dei metacaratteri, occorre anteporre \

IL COMANDO echo

echo [*argomenti*]

Visualizza gli argomenti in ordine, separati da singoli blank

Esempio:

```
% echo uno due tre
uno due tre
%
```

FILENAME GENERATION - ESEMPI

```
% echo /*/tty*
/bin/tty /bin/ttyhstmgr /dev/tty /dev/ttya /dev/ttyb/
dev/ttyp0/dev/ttyp1 /dev/ttyp2 /dev/ttyp3 /dev/ttyp4
/
dev/ttyp5 /dev/ttyp6 /dev/ttyp7 /dev/ttyp8 /dev/ttyp9
/
dev/ttypa /dev/ttypb /dev/ttypc /dev/ttypd /dev/ttype
/
dev/ttypf /dev/ttyq0/dev/ttyq1 /dev/ttyq2 /dev/ttyq3
/
dev/ttyq4 /dev/ttyq5 /dev/ttyq6 /dev/ttyq7 /dev/ttyq8
/
dev/ttyq9 /dev/ttyqa /dev/ttyqb /dev/ttyqc /dev/ttyqd
/
dev/ttyqe /dev/ttyqf /dev/ttyr0 /dev/ttyr1 /dev/ttyr2
/
dev/ttyr3 /dev/ttyr4 /dev/ttyr5 /dev/ttyr6 /dev/ttyr7
/
dev/ttyr8 /dev/ttyr9 /dev/ttyra /dev/ttyrb /dev/ttyrc
/
dev/ttyrd /dev/ttyre /dev/ttyrf /etc/ttydefs
/etc/ttysrch
%
```

IL COMANDO FIND

find *pathname…* [*expression*]

discende ricorsivamente le directories specificate (*pathname…*), cercando tutti i files che rendono vera la *expression* booleana.

Molto flessibile:

- ricerca files di specificati attributi (filename, tipo, permessi, proprietario, gruppo, numero di links, dimensione, tempo dell'ultima modifica/accesso …)
- and, or, not di attributi
- può eseguire automaticamente, o previa conferma, uno o più comandi sui files individuati

ESEMPIO

Rimuovi tutti i files nella working directory (o più sotto) di nome `a.out` il cui ultimo accesso è anteriore a 7 giorni:

```
% find . -name a.out -atime +7 -exec rm {} \;
%
```

Come si interpreta:

`.`	pathname della directory da cui effettuare la ricerca
`-name a.out`	nome del file da cercare
`-atime +7`	access time superiore a 7 giorni
`-exec rm {} \;`	esegui il comando `rm` sul file corrente (\; termina il comando da eseguire)

6. SICUREZZA

IDENTIFICAZIONE DEGLI UTENTI

Ogni utente viene identificato da uno **user name** di al più 8 caratteri, assegnato dall'amministratore del sistema

Ad esso corrisponde biunivocamente uno **user-id** numerico, assegnato dal sistema

User name e user-id sono **pubblici**

Esempio:

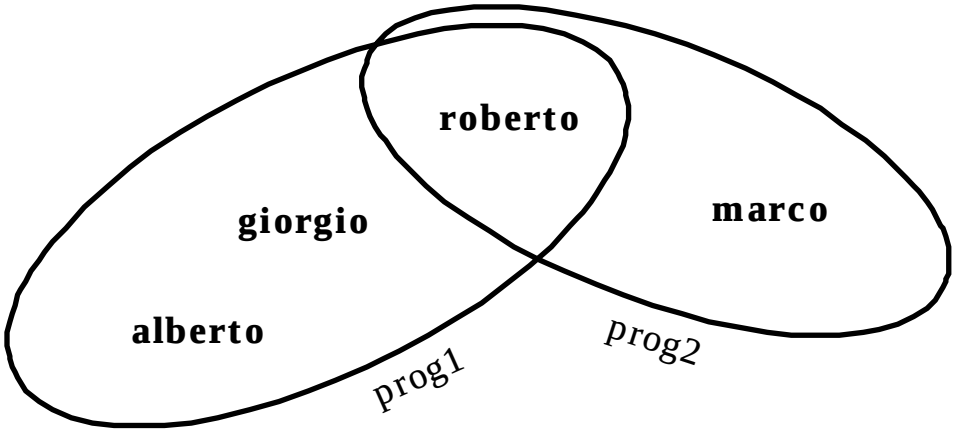
```
login: roberto
```

GRUPPI

Ogni utente può far parte di uno o più **gruppi**, definiti dall'amministratore del sistema

Ogni gruppo è identificato da un **group name** di al più 8 caratteri, associato biunivocamente a un **group-id** numerico

Esempio:



Ad ogni istante, solo la appartenenza a un unico gruppo è **attiva**

Il file pubblico `/etc/group` contiene la lista dei gruppi e delle membership.

ESEMPIO

```
% cat /etc/group
root::0:root
other::1:
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
adm::4:root,adm,daemon
uucp::5:root,uucp
mail::6:root
tty::7:root,tty,adm
lp::8:root,lp,adm
nuucp::9:root,nuucp
staff::10:
users::20:
user::11:
usrmail::100:
daemon::12:root,daemon
nobody::60001:
noaccess::60002:
news::30:
%
```

IL COMANDO `id`

`id [-a]`

"Identifier"

- stampa user-id, user name, group-id, group name dell'utente (gruppo attivo)
- con l'opzione `-a`, stampa tutti i gruppi di appartenenza

Esempio:

```
% id
uid=207(roberto) gid=100(usrmail)
%
```

IL SUPERUSER

È un utente privilegiato (user name = `root`), al quale sono riservati i compiti di amministrazione del sistema

Esempi:

Creazione utenti (`useradd, userdel, usermod`)

Modifica gruppi (`addgrp, delgrp`)

IL COMANDO `newgrp`

`newgrp` *`groupname`*

"new group"

- associa l'utente al gruppo specificato
- se il gruppo ha una password, e se l'utente non è membro di quel gruppo, il comando richiede la password

SICUREZZA

I meccanismi di sicurezza che Unix possiede per la protezione da accessi indesiderati sono di tre tipi:

- **Accesso al sistema**

  L'uso del sistema è consentito soltanto agli utenti autorizzati, mediante uno schema di login/logout

- **Accesso ai files**

  L'accesso ai files è consentito soltanto agli utenti autorizzati, mediante uno schema di permessi di accesso

- **Accesso ai processi**

  L'accesso ai processi (ad esempio per terminarli) è consentito solo agli utenti autorizzati

N.B. Al superuser non è inibito alcun accesso

LOGIN

```
login: roberto
Password:
Last login: Sun Mar 19 15:10:29 from
xxxx
Sun Microsystems Inc.   xxxxxxxx
%
```

Le password sono registrate, in forma crittografata, nel file pubblico `/etc/passwd`, assieme a user name, user-id e altre informazioni sull'ambiente

Esempio:

```
% cat /etc/passwd
.....
roberto:x:207:100:Roberto
Polillo:/usermail/roberto:/bin/csh
.......
%
```

DEFINIZIONE E MODIFICA PASSWORD

Ogni utente può definire (e in seguito modificare) una propria password, con apposito comando:

passwd

Esempio:

```
%passwd
passwd:  Changing password for roberto
Old password:
New password:
Re-enter new  password:
%
```

ESEMPIO

```
% passwd
passwd:  Changing password for roberto
Old password:
New password:
Passwords must differ by at least 3 positions
New password:
Password is too short - must be at least 6
characters.
New password:
Password must contain at least two alphabetic
characters and
at least one numeric or special character.
Too many failures - try later.
%
```

COME NON SCEGLIERE LA PASSWORD

Non scegliete:

- il vostro user-id
- il vostro nome o cognome (o una combinazione di entrambi)
- il vostro numero di telefono (neanche al contrario!)
- la vostra data di nascita (o un suo anagramma)
- una parte del vostro indirizzo
- il nome di battesimo (o la data di nascita) della mamma
- il vostro numero di matricola
- ...

...qualcuno ci ha già pensato!

PER CONTROLLARE ...

%last roberto

Informazioni sull'ultimo login di roberto

%last

Informazioni sull'ultimo login di tutti

Esempio:

```
%last roberto
roberto  pts/1   194.20.15.99  Sun Mar  5 16:24  still
logged in
roberto  pts/1   194.20.15.99  Sun Mar  5 16:19 - 16:23
(00:03)

wtmp begins Fri Sep  9 16:12
%
```

IL COMANDO su

su [*username*]

"substitute user"

- permette di diventare l'utente *username* senza chiudere la sessione di lavoro
- se *username* manca, si assume `root`

Esempio:

```
% su
Password:
```

PROTEZIONE DI UN PROCESSO: GENERALITÀ

A ciascun processo sono associati alcuni attributi:

- **Real User ID** e **Effective User ID**
  Normalmente sono identici, e coincidenti con lo User ID dell'utente che lo ha lanciato

- **Real Group ID** e **Effective Group ID**
  Normalmente sono identici, e coincidenti con il Group ID dell'utente che lo ha lanciato

Questi attributi vengono controllati prima di permettere ad un utente di dato User Id e Group Id di effettuare operazioni critiche sui processi

SET USER ID

Tra gli attributi di ogni file esistono due bit detti "set user Id" e "set group Id"

Se il bit di "set user Id" é settato in un file esegui-bile, il processo che lo eseguirà avrà:

- il **Real User Id** uguale allo User Id dell'utente che ha lanciato il processo, e
- l'**Effective User Id** uguale allo User Id dell'owner del file eseguibile

Se il bit di "set group Id" é settato in un file esegui-bile, il processo che lo eseguirà avrà:

- il **Real Group Id** uguale al Group Id dell'utente che ha lanciato il processo, e
- l'**Effective Group Id** uguale al Group Id dell'owner del file eseguibile

Tutto questo serve per realizzare uno speciale meccanismo di protezione ai files (vedi in seguito)

PROTEZIONE DI UN FILE: GENERALITÀ

A ciascun file (normale, speciale, directory) sono associati alcuni attributi:

- **Proprietario (owner)**
  È l'utente che ha creato il file

- **Gruppo**
  È il gruppo a cui il proprietario appartiene

- **Permessi (permissions)**
  Il tipo di operazioni che il proprietario, i membri del suo gruppo o gli altri utenti possono compiere sul file

Proprietario, gruppo e permessi iniziali sono assegnati dal sistema al file al momento della sua creazione.

Il proprietario potrà successivamente modificare tali attributi con appositi comandi (*chown*, *chgrp*, *chmod*)

IL COMANDO chown

chown *newuserid file...*

"ch<u>ange own</u>er"

- *newuserid* diventa il nuovo proprietario dei *file...*
- il comando può essere eseguito solo dal proprietario "cedente" (o dal superuser)

IL COMANDO `chgrp`

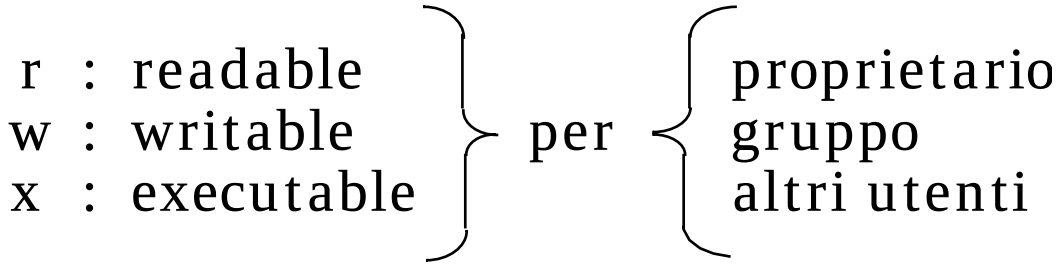
`chgrp` *`newgroupid file...`*

"change group"

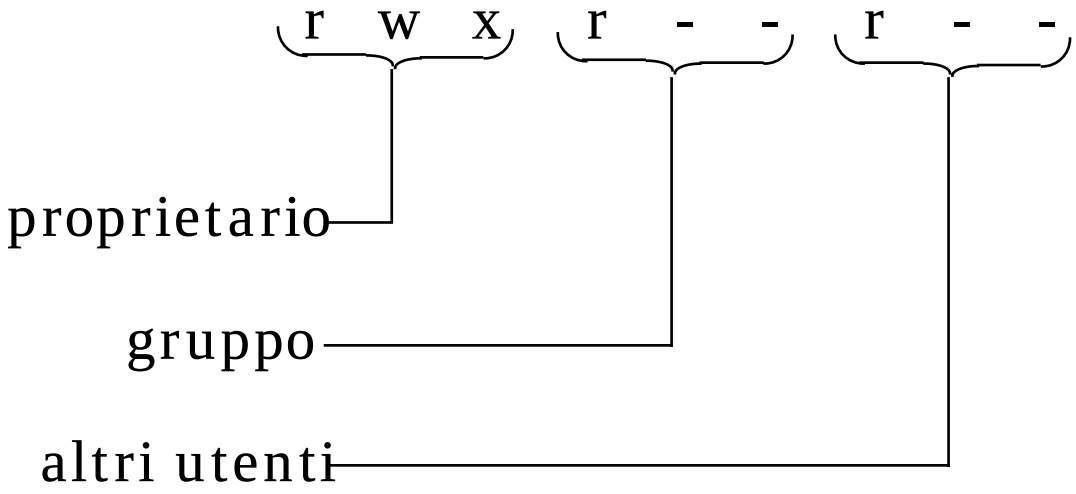
- *`newugroupid`* diventa il nuovo gruppo dei *`file...`*
- il comando può essere eseguito solo dal proprietario (o dal superuser)

PERMESSI

Ad un file possono essere attribuiti i seguenti permessi:



Esempio:



In binario: 1 1 1 1 0 0 1 0 0

In ottale: 7 4 4

PERMESSI: SIGNIFICATO

Per i file ordinari significano:

r: leggere il contenuto
w: modificare il contenuto
x: eseguire il file (ha senso solo se il file contiene un programma)

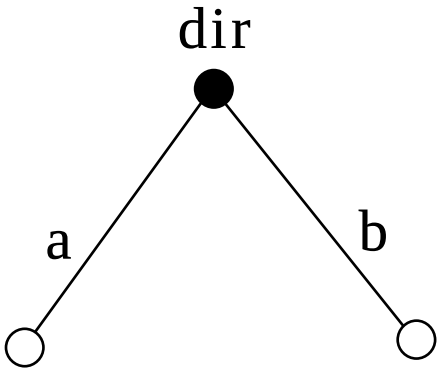
Per le directories significano:

r: leggere la directory (es.: `ls`, con x abilitato)
w: modificare la directory, cioè aggiungere, rinominare, rimuovere files (con x abilitato)
x: accesso (scansione) della directory (per leggere, modificare, eseguire un file in essa contenuto)

Per i device significano:

r: leggere dal device (input)
w: scrivere sul device (output)
x: non significativo

PERMESSI: ESEMPIO

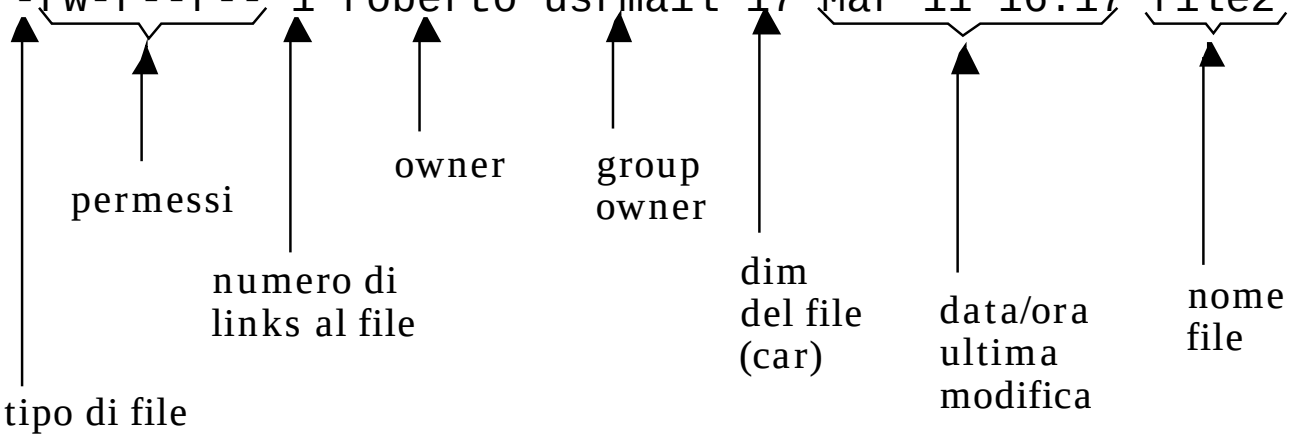


Per rimuovere a deve essere

```
a:   w
dir: w,x
```

PER VEDERE I PERMESSI

```
% ls -l
total 4
-rw-r--r-- 1 roberto usrmail 17 Mar 11 16:16 file1
-rw-r--r-- 1 roberto usrmail 17 Mar 11 16:17 file2
```



tipo di file
- file ordinario
d directory
b device a blocchi
c device a caratteri
s link simbolico
...

PERMESSI INIZIALI

Alla creazione di un file, Unix assegna i seguenti permessi:

- Per i files ordinari non eseguibili:

  `rw-rw-rw` ⟶ `110 110 110` ⟶ `666`

- Per i files ordinari eseguibili e per directories:

  `rwx rwx rwx` ⟶ `111 111 111` ⟶ `777`

L'utente può "mascherare" alcuni di questi permessi mediante il comando `umask`, che di solito è eseguito nel file dei comandi di inizializza-zione eseguito dalla shell

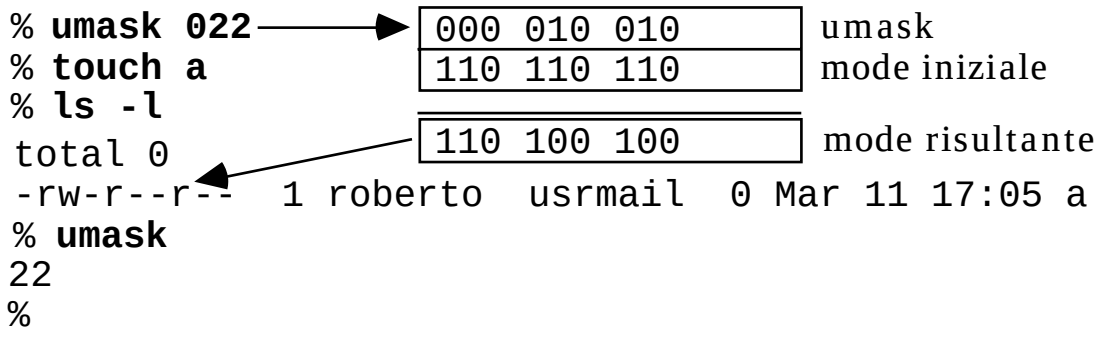
IL COMANDO umask

umask [*mask*]

"user mask"

- "maschera" dal "mode" iniziale i permessi specificati da *mask* (XOR)
- se *mask* non è specificato, stampa il valore della maschera corrente

Esempio:



IL COMANDO chmod

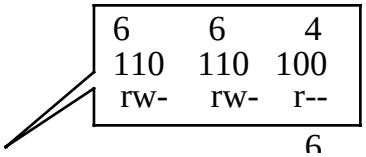
chmod *permissions filename...*

"change mode"

- attribuisce le *permissions* a *filename* (solo da parte del proprietario del file!)
- *permissions* può essere espresso in forma ottale o simbolica

chmod: ESEMPI

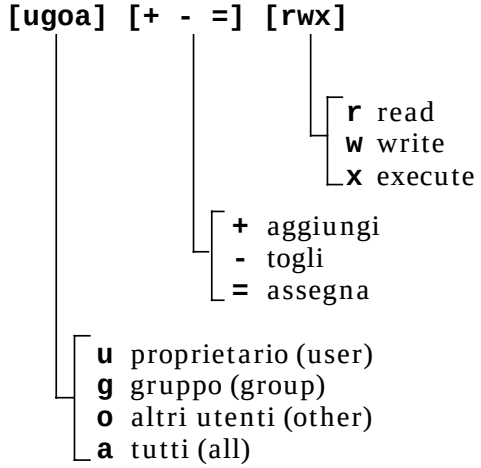
Permessi in forma ottale:



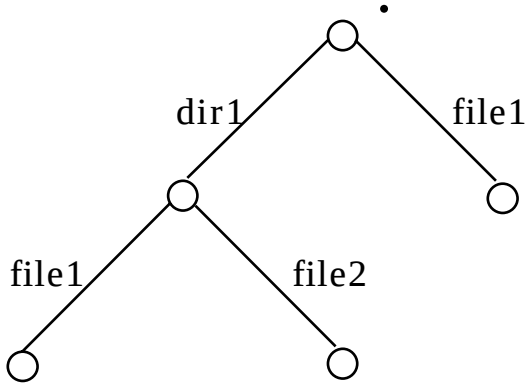

```
% chmod 664 file1 file2
% ls -l
total 4
-rw-rw-r--  1 roberto  usrmail  35 Mar 11 16:34 file1
-rw-rw-r--  1 roberto  usrmail  17 Mar 11 16:17 file2
%
```

Permessi in forma simbolica:

```
% ls -l
total 4
-rw-rw-r--  1 roberto  usrmail  35 Mar 11 16:34 file1
-rw-rw-r--  1 roberto  usrmail  17 Mar 11 16:17 file2
% chmod ugo+x  file1
% chmod o=rwx  file2
% ls -l
total 4p
-rwxrwxr-x  1 roberto  usrmail  17 Mar 11 16:34 file1
-rw-rw-rwx  1 roberto  usrmail  17 Mar 11 16:17 file2
%
```



PERMESSI: ESEMPI




```
% ls -R
dir1   file1

./dir1:
file1  file2
% chmod u-rwx dir1 file1
% rm file1
rm: file1: override protection 44 (y/n)? n
% rm -r dir1
rm: cannot read directory dir1: Permission
denied
% ls dir1
can not access directory dir1
% cd dir1
dir1: Permission denied
% cp file1 dir1
cp: cannot open file1: Permission denied
%
```

UN ALTRO TIPO DI PROTEZIONE PER I FILE

L’esigenza:

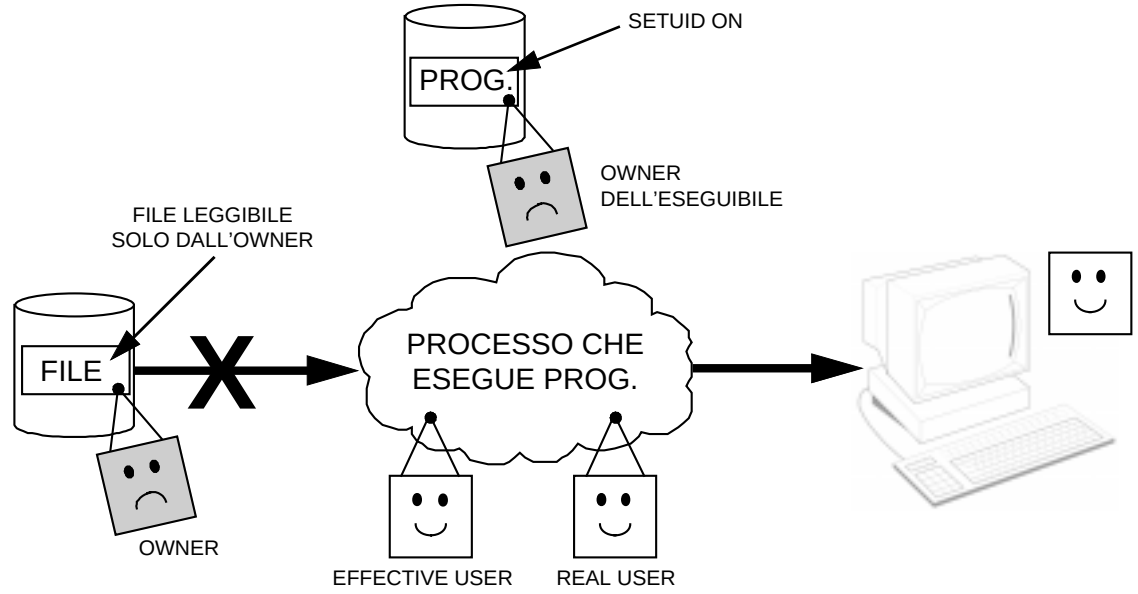
Fare in modo che un dato file sia accessibile in una specificata modalità solo attraverso uno specifico programma

La soluzione:

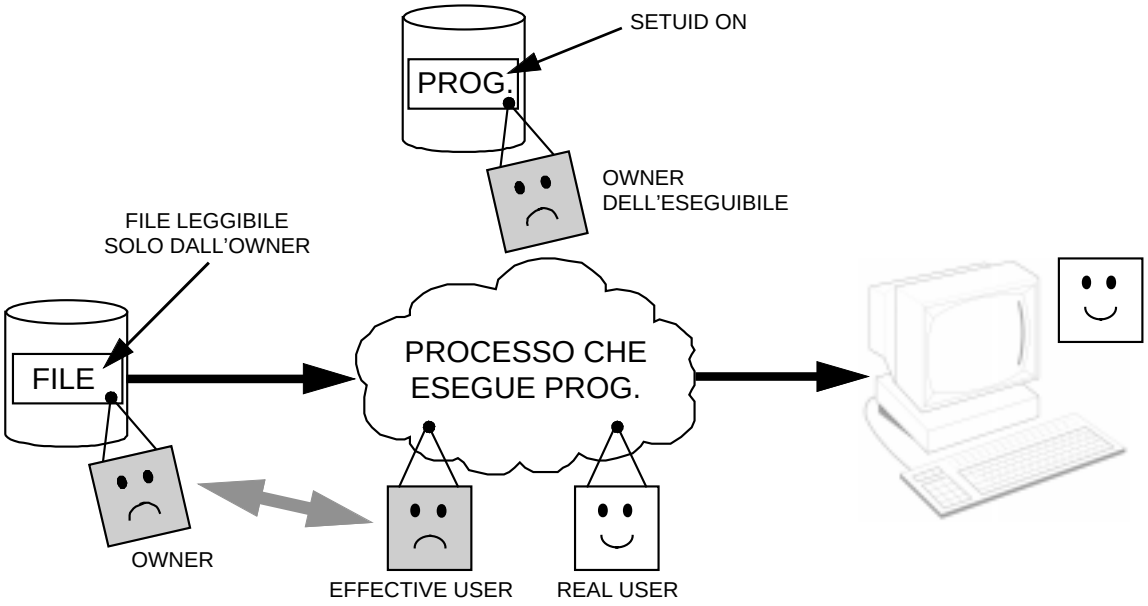
programma "setuid": quando viene eseguito, il programma si comporta come se esso fosse eseguito dall'owner del suo eseguibile (e non da chi lo ha lanciato)

ESEMPIO

Programma normale



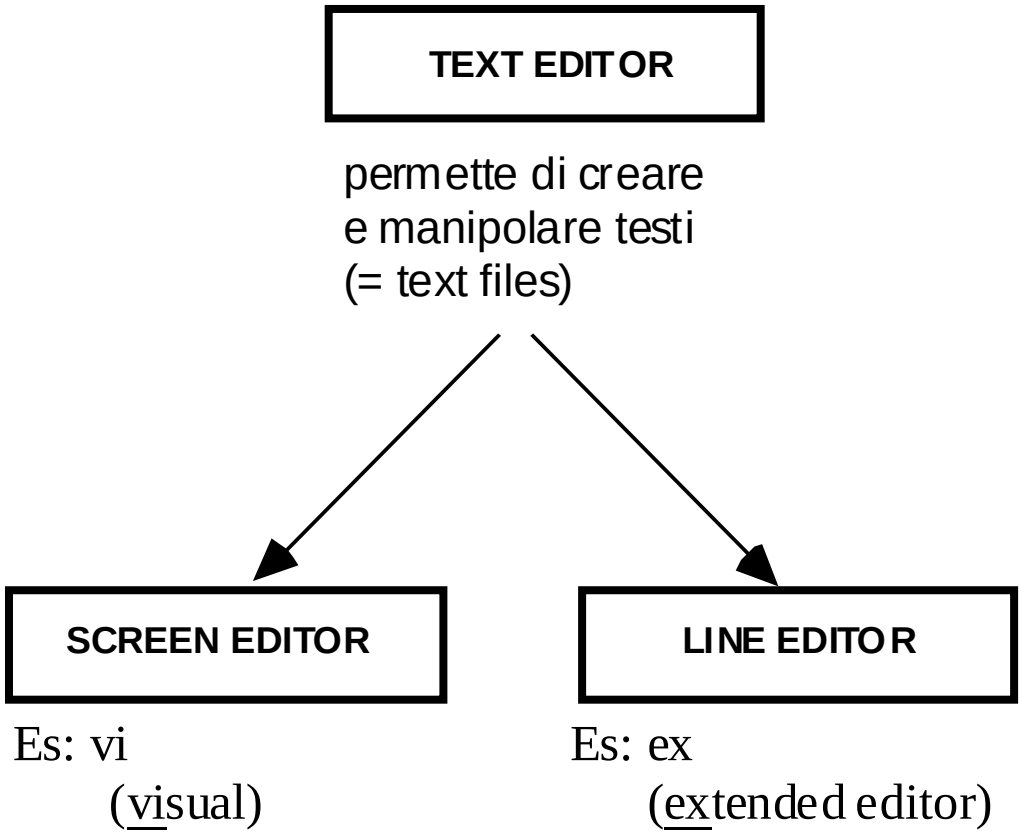
Programma set user id



(Es.: login e /etc/passwd; lpr e file di spooling,...)

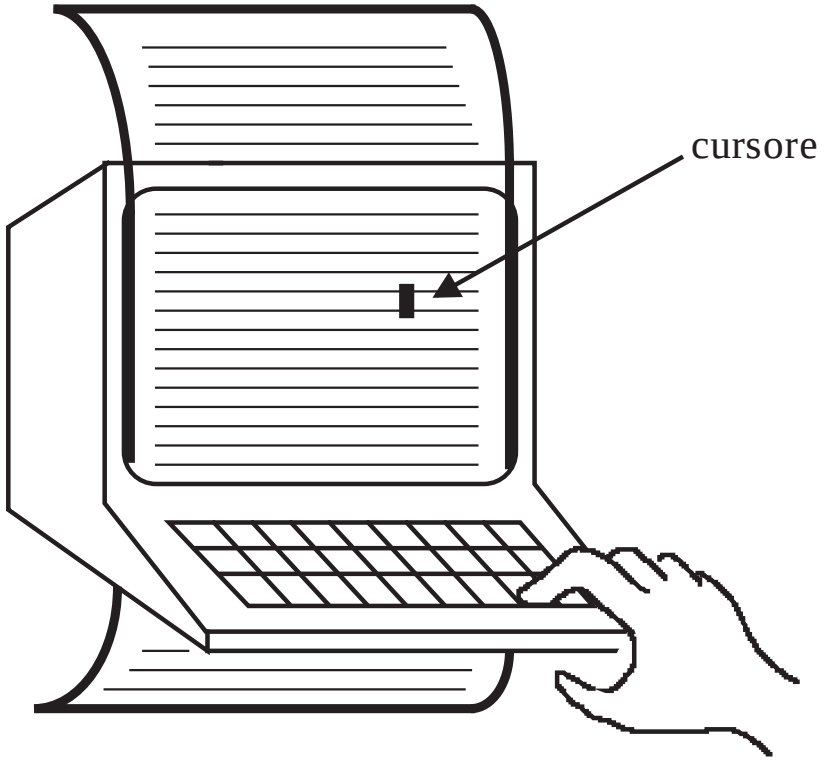
7. EDITORS

TIPI DI EDITORS

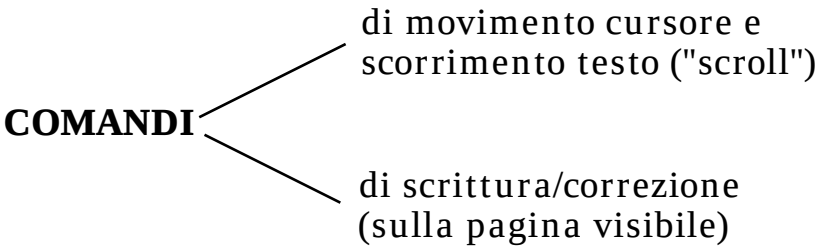


SCREEN EDITORS

Per **terminali video**



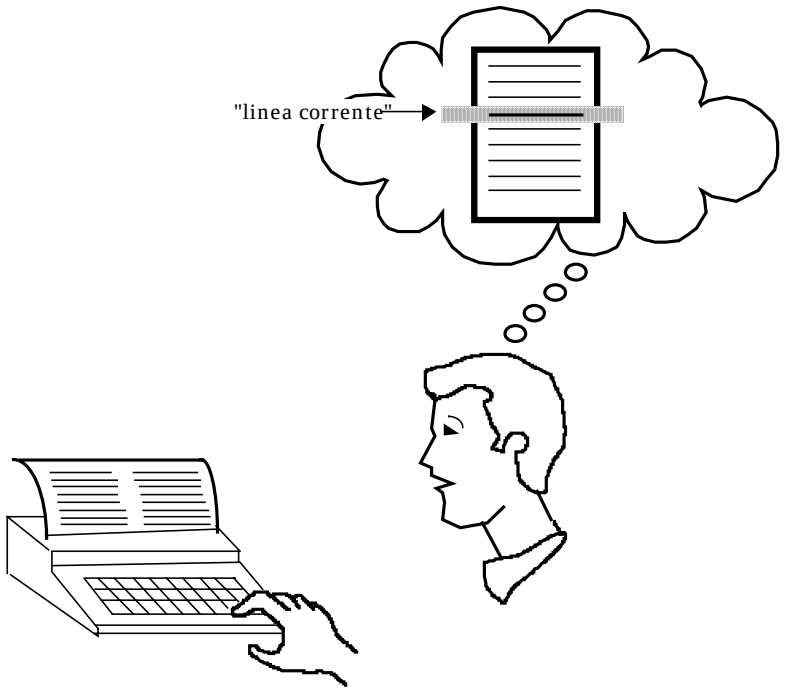
- il video è una "finestra" sul testo
- il cursore è la "penna" dell'utente



LINE EDITORS

Per **terminali stampanti**

(c'erano una volta....)

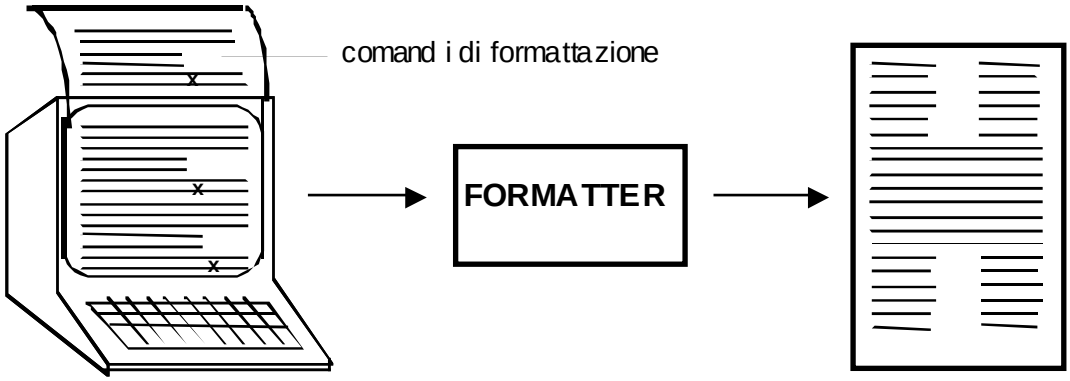


- il testo è "dentro" la macchina (e non si vede)
- l'utente "pensa" alla "linea corrente"
- ogni tanto chiede la stampa del testo corretto

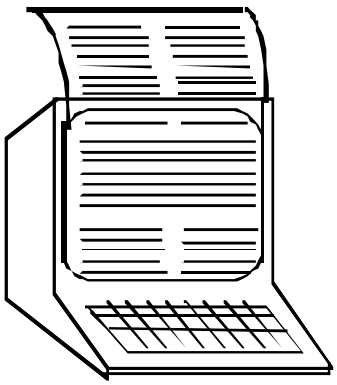
COMANDI
- di definizione linea corrente
- di visualizzazione testo (= stampa su carta)
- di scrittura/correzione (della linea corrente o sulla/e linea/e specificata/e

SCREEN EDITORS E WORD PROCESSORS

- **L'approccio "tradizionale" (screen editors)**



- **L'approccio di oggi (word processors)**

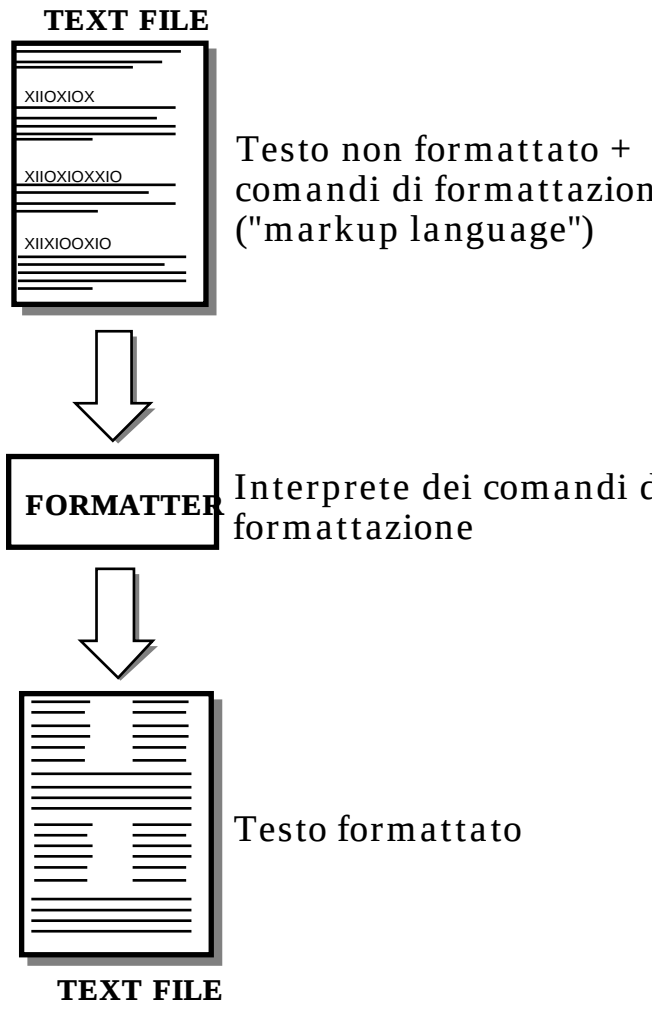


"what you see is what you get"
(wysiwyg)

TEXT FORMATTERS

Sono strumenti per "impaginare" un testo:

- marginature
- allineamenti
- salti pagina
- interlinea
- titoli
- ecc...



Esempi: `nroff, troff`

EDITORS IN UNIX

Screen editors:

`vi`	**vi**sual editor

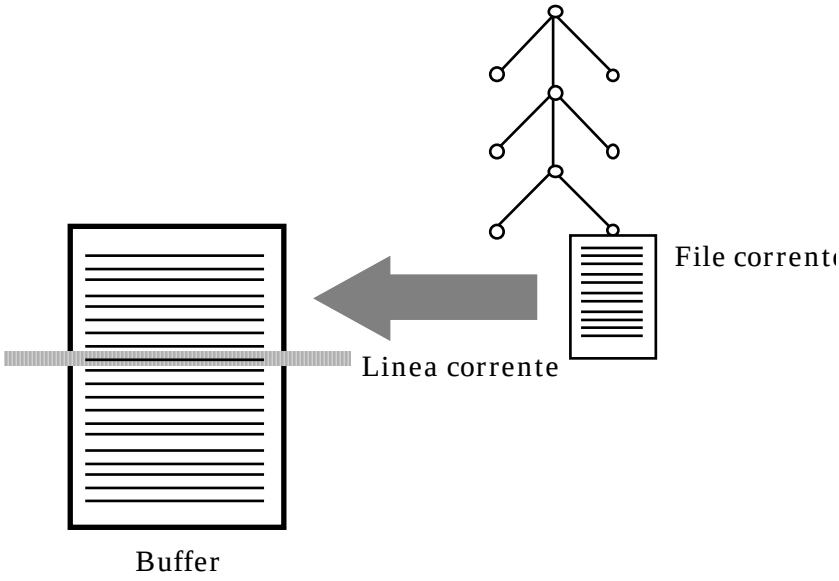
Line editors:

`ed`	**ed**itor
`ex`	**ex**tended editor: estensione di ed
`edit`	estensione semplificata di ex
`red`	**r**estricted **ed** : "read only" ed
`sed`	**s**tream **ed**itor: filtro

IL LINE EDITOR
ex

PER EDITARE UN FILE ESISTENTE

Le operazioni di editing vengono effettuate su una copia del file corrente, contenuta nel "file buffer"



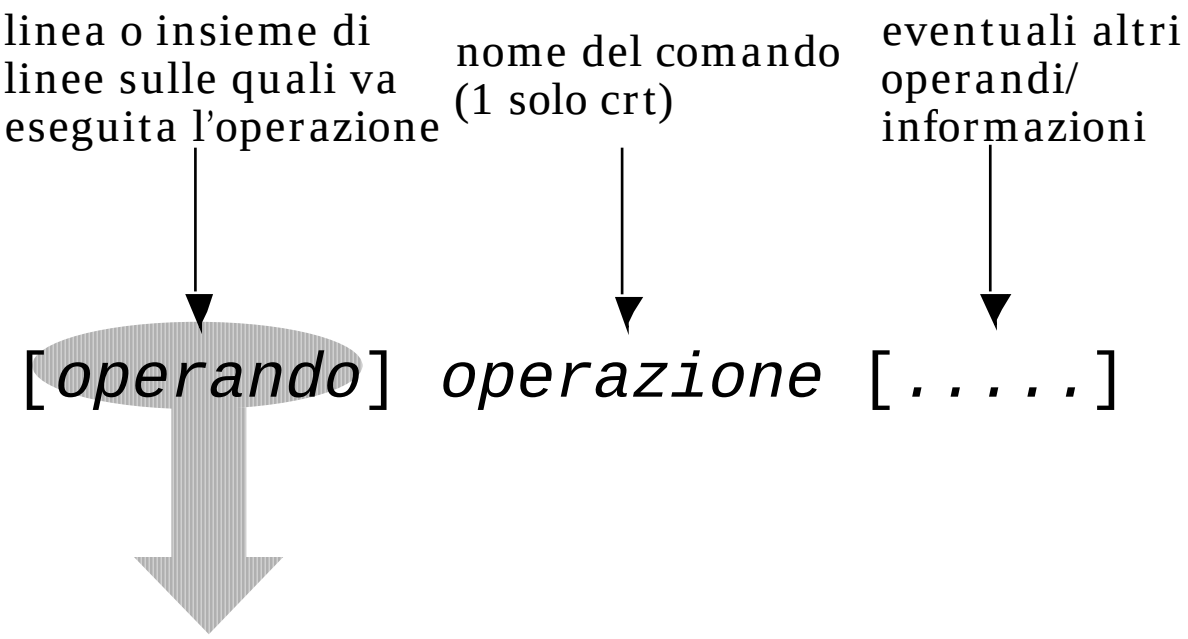
Esempio:

```
% ex inventario
"inventario" 23 lines, 453 characters
:


...
:q
%
```

PER CREARE UN NUOVO FILE

```
% ex nuovo
"nuovo" [New file]
:

...
:q
%
```

ex: FORMATO COMANDI



linea	la linea specificata
linea1,linea2	il gruppo di linee contigue fra *linea1* e *linea2* comprese
g/*pattern*/	tutte le linee che contengono il *pattern* specificato
v/*pattern*/	tutte le linee che non contengono il *pattern* specificato

FORMATO OPERANDO: *linea*

Una *linea* può essere specificata fornendone:

La posizione assoluta

Es. `5` la quinta linea

La posizione implicita

Es.
- `.` la linea corrente
- `-` la linea precedente alla corrente
- `+` la linea successiva alla corrente
- `$` l'ultima linea

Il contenuto

Es. `/ABC/` la prima linea, dopo la corrente, contenente la stringa ABC

`?ABC?` la prima linea, prima della corrente, contenente la stringa ABC

L'identificatore simbolico

Es. `'a` la linea a cui è stato attribuito dall'utente il nome a (comando mark)

La posizione relativa

Es. `.+5` `$-7` `/ABC/-5` `'a+7`

FORMATO OPERANDO: *pattern*

`/pattern/`

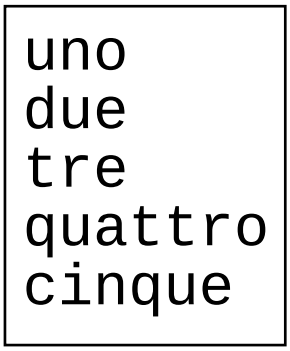
Specificato utilizzando i seguenti metacaratteri:

- • un carattere qualsiasi
- **^** inizio linea
- **$** fine linea
- * ripetizione (0....n) del carattere precedente
- **[ ]** alternativa (singoli caratteri) (anche **[c_1 - c_n]**)
- **...**

N.B. \ fa perdere il significato speciale ai metacaratteri

ESEMPIO

numeri




```
% ex numeri
"numeri" 5 lines, 22 characters
:.
cinque
:.-
quattro
:1,3
uno
due
tre
:$-1
quattro
:/tre/
tre
:
```

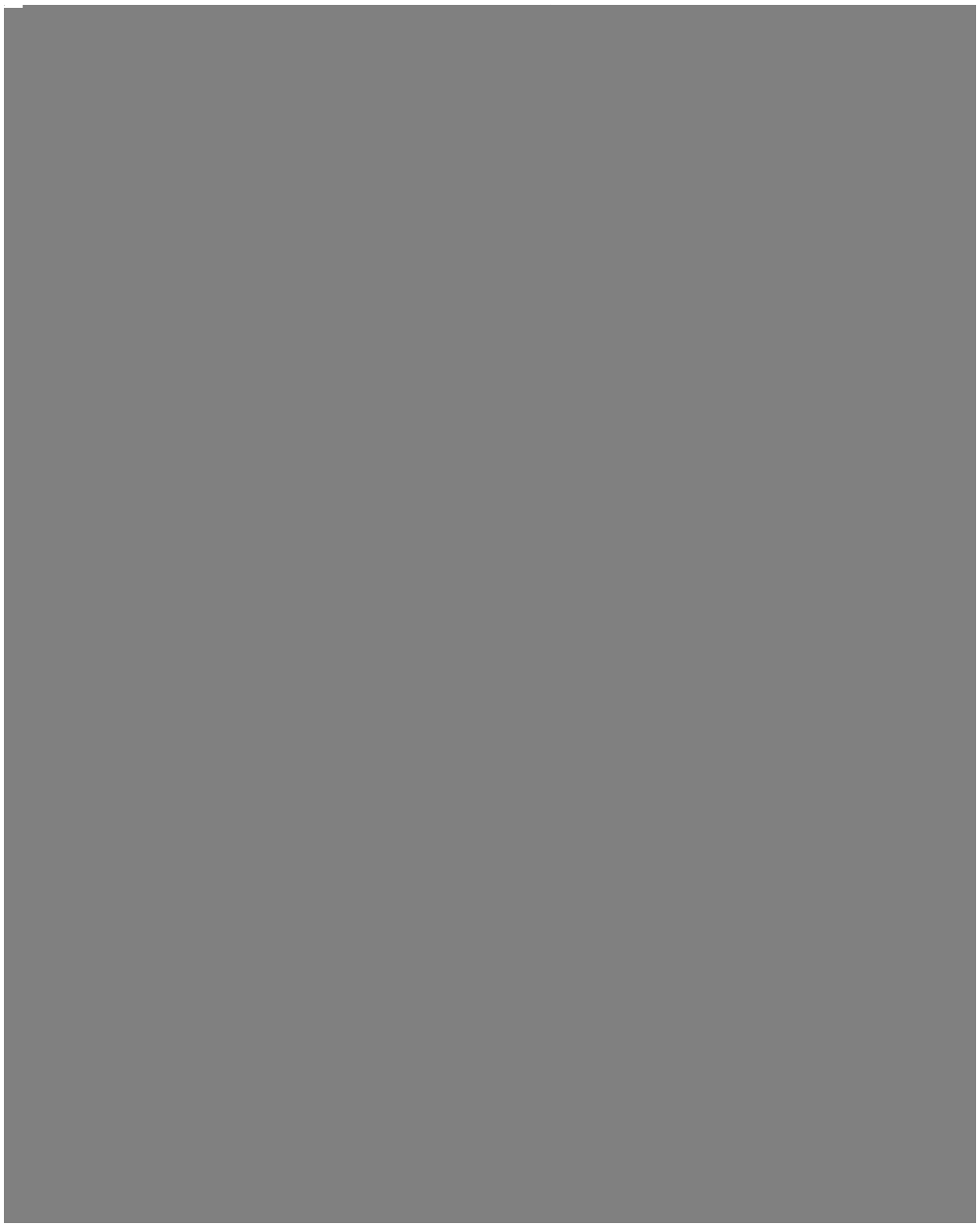
OPERAZIONI SUL BUFFER

TIPO	CODICE	NOME	ESEMPI
Visualizzazione linea o gruppo di linee	[p]	print (car. stampabili)	•,$p /a/,/z/-1p 1
	l	list (tutti i caratteri)	10,$-5l
Cancellazione linea o gruppo di linee	d	delete	1,$d /^A$/d
Inserimento di linea o gruppo di linee	a	append (dopo)	$-7a xxxx xxxx • 0a xxxx •
	i	insert (prima)	1i xxxxx xxxx •
Sostituzione di linea o gruppo di linee	c	change	•+10,/AA/c xxxx xxx •
Sostituzione di stringa (in linea o gruppo di linee)	s	substitute	1,$s/AA/BB/ (solo la prima occorrenza in ogni linea; se no suffisso g)

OPERAZIONI SUL BUFFER (SEGUE)

TIPO	CODICE	NOME	ESEMPI
Spostamento di linea o gruppo di linee	m	move	•,•+5m/xx/-1 spostamento dopo la linea /xx/-1
Copia di linea o gruppo di linee	t	transfer	5,7t$ copia dopo la linea $
Attribuzione di un nome a una linea o gruppo di linee	k	key (mark)	5kx per riferire la 5: 'x (un solo char)
Annullamento ultima modifica	u	undo	u (senza altri parametri)
Visualizzazione numero di linea	=		• = $ = 5 =

OPERAZIONI SUL FILE CORRENTE



OPERAZIONI SUL FILE CORRENTE

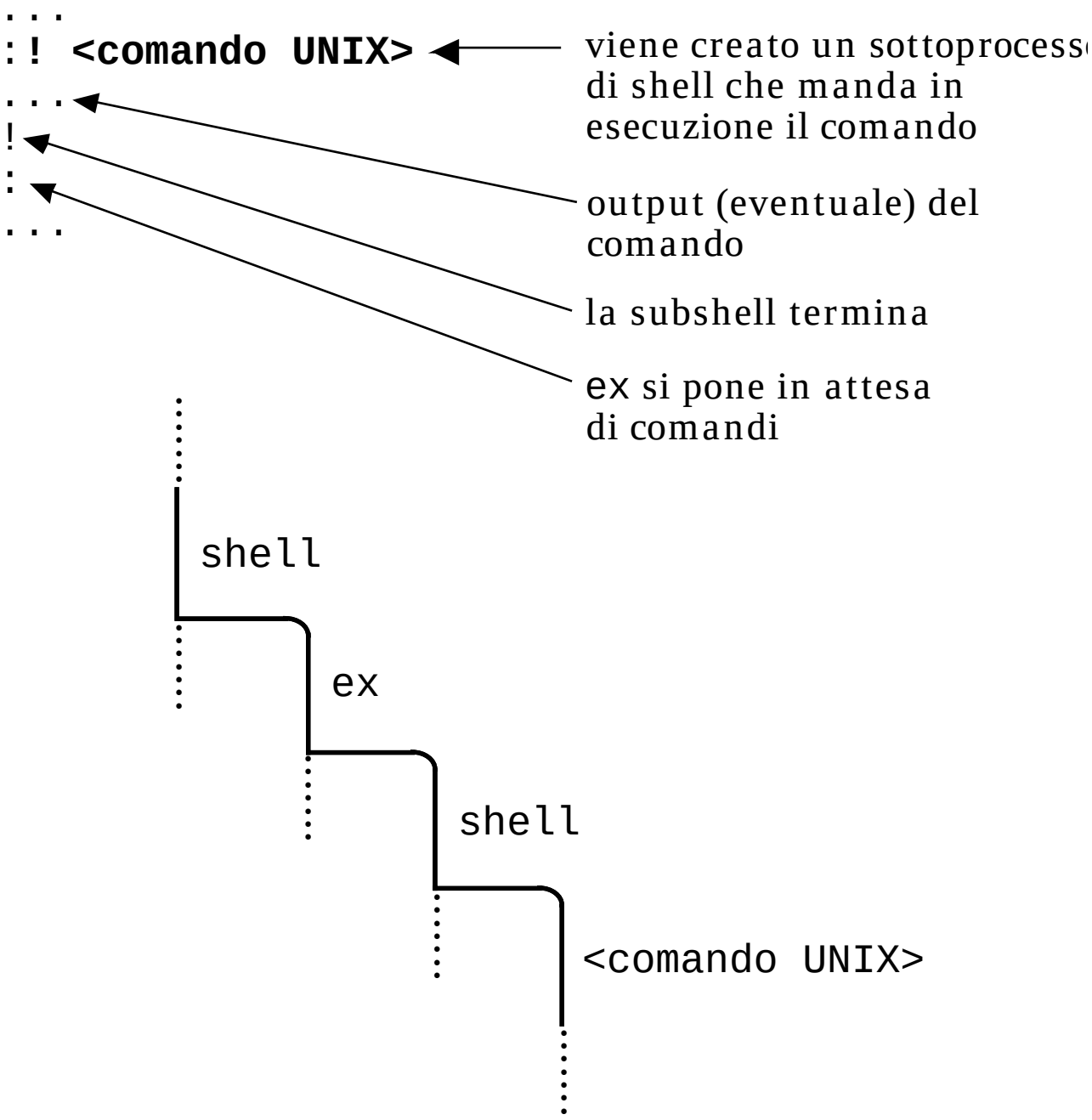
TIPO	CODICE	NOME	ESEMPI
Definizione/modifica del file corrente	`f`	file	`f alfa` `f <---` stampa il nome del file corrente

Caricamento nel buffer del file corrente	`e`	edit	`e alfa`
			equivalente a:
			`f alfa`
			`e`

Accodamento nel buffer del file corrente	`r`	read	`r alfa`
			equivalente a:
			`f alfa`
			`r`

Sostituzione del file corrente a partire dal buffer	`w`	write	`w alfa`
			equivalente a:
			`f alfa`
			`w`

IL COMANDO !


:!
<comando UNIX>
viene creato un sottoprocesso di shell che manda in esecuzione il comando
output (eventuale) del comando
la subshell termina
ex si pone in attesa di comandi
shell
ex
shell
<comando UNIX>

8. LA SHELL

CHE COS'È UNA SHELL

È un interprete di comandi, che realizza l'interfac-cia fra l'utente e il sistema

È un normale programma (comando), come tutti gli altri

In Unix sono disponibili varie shell:

- Ogni utente può utilizzare la shell che desidera, e cambiarla durante la sessione d'uso
- Al login viene lanciata la shell di default, specificata nel file `/etc/passwd` per ogni utente

Esempio:

```
% grep roberto /etc/passwd
roberto:x:207:100:Roberto
Polillo:/usermail/roberto:/bin/csh
%
```

ESEMPI DI SHELL

Shell	Comando	Chi	Complessità relativa (linee di codice)
rc	rc	B. Rakitzis	1,00
Bourne Shell	sh	S.R. Bourne	1,17
C-Shell	csh	UCB	2,03
Bourne Again Shell	bash	GNU, LINUX	3,36
Zsh	zsh	P. Falstad	3,53
Korn Shell	ksh	David Korn (AT&T)	3,73
Tcsh	tcsh		5,27

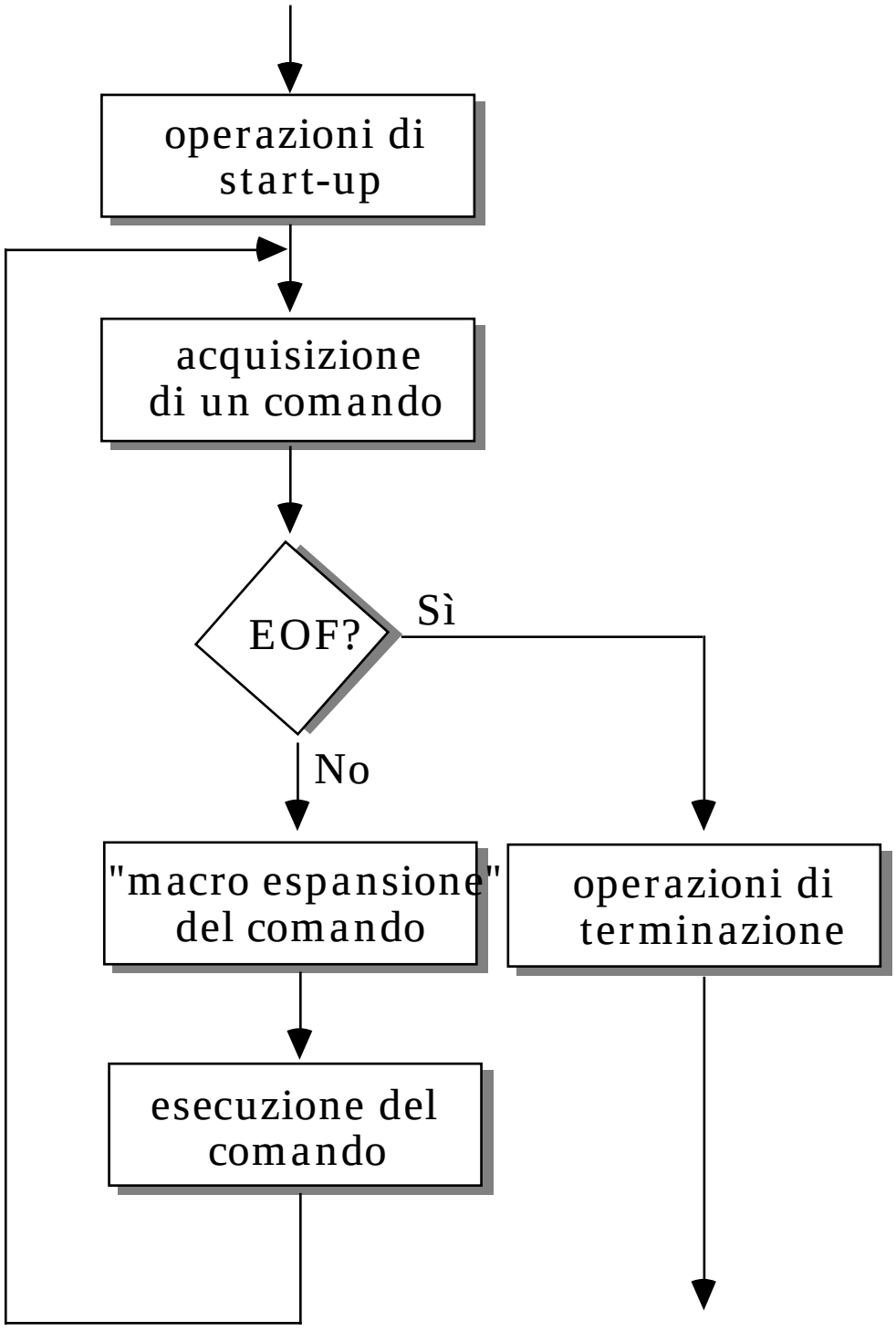
FUNZIONI TIPICHE DI UNA SHELL

La shell interpreta un vero e proprio linguaggio di programmazione che permette all’utente di con-trollare la esecuzione di comandi, sia in modo in-terattivo che in modo "batch" ("script" di shell)

Funzioni tipiche:

- esecuzione di comandi
- redirezione dell'input e dell'output
- "pipelines" di comandi
- job control
- variabili e assegnamenti
- strutture di controllo
- "scripts"
- supporto all'uso interattivo
- supporto al debugging degli script

CICLO DI ESECUZIONE DELLA SHELL



ESEMPIO

<operazioni di startup>

**%echo I files: * stanno nella **

>directory corrente"."

I files: aa bb cc stanno nella directory corrente.

%^D

<operazioni di chiusura>

TIPI DI COMANDI

Ci sono tre tipi di comandi che una shell può mandare in esecuzione:

1)- oggetti eseguibili

Sono file contenenti programmi in formato oggetto

Esempio: `ls`

2)- comandi built-in

Sono comandi che realizzano funzioni semplici, eseguite direttamente dalla shell,

Esempio: `cd`

3)- script

Sono "programmi" in linguaggio di shell

Esempio: `cc`

Nota: Il sistema può riconoscere i primi perchè iniziano con un "magic number" generato dal linker

STARTUP FILES

Durante le operazioni di inizializzazione, la shell personalizza l'ambiente di uso ...

... eseguendo uno o più script di inizializzazione, contenuti in files di pathname prefissato (**startup files**) ...

... che si trovano, tipicamente, alcuni nella home directory dell'utente, altri in directory generali

Note:

- il meccanismo è diverso per ogni shell, e può essere diverso a seconda delle modalità di esecuzione della shell (shell di login oppure no, shell interattiva oppure no, ...)
- in genere i file di startup vengono definiti all'amministratore del sistema

ESEMPIO: LO STARTUP DI `sh`

I primi comandi vengono eseguiti dai files:

`/etc/profile`

`$HOME/.profile`

(nel caso di esecuzione interattiva)

Esempio:

```
$cat .profile
TERM=vt100
export TERM
stty erase "^?" kill "^U" intr "^C"eof
"^D"
PATH=’.:~/bin:/bin:/usr/bin’
```

CAMBIARE SHELL: ESEMPIO

Possiamo lanciare una nuova shell come un normale comando, ad esempio:

```
login: roberto
Password:
Last login:Tue May 23 21:17:30 ...
Sun Microsystems Inc. ...
% sh
$ ksh
$ csh
% ^D $ ^D
$ ^D % ^D logout
```

LE SHELL PIU' DIFFUSE

Bourne shell: sh

La prima, la più diffusa, la più semplice

C shell: csh

Sviluppata a Berkeley, ha introdotto funzioni per l'uso interattivo

Korn shell: ksh

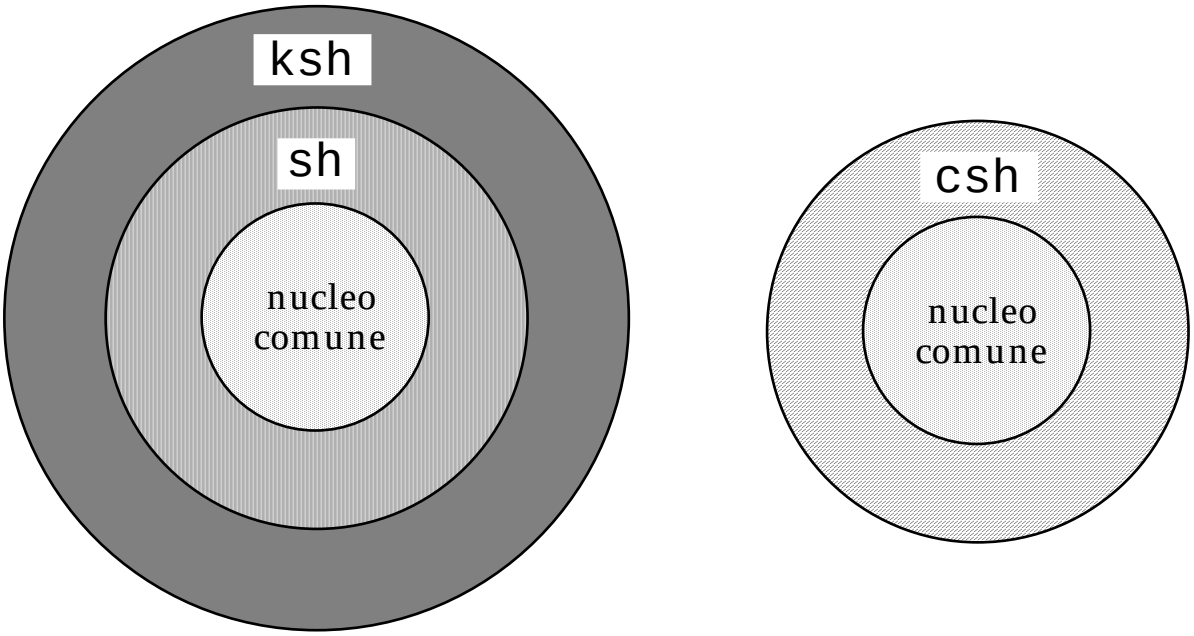
La più recente delle tre; combina utili caratteri-stiche delle prime due. È un superset di sh

Nota

POSIX.2 ha adottato più o meno sh, con caratteri-stiche di ksh

RELAZIONI FRA sh, csh E ksh

Le tre shell principali hanno un nucleo di funzioni in comune:



In questo corso, descriveremo:

- le principali funzioni del nucleo comune
- alcune funzioni specifiche (principalmente di `sh`, e quindi anche di `ksh`).

  In tal caso, nel seguito sarà sempre specificato a quale shell ci si riferisce

9. REDIREZIONE DELL'INPUT/OUTPUT

FILE STANDARD

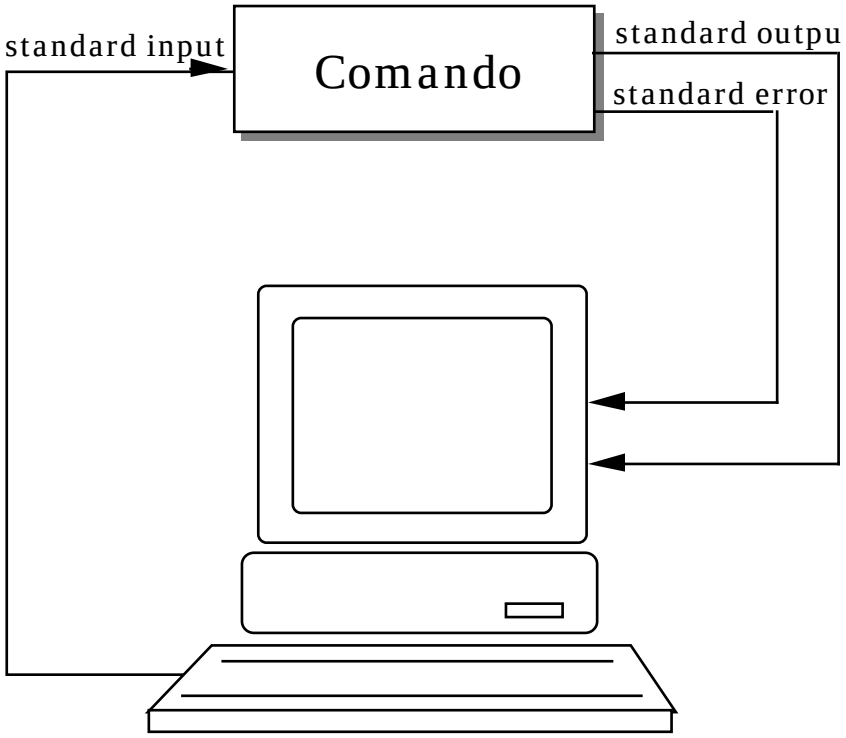
Normalmente, un programma (comando) opera su più file

In Unix esiste il concetto di **file standard**:

File standard	**Che cos'è**
standard input	il file da cui normalmente il programma acquisisce i suoi input
standard output	il file su cui normalmete un programma produce i suoi output
standard error	il file su cui normalmente un programma invia i messaggi di errore

REDIREZIONE DEI FILE STANDARD

Normalmente, i file standard sono associati al video e alla tastiera come segue:



La shell può variare queste associazioni di default **redirigendo** i files standard su qualsiasi file nel sistema

RICHIAMI SULL' I/O IN C

La gestione dei files in C viene effettuata tramite apposite funzioni della C Standard Library

Prima di essere utilizzato, un file deve essere aperto con la funzione `fopen`:

```
#include <stdio.h>
...
FILE *fp
...
fp = fopen(name, mode);
```

(`fp` è un pointer a una struttura di tipo `FILE` che contiene informazioni sul file ed è definita in `<stdio.h>`)

Quindi si può accedere al file, ad es.:

```
fprintf(fp, "gfdjgfd");
```

Infine il file deve essere chiuso:

```
fclose(fp);
```

FILES STANDARD IN C

In un programma C i files standard sono sempre disponibili, senza che occorra aprirli o chiuderli, poichè ciò è fatto automaticamente

I relativi file pointer sono resi disponibili nelle seguenti variabili (di tipo FILE *):

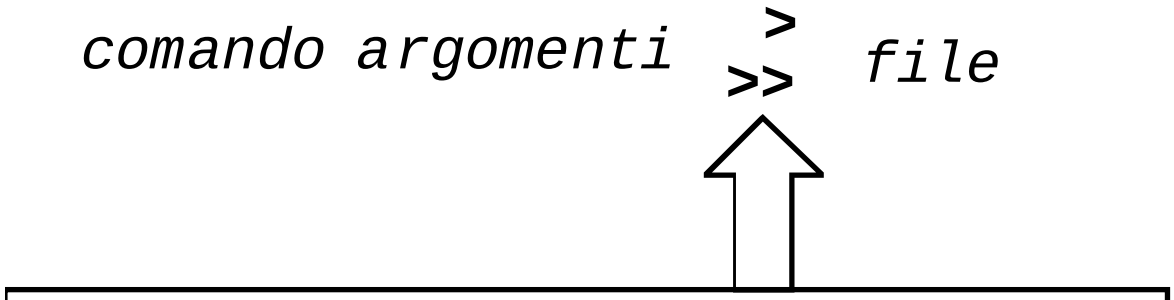
`stdin`	standard input
`stdout`	standard output
`stderr`	standard error

dichiarate in `<stdio.h>`

Esempio:

```
#include <stdio.h>
...
fprintf(stdout, "ciao")
```

REDIREZIONE DELLO STANDARD OUTPUT



Redirige lo standard output del comando sul *file*:

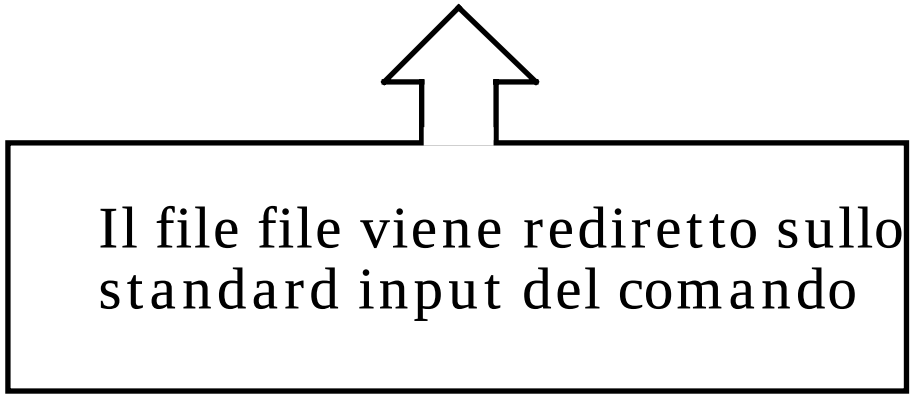
- se `file` non esiste, viene creato
- se `file` non esiste, viene riscritto (>) oppure il nuovo output viene accodato (>>)

Esempio:

```
% who
roberto  pts/1   Jun 10 23:12
  (polillo.inet.it)
% who > whofile
% who > /dev/pts/1
roberto  pts/1   Jun 10 23:12
  (polillo.inet.it)
%
```

STANDARD INPUT REDIRECTION

command arg1 ... argn < file



IL COMANDO `cat`

`cat` *`file...`*

"concatenate"

Concatena i *`file`* e li scrive sullo standard output...

```
% cat file1 file2
ei fu
siccome immobile
% cat file1 file2 > file3
%
```

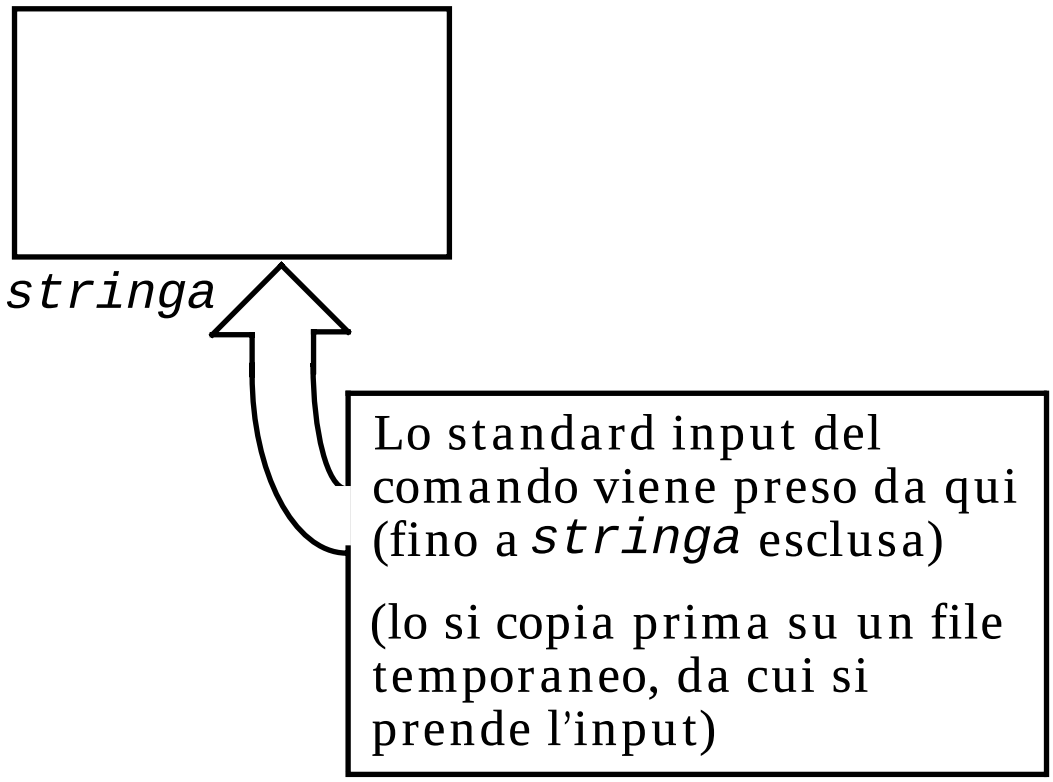
file1	ei fu
file2	siccome immobile

... a meno che manchino gli argomenti, nel qual caso scrive lo standard input sullo standard output

```
% cat < file1
ei fu
% cat < file1 > file3
% cat > file3 < file1
%
```

HERE INPUT

comando argomenti **<<** *stringa*



Esempio:

```
%cat << :
caro amico,
leggi questa
lettera
:
caro amico,
leggi questa
lettera
%
```

REDIREZIONE DELLO STANDARD ERROR (I)

In sh e ksh:

comando argomenti **2>** *file*

2>>

(Analogo a > e >>)

Esempio:

```
$ rm file
file: No such file or directory
$ rm file 2> error
$ cat error
file: No such file or directory
$
```

REDIREZIONE DELLO STANDARD ERROR (II)

In csh:

comando argomenti **>&** *file*

>>&

Redirige standard output e standard error **sullo stesso file**

Nota

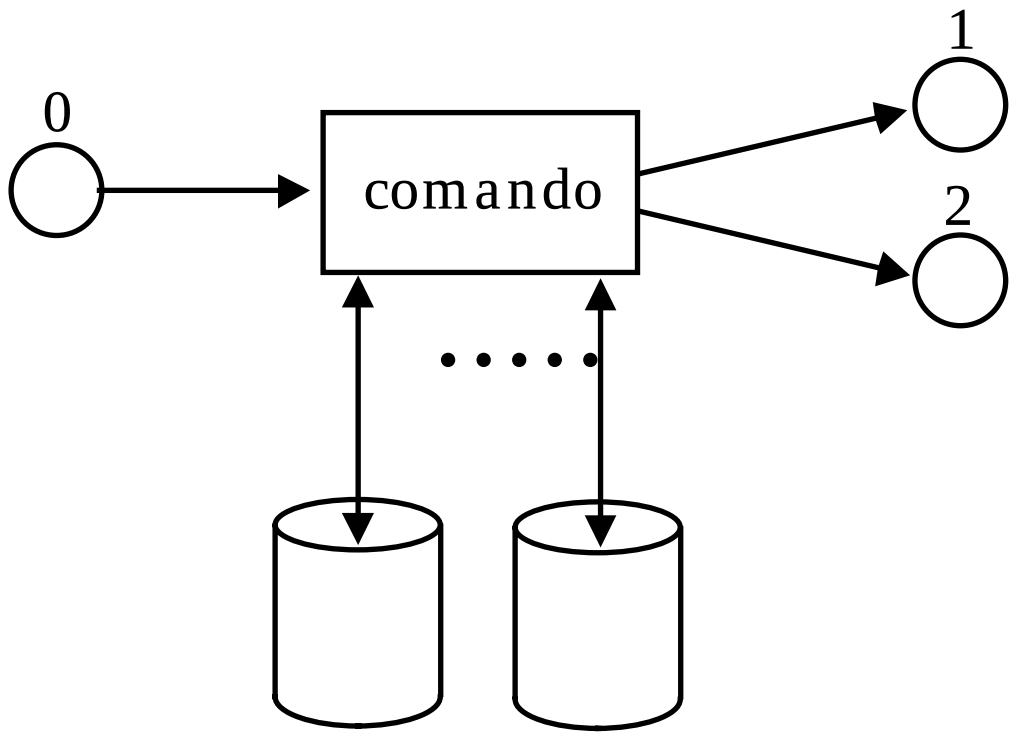
Lo standard error non si può redirigere singolarmente, se non con un trucco:

(*comando* > file1) >& file2

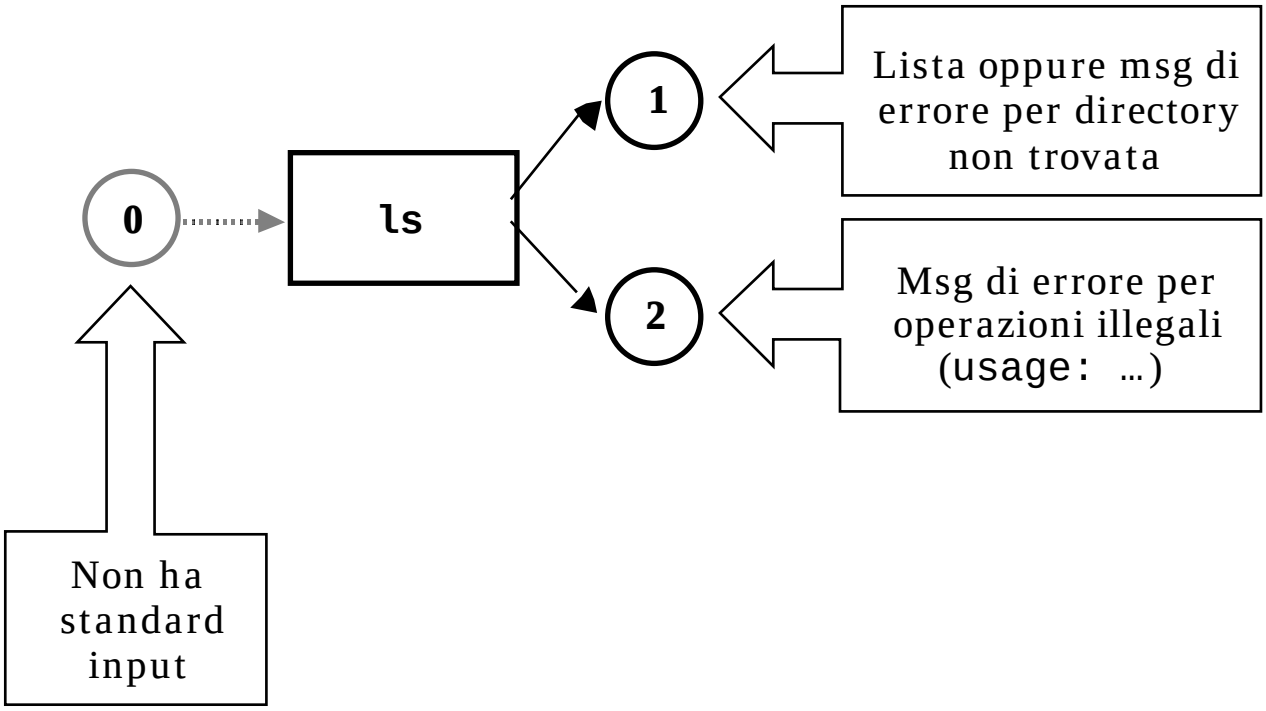
OSSERVAZIONE

Per redigere correttamente, è necessario conoscere, di ogni comando:

- come usa lo standard input
- come usa lo standard output
- come usa l'error output
- come usa eventuali altri files



ESEMPIO 1: `ls`



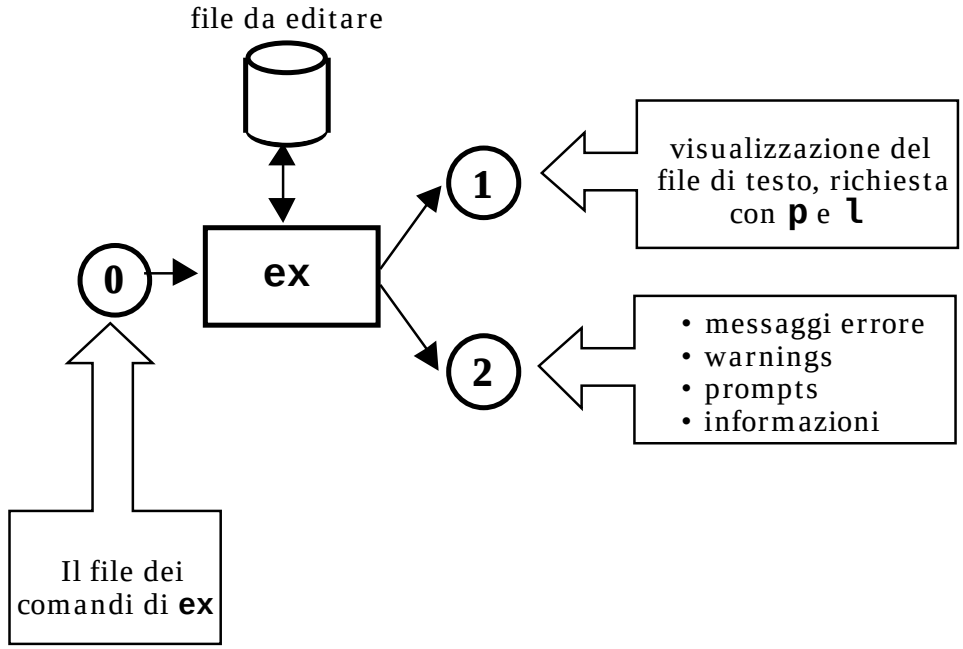
Inoltre accede ai files di sistema:

`/etc/passwd` per trovare lo user name

`/etc/group` per trovare il group name

(verificare con la propria versione di **`ls`**)

ESEMPIO 2: ex



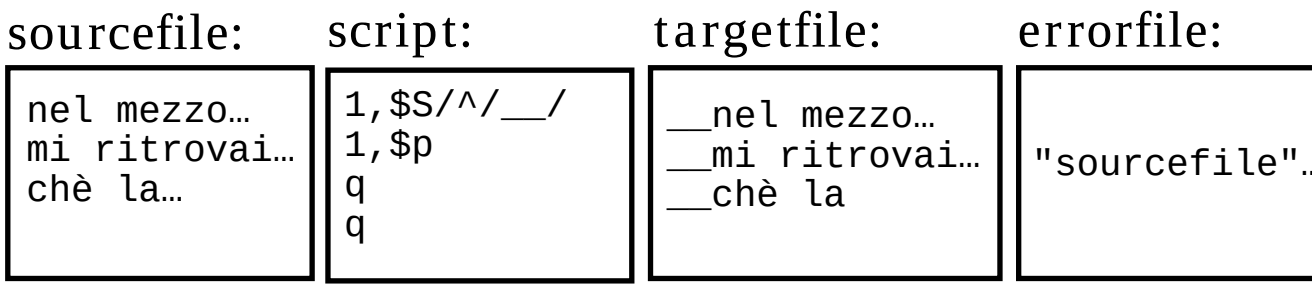
Esempio:

```
%ex sourcefile
"sourcefile" 253 car 20 lines
:1,2p          <-- input
caro           <-- output
amico          <-- output
:              <-- error
```

N.B. sed (stream editor) prende il file da editare dallo standard input

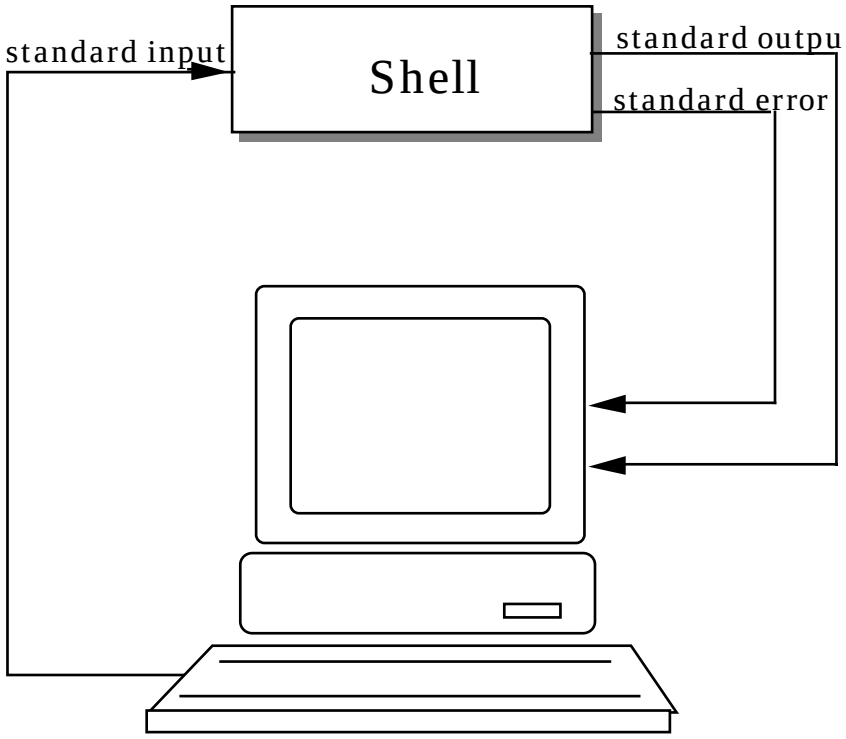
ESEMPIO: USO DI ex PER ELABORARE TESTI

```
ed sourcefile > targetfile 2> errorfile <
script
```



INPUT E OUTPUT DELLA SHELL

La shell legge i comandi dal suo standard input (fino a EOF), li interpreta, e produce i risultati sul suo standard output



... che possono essere rediretti

I files standard della shell vengono **ereditati** dai processi da questa creati

ESEMPI

```
% sh
$ echo ciao
ciao
$ ^D % sh > outfile
$ echo ciao
$ cat outfile
cat: input/output files 'outfile' identical
$ ^D % cat outfile
ciao
% echo 'echo ciao' > script
% sh < script
ciao
% sh < script > output
% cat output
ciao
%
```

COMMAND SUBSTITUTION

In una linea di comandi alla shell, un comando fra accenti gravi (‘...‘) viene sostituito dal suo stan-dard output

Esempio:

```
%echo the date today is ‘date‘
the date today is Mon Aug 19 17:32:07
MET 1996
%
```

Nota

Eventuali `newline` nell’output sono rimpiazzati da spazi

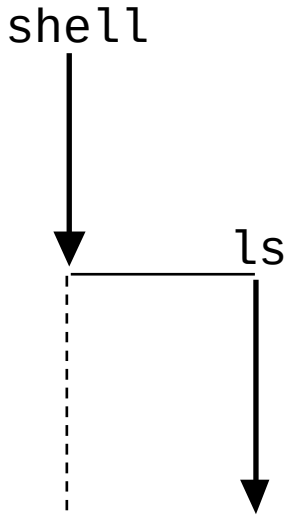
10. ESECUZIONE DEI COMANDI

ESECUZIONE DI UN OGGETTO ESEGUIBILE

La shell, utilizzando i servizi del kernel, crea un nuovo processo che esegue il comando, provve-dendo a "passargli" in modo opportuno gli even-tuali parametri

Esempio:

```
%/bin/ls dir1 dir2
a b c d
%
```



Esempio:

```
%ls dir1 dir2
a b c d
%
```

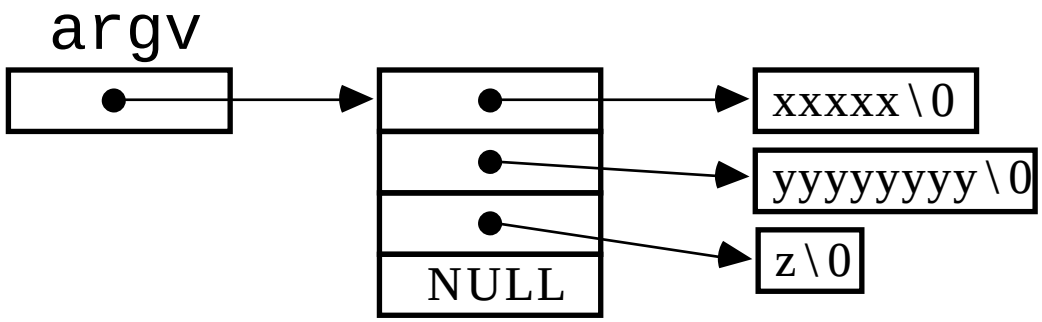
(Nel secondo caso, la shell completa il pathname del comando utilizzando la variabile `PATH` - vedi più oltre)

RICHIAMI DI C: PASSAGGIO DI PARAMETRI A UN `main`

Secondo lo standard C, un `main` viene chiamato con due parametri, ad es.:

```
int main(int argc, char *argv[])
{ ...
}
```

`argc`	numero di argomenti + 1
`argv`	pointer a un array di stringhe di caratteri che costituiscono gli argo- menti (uno per stringa)
`argv[0]`	contiene il nome del file
`argv[argc]` NULL	deve essere il pointer nullo



ESEMPIO: IL COMANDO echo

```
#include <stdio.h>
/* echo command */
int main(int argc, char *argv[])
{
  int i;
  for (i=1, i < argc; i++)
   printf("%s%s",argv[i],(argc>1)?"
":"" );
  printf("\n");
  return 0;
}
```

oppure:

```
   ...
   printf("%s%s",*++argv,(argc>1)?"
":"" );
   ...
```

Esecuzione:

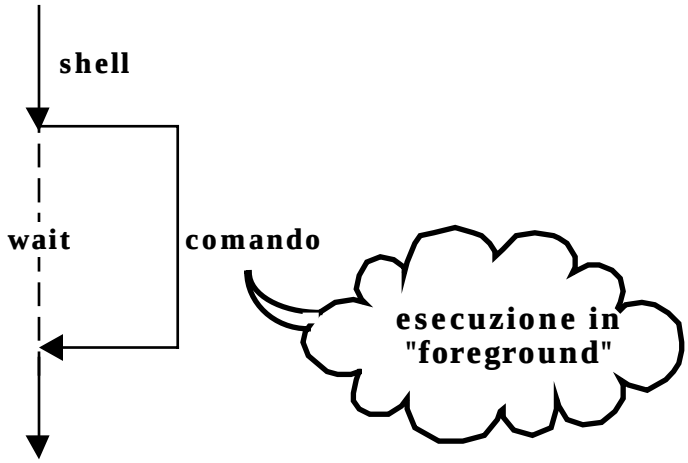
```
%echo hallo world!
hallo world!
%
```

ESECUZIONE IN FOREGROUND

Normalmente, dopo avere creato il processo che esegue il programma richiesto, la shell si pone in uno stato di wait, in attesa che il programma termini ...

... dopo di che, la shell riprende la sua esecuzione per acquisire il prossimo comando:

Esempio:



Nota: Il comando termina producento un **exit code** intero, che sarà disponibile in una variabile di shell, di nome:

`$?` in `sh` e `ksh`

`$status` in `csh`

ESECUZIONE IN BACKGROUND

In questo caso la shell crea il processo che esegue il programma richiesto, ma riprende subito la esecuzione, senza aspettare che il processo termini

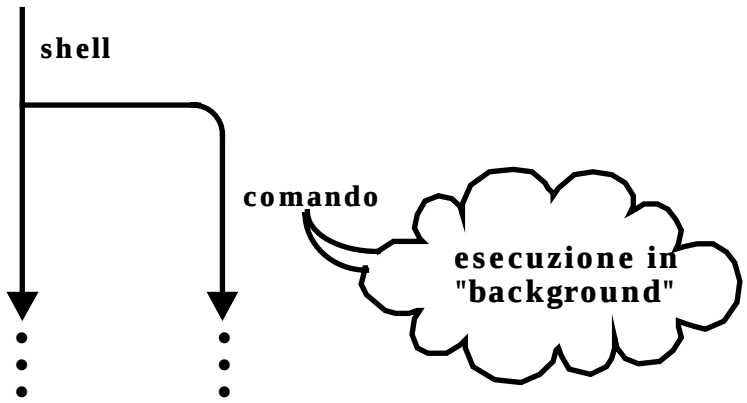
```
%comando argomenti &
```

Esempio (sh):

```
$ sort file > out &
```

`10759 <--` process-id (PID) del processo: è un numero assegnato dal kernel ad ogni processo, che lo identifica univocamente

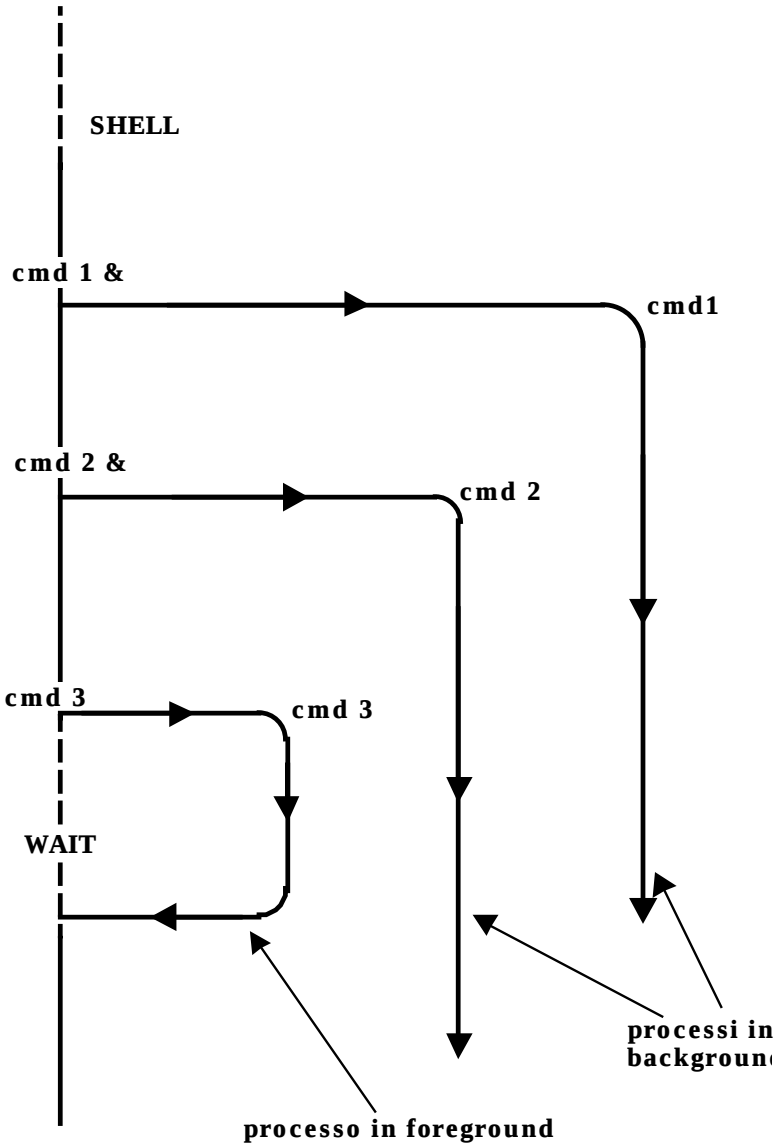
```
$
```



Nota: In questo caso l'exit code non sarà disponibile

ESEMPIO

```
%cmd1 & cmd2 & cmd3
1314
1315
<eventuale output di cmd3>
%
```

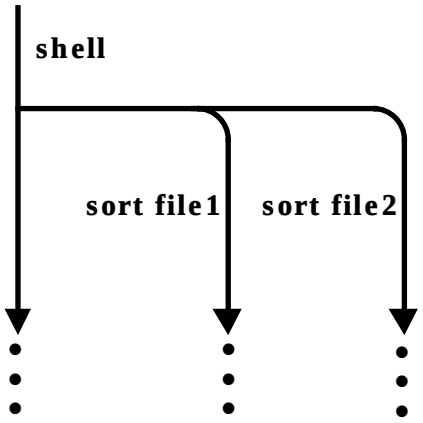


REDIREZIONE E PROCESSI DI BACKGROUND

Attenzione ai file standard non rediretti!

Esempi:

```
% sort file1 > out1 & sort file2 > out2
&
1260
1261
%
```




```
% sort file1 & sort file2 &
1267
a
b
c
1268
% a
b
c
%
file1 = file2 =  a
                 b
                 c
```

Nota: se un processo di background cerca di leggere da un terminale, riceve un segnale di errore che causa la terminazione del processo

IL COMANDO `ps`

`ps` [*`opzioni`*]

"process status"

Stampa alcune informazioni sui processi attivi

Opzioni:

nulla	solo i processi associati al terminale
`-f`	lista estesa
`-e`	tutti i processi
`...`	

IL COMANDO ps: ESEMPIO 1

```
% ps
   PID   TTY          TIME   COMD
   935   pts/5        0:01   csh
  1193   pts/5        0:00   ps
```

In questo caso: solo la shell corrente **sh** e **ps** stesso.

Legenda:

PID: process-ID

TTY: terminale che controlla il processo

TIME: tempo cumulativo di esecuzione

CMD: comando

IL COMANDO PS: ESEMPIO 2

```
% ps -f
     UID   PID  PPID   C      STIME   TTY
TIME COMD
 roberto 10715 10713 161   10:25:28   pts/1
0:02 -csh
 roberto 10885 10715  26   11:36:32   pts/1
0:00 ps -f
%
```

Legenda:

UID: process owner

PID: process-ID

PPID: parent process-ID

STIME: starting time: ora, minuto, secondo in cui il processo è stato creato

TTY: terminale che controlla il processo

TIME: tempo cumulativo di esecuzione

CMD: comando

ESEMPIO

```
login: roberto
Password:
Last login: Tue May 23 21:17:30 from
xxxx
Sun Microsystems Inc. xxxx
% sh
$ ksh
$ csh
% ps
  PID TTY TIME COMD
  801 pts/1  0:00 csh
  798 pts/1  0:00 sh
  802 pts/1  0:00 ps
  799 pts/1  0:00 ksh
  792 pts/1  0:01 csh
% ^D $ ^D
$ ^D % ^D logout
```

IL COMANDO `time`

`time` *comando*

Il *comando* viene eseguito e, al suo termine, **`time`** ne visualizza il tempo di esecuzione (in secondi o ore, minuti e secondi)

Esempio:

```
$ time sleep 5
real        5.1        <--tempo totale trascorso
user        0.0        <--tempo di esecuzione del
comando
sys         0.0        <--tempo di esecuzione del
sistema
$
```

Nota: In `csh` è built-in, ed è diverso

IL COMANDO WAIT

`wait [`*`processid`*`]`

Sospende l'esecuzione fino a che il figlio di specifi-cato *`processid`* non sia terminato.

Se nessun argomento è specificato, sospende la esecuzione fino alla terminazione di **tutti** i processi creati dalla shell

JOB CONTROL

Esistono vari comandi per il controllo dei processi

(non in sh)

Essi permettono, normalmente, di:

- terminare un processo
- sospendere un processo
- far ripartire un processo sospeso
- mandare in background un processo di foreground
- portare in foreground un processo di background
- ...

IL COMANDO `kill`

`kill` [**`-`***`signal`*] *`processid`* **`...`**

Invia il **segnale** specificato ai processi indicati.

I processi possono ignorare il segnale ricevuto, oppure trattarlo

Se non fanno nessuna di queste due cose, essi vengono terminati

Alcuni segnali:

`TERM`	`-15`	Terminazione del processo (è il segnale di default)
`KILL`	`-9`	Terminazione del processo (non può essere ignorato nè trattato)
`STOP`		Sospensione del processo (non può essere ignorato nè trattato)
`CONT`		Ripartenza di un processo sospeso

TERMINAZIONE DI UN PROCESSO

Un processo può essere terminato solo dal suo **proprietario** (l'utente che lo ha creato) o dal superuser

Per terminare un processo di foreground:

```
ctrl c
```

Per terminare un processo di background:

```
kill processid ...
kill -15  processid ...
kill -TERM  processid ...
```

In casi estremi:

```
kill -9 processid ...
```

Esempio:

```
% sleep 10000 &
[1] 11468
% kill 11468
[1] Terminated      sleep 10000
%
```

ORFANI

Se un processo rimane orfano (viene terminata la shell padre), esso viene adottato dal processo di sistema 1 (init)

Esempio:

```
% sh
$ sleep 10000 &
11336
$ ps -f
     UID   PID  PPID   C    STIME TTY    TIME COMD
 roberto 11335 11274  38 15:10:30 pts/2  0:00 sh
 roberto 11338 11335  26 15:10:49 pts/2  0:00 ps -f
 roberto 11274 11272 176 14:57:04 pts/2  0:01 -csh
 roberto 11336 11335  14 15:10:38 pts/2  0:00 sleep
10000
$ kill -9 11335   # qui -15 non basta !
Killed
% ps -f
     UID   PID  PPID   C    STIME TTY    TIME COMD
 roberto 11341 11274  12 15:11:55 pts/2  0:00 ps -f
 roberto 11274 11272   2 14:57:04 pts/2  0:01 -csh
 roberto 11336     1  14 15:10:38 pts/2  0:00
sleep 10000
%
```

HANG-UP

Al logout dell'utente, eventuali processi di background vengono automaticamente terminati ...

... a meno che essi non siano stati "protetti" con il comando

nohup *comando*

Nota

Le varie shell possono avere comportamenti diversi: verificare sul proprio sistema

11. PIPELINES

LA FILOSOFIA DI UNIX

Due principi fondamentali:

1. Ogni comando deve fare una sola cosa e farla bene

Small is beautiful

2. Quando serve un nuovo strumento, è meglio combinare strumenti esistenti piuttosto che costruirne uno nuovo

Le **pipeline** sono lo strumento fondamentale a supporto di questa filosofia

PIPELINES

L’operatore di pipe | richiede alla shell di connet-tere fra loro due (o più) comandi:

command1 | *command2* | ...

redirigendo lo standard output del primo nello standard input del successivo, e così via

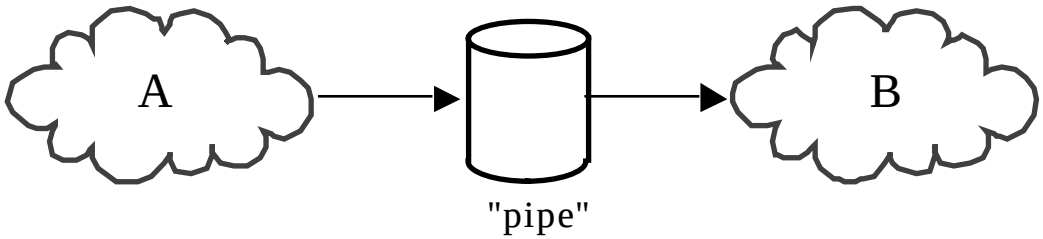
ESECUZIONE DI UNA PIPELINE

Per eseguire:

A | B

la shell:

- crea uno file temporaneo speciale, di tipo **pipe** (invisibile all'utente)
- redirige lo standard output di A sulla pipe, e crea un processo che esegue A
- redirige lo standard input di B sulla pipe, e crea un processo che esegue B



- il processo produttore (A) e il processo consu-matore (B) si sincronizzano automaticamente
- quando A e B terminano, la pipe viene distrutta

ESEMPIO

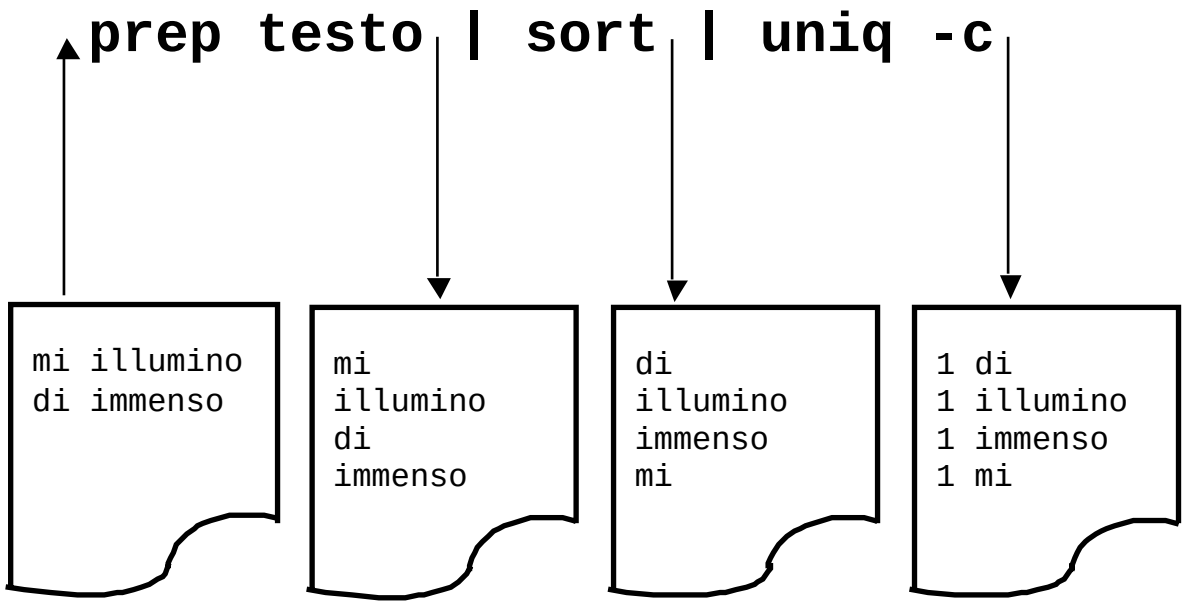
```
% ps -fe | grep roberto
 roberto 11453 11451 80 15:55:43 pts/1    0:02 -
csh
 roberto 11524 11453 10 16:12:32 pts/1    0:00
grep roberto
 roberto 11523 11453 48 16:12:31 pts/1    0:00 ps
-fe
%
```

Nota:

`ps -fe` produce sullo standard output la lista dei processi relativi a **tutti** gli utenti

PIPELINES - ESEMPIO

Listare in ordine alfabetico tutte le parole differenti di un testo, con accanto il numero di occorrenze



IL COMANDO `tee`

`tee` [**`-a`**] *`file...`*

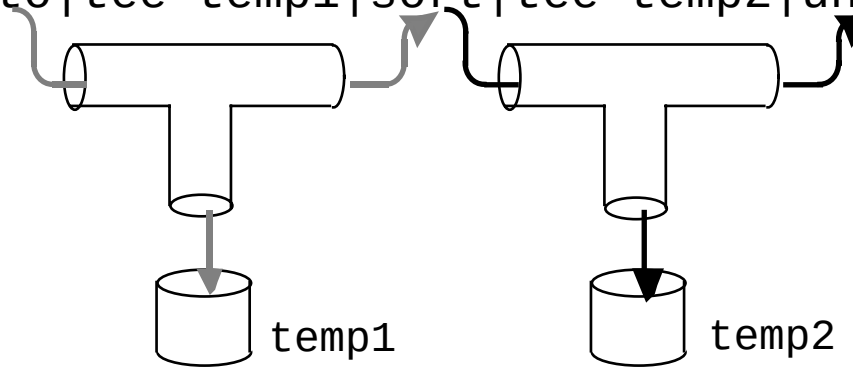
Copia lo standard input sullo standard output e su *`file`*.

Se *`file`* non esiste viene creato, se esiste viene riscritto...

... a meno che non sia specificata l'opzione `-a` ("append"): in tal caso il nuovo contenuto viene accodato

Esempio:

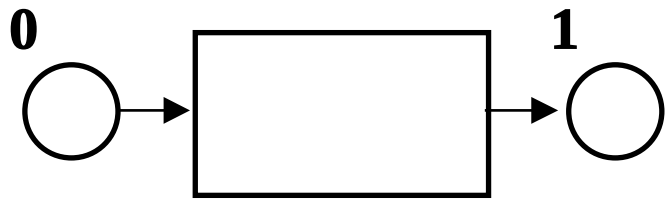
```
prep testo|tee temp1|sort|tee temp2|uniq -c
```



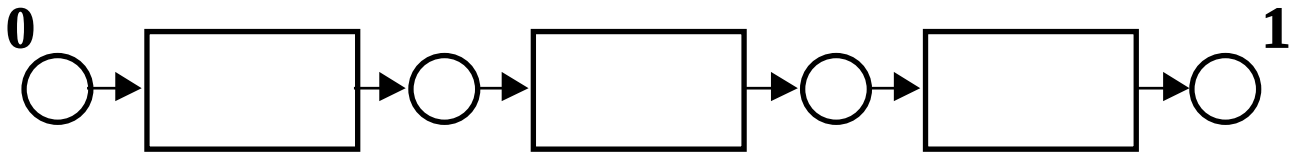
12. FILTRI

FILTRI

Sono programmi che trasformano il loro standard input nel loro standard output:



Sono molto utili per realizzare pipelines:



ALCUNI FILTRI UTILI

`sort`	ordinamento
`grep`	seleziona linee di specificata struttura
`prep`	scompone linee in parole
`uniq`	sopprime linee ripetute
`head`	mostra le prime linee
`tail`	mostra le ultime linee
`wc`	conta caratteri, linee, parole
`crypt`	codifica/decodifica
`pr`	impagina un testo

IL COMANDO `sort`

sort [*options*] [*file...*]

ordina il file ottenuto concatenando i *file* indicati, e scrive il risultato sullo standard output; se non è indicato alcun *file* ordina lo standard input.

Opzioni:

-r (reverse) inverte il senso dell'ordinamento

-b ignora i blank iniziali (spazi e tabulazioni)

-n sort numerico

.... e molte altre

IL COMANDO `sort`: ESEMPIO

```
% cat numeri
10
1
2
5
77
750
8
% sort numeri
1
10
2
5
750
77
8
% sort -n numeri
1
2
5
8
10
77
750
%
```

IL COMANDO `grep`

grep [*options*] *pattern* [*file...*]

"global regular expression print"

cerca nei *file* specificati o (se mancano) nello standard input le linee che contengono il *pattern*, e le trasferisce sullo standard output.

Alcune opzioni:

-c (count) produce solo il numero delle linee che contengono il *pattern*

-v produce solo le linee che **non** contengono il *pattern*

-n ogni linea prodotta è preceduta dal suo numero d'ordine nel *file*

...

IL COMANDO grep

pattern:

•	crt qualsiasi
[C_1 C_2...C_n]	C_1 o C_2 o....o C_n
[^C_1 C_2...C_n]	crt qualsiasi ≠ C_1 , ... , C_n newline
pattern,*pattern*	concatenazione
(*pattern*)	parentesi
*pattern**	0 o più volte il *pattern*
pattern+	1 o più volte il *pattern*
pattern{*m*}	*m* volte il *pattern*
pattern{*m*,}	*m* o *m* +1 o..... volte il *pattern*
pattern{*m*,*n*}	da *m* a *n* volte il *pattern*
^	inizio linea
$	fine linea
\$0_1 0_2 0_3$	il carattere ASCII rappresentato dalle 3 cifre ottali specificate

N.B.: i metacaratteri **^ $ • [ ] * + ( ) \ { }** vengono quotati con ****

IL COMANDO grep: ESEMPIO

Una semplice agenda telefonica

```
% cat > tel << :
roberto 48000529
marco   3452328
mario   5567843
luigi   67421467
:
% grep marco tel
marco   3452328
%
```

IL COMANDO `grep`: ESEMPI

1) lista i soli comandi eseguibili nella directory corrente

```
ls -1 -F | grep '^.*\*$'
```

2) lista le sole direttrici nella directory corrente

```
ls -1 -F | grep '^.*/$'
```

3) lista i soli files dati nella directory corrente

```
ls -1 -F | grep -v '^.*[\*/]$'
```

ESEMPI (SEGUE)

```
% ls
dir1   dir2   exe1   exe2   file1
file2
% ls -1 -F
dir1/
dir2/
exe1*
exe2*
file1
file2
% ls -1 -F | grep '^.*\*$'
exe1*
exe2*
% ls -1 -F | grep '^.*/$'
dir1/
dir2/
% ls -1 -F | grep -v '^.*[\*/]$'
file1
file2
%
```

IL COMANDO `prep`

prep [*options*] *file...*

"prepare text"

Legge i *file* nell'ordine e li trascrive sullo standard output, una "parola" per linea.

Se nessun *file* è indicato, trascrive lo standard input.

Alcune opzioni:

-d	numera ogni "parola"
-i	il file che segue è considerato "ignore file"
-o	il file che segue è considerato "only file"

...

IL COMANDO `uniq`

uniq [*options*] [*input*[*output*]]

"Unique"

Trasferisce l'*input* sull'*output* sopprimendo duplicazioni contigue di linee.

Se non sono specificati *input* o *output* usa quelli standard

Alcune opzioni:

-u in output solo quelle non ripetute

-c in output anche il contatore del numero di occorrenze

...

PIPELINES - ESEMPIO

Listare in ordine alfabetico tutte le parole differenti di un testo, con accanto il numero di occorrenze, in ordine di frequenza (prima le parole più frequenti)

```
prep testo | sort | uniq -c | sort -n -r
```

Output:

```
4    dei
3    in
3    filtri
2    questo
2    input
2    e
2    del
1    voi
1    visti
...
1    a
```

Questo file viene utilizzato come input per gli esempi di utilizzo dei filtri e dei pipeline.

La comprensione del funzionamento del meccanismo dei pipeline e dei singoli filtri vi risulterà molto utile in futuro.

Su questo input useremo tutti i filtri appena visti ma solo in alcune delle loro possibilità a voi provarli in modo completo

PIPELINES - ESEMPIO

Listare tutte le parole di due lettere di un testo, con il numero di occorrenza, in ordine di occorrenza

```
prep testo|grep'^..$'|sort|uniq -c|sort -n -r
```

Output:

```
3    in
1    vi
1    su
1    ma
1    la
1    di
```

Questo file viene utilizzato come input per gli esempi di utilizzo dei filtri e dei pipeline.

La comprensione del funzionamento del meccanismo dei pipeline e dei singoli filtri vi risulterà molto utile in futuro.

Su questo input useremo tutti i filtri appena visti ma solo in alcune delle loro possibilità a voi provarli in modo completo

IL COMANDO head

head [*-n*] [*filename...*]

Copia le prime *n* linee di ogni *file* sullo standard output (default: *n*=10)

Se nessun *file* è specificato, copia linee dallo standard input

IL COMANDO `tail`

tail [*options*] [*file*]

Copia il *file* sullo standard output, iniziando da un "posto" specificato (linee, blocchi o caratteri dall'inizio o dalla fine del *file*)

Se non è specificato nessun *file*, copia lo standard input

Esempio:

```
% cat tel
roberto 48000529
marco   3452328
mario   5567843
luigi   67421467
% tail +3 tel
mario   5567843
luigi   67421467
% tail -15c tel
igi   67421467
%
```

IL COMANDO `wc`

`wc` [**`-cClw`**] [*`file...`*]

"word count"

Conta caratteri (`-C`), bytes (`-c`), linee (`-l`), parole (`-w`) nei *`file`* specificati (o nello standard input)

Opzione di default è `-lwc`

Esempio:

```
% wc tel
     4     8    66 tel
%
```

IL COMANDO `crypt`

crypt [*password*]

Codifica/decodifica lo standard input utilizzando la *password*

Esempio:

```
% crypt roberto < numeri > encrypted
% cat encrypted
    [8Xw_2zj9}?J@[x57'kxeF-Z\
% crypt roberto < encrypted
10
1
2
5
77
750
8
%
```

IL COMANDO pr

pr[**-h** *title*] [**-l** *pagelenght*][*file...*]

Produce sullo standard output il *file* paginato ed etichettato

title	intestazione del file
pagelenght	numero di linee per pagina

N.B. È un filtro, **non** serve per stampare

13. VISUALIZZAZIONE E STAMPA DI FILES

PAGINATORI

Mostrano a video uno o più files, una pagina alla volta, ed eseguono comandi di scorrimento forniti dall'utente

Ce ne sono vari:

more

page

pg

less **more** sofisticato

IL COMANDO more

more[**-cs**][**+***startline*][**+/***pattern*][*file...*]

startline:	la linea da cui iniziare
pattern:	un pattern da ricercare
s	squeeze linee bianche

Esempi:

```
%more memo
glhlhjgmjg jhlkjh hhhhljkhljkh
hkjh hhghjfjhg fjhgfjhgf
ghjgkhjgkhjg hlkjhlkjh
--more-- (40%)
```

IL COMANDO `more` (SEGUE)

Comandi possibili dopo `--more--`:

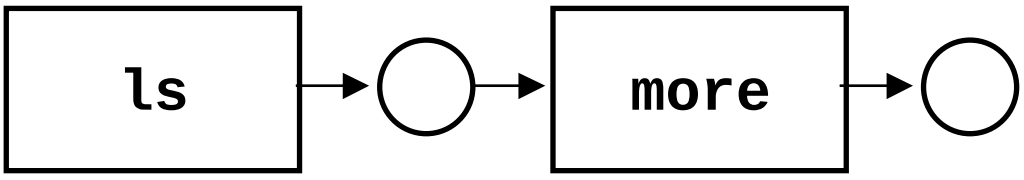
spazio	prossima schermata
return	prossima linea
n return	prossime *n* linee
q	quit
h	help
d	avanti (down) mezza schermata
*n***f**	avanti (forward) *n* schermate
b	indietro (back) una schermata
*n***b**	indietro (back) *n* schermate
v	lancia `vi` sul file visualizzato
/pattern	cerca *pattern* in avanti
n	ripeti la ricerca precedente (next)
!*command*	esegui *command* (di shell)
=	mostra num. della linea corrente
.	ripeti il comando precedente

USO DI `more` IN PIPELINE

Il file da visualizzare può essere anche fornito a `more` sullo standard input

Un uso tipico:

```
ls -R | more
```



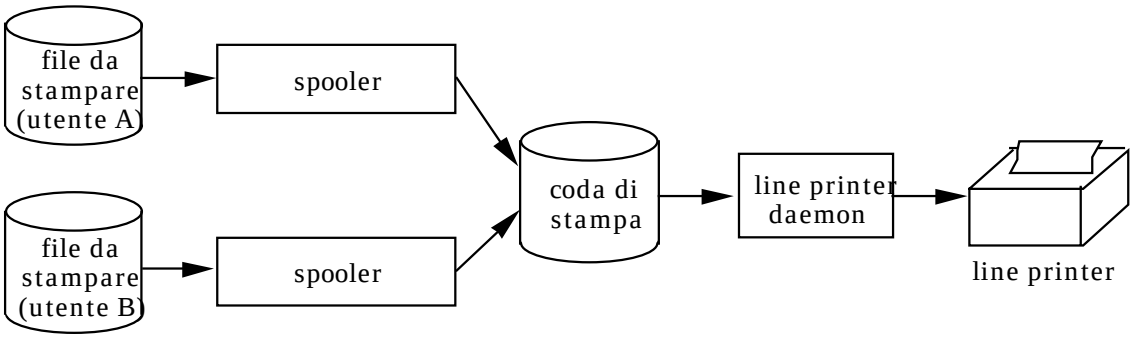
STAMPA

Per stampare un file prodotto da un comando, si potrebbe semplicemente redirigerne lo standard output sul file speciale che rappresenta la stam-pante, ad es.:

```
%sort file > /dev/lp1
```

Ma stampe concorrenti di più utenti risulterebbero interfoliate ...

... quindi è opportuno usare tecniche di spooling:



"**spool**"=simultaneous **p**eripheral **o**perations **o**ff**l**ine

COMANDI DI STAMPA

Berkeley	System V	Descrizione
`lpr`	`lp`	spooler di stampa (la stampa viene fatta dal line printer daemon)
`lpq`	`lpstat`	mostra quali job di stampa sono in attesa
`lprm`	`cancel`	cancella un job di stampa

14. SCRIPT DI SHELL

CHE COS'È UNO SCRIPT

Uno script è un programma scritto nel linguaggio di shell, registrato in un file

Esempio:

```
#Questo è un esempio di script
echo Data di oggi:
date
```

... ma ci sono script **molto** complessi!

ESECUZIONE DI UNO SCRIPT

Basta eseguire un comando di shell, redirigendo-ne lo standard input sullo script:

```
%sh  < script1
```

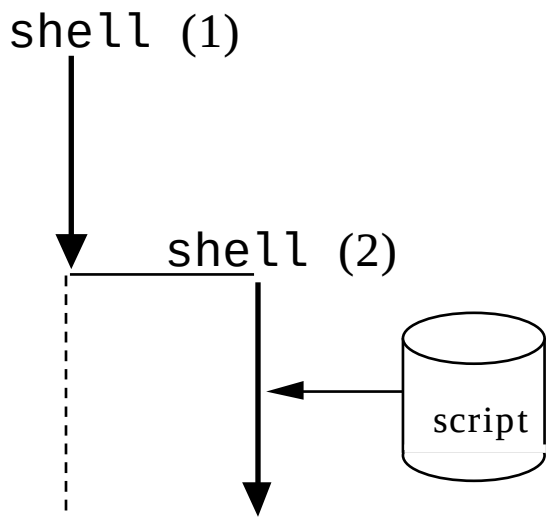
oppure, equivalentemente:

```
%sh  script1
```

oppure, più semplicemente:

```
%script1
```

Nell'ultimo caso, il kernel si accorge che è uno script (manca il magic number), individua la shell richiesta, e la esegue usando lo script come stan-dard input:



ESEMPIO

```
% sh < script1
Data di oggi:
Mon Aug 20  17:12:15  MET 1996
% sh script1
Data di oggi:
Mon Aug 20  17:12:16  MET 1996
%script1
script1: Permission denied
% chmod +x script1
% script1
Data di oggi:
Mon Aug 20  17:12:18  MET 1996
% script1 > out
% cat out
Data di oggi:
Mon Aug 20  17:12:18  MET 1996
%
```

ESEMPIO

```
% cat sveglia
ps -f
sleep 3600
echo sveglia!
% ps -f
     UID   PID  PPID   C    STIME   TTY TIME COMD
 roberto 11453 11451 182 15:55:43 pts/1 0:02 -csh
 roberto 11666 11453  26 16:47:43 pts/1 0:00 ps -f
% sveglia
     UID   PID  PPID   C    STIME   TTY TIME COMD
 roberto 11453 11451 210 15:55:43 pts/1 0:02 -csh
 roberto 11668 11667  29 16:47:49 pts/1 0:00 ps -f
 roberto 11667 11453  13 16:47:48 pts/1 0:00
/bin/sh sveglia
```

CHI ESEGUE UNO SCRIPT

La prima riga di uno script dice al kernel chi deve interpretarlo:

Prima riga uguale a	**Interprete richiesto**	**Note**
#	csh	
#!<pathname>	il programma <pathname>	la forma migliore
altrimenti	sh	

Esempio:

```
#!/bin/ksh
# Questo è uno script per ksh
echo ciao
```

SCRIPT CHE CHIAMANO SCRIPT

```
%cat a
echo sono a e chiamo b
b
echo sono ancora a
%cat b
echo sono b
ps
% a
sono a e chiamo b
sono b
     UID   PID  PPID  C    STIME TTY          TIME
COMD
 roberto 11453 11451 80 15:55:43 pts/1      0:02 -
csh
 roberto 11792 11791  3 17:02:20 pts/1      0:00
/bin/sh b
 roberto 11791 11453 14 17:02:20 pts/1      0:00
/bin/sh a
 roberto 11793 11792 33 17:02:20 pts/1      0:00
ps -f
sono ancora a
%
```

RICORSIONE

Uno script può chiamare se stesso (direttamente o indirettamente).

Esempio:

```
% cat > script1 <<:
echo a
script1
:
% chmod +x script1
% script1
a
a
a
a
a
.................
a
a
a
a
a
a
a
a
script1: fork failed - too many
processes
%
```

PASSAGGIO DI PARAMETRI A UNO SCRIPT

A uno script possono essere passati parametri come in un normale comando:

`script par1 par2 ... parN`

Esempio:

```
% cat eco
echo I parametri di chiamata sono: $1 $2 $3
% eco Mario Luigi
I parametri di chiamata sono: Mario Luigi
%
```

(Vedremo più oltre i dettagli)

IL COMANDO "."

`. nomefile`

Chiede alla shell **corrente** di interpretare lo script contenuto in *`nomefile`* (senza, quindi, creare una subshell)

SEQUENZA

cmd1; *cmd2*; ...

I comandi separati da "**;**" vengono eseguiti in sequenza

In sh, se voglio usare più righe:

```
${  ....
>.....
>}
```

PARENTESI (...)

(*cmd1***;** *cmd2***; ...)**

I comandi racchiusi fra parentesi (...) vengono eseguiti da una **subshell** della shell corrente

Esempio:

```
% ps
   PID TTY          TIME COMD
 11920 pts/1        0:01 csh
 11979 pts/1        0:00 ps
% (ps;)
   PID TTY          TIME COMD
 11981 pts/1        0:00 ps
 11920 pts/1        0:01 csh
 11980 pts/1        0:00 csh
%
```

PARENTESI: (SEGUE)

I comandi racchiusi fra parentesi (...) e {...} possono essere trattati come un unico comando (redirezione, pipeline, background, ...)

Esempio:

```
% (echo ciao; echo caro;)
ciao
caro
% (echo ciao; echo caro;)>out
% cat out
ciao
caro
%
```

15. VARIABILI DI SHELL

VARIABILI

- nome: stringa di lettere, cifre e _,iniziante con una lettera
- tipo: solo stringa
- non debbono essere dichiarate
- assegnamento (**sh**):

nome=*valore*

senza spazi

- per indicare il valore di una variabile:

$*nome*

ESEMPI (sh)

```
$ a=ciao
$ echo $a
ciao
$ a=15
$ echo $a
15
$ echo $b

$ a = ciao
a: not found
$ a= ciao
ciao: not found
$    c=ciao
$ echo $c
ciao
$ 1a=ciao
1a=ciao: not found
$ a=ciao b=hallo
$ echo $a
ciao
$ echo $b
hallo
$
```

ESEMPI (sh)

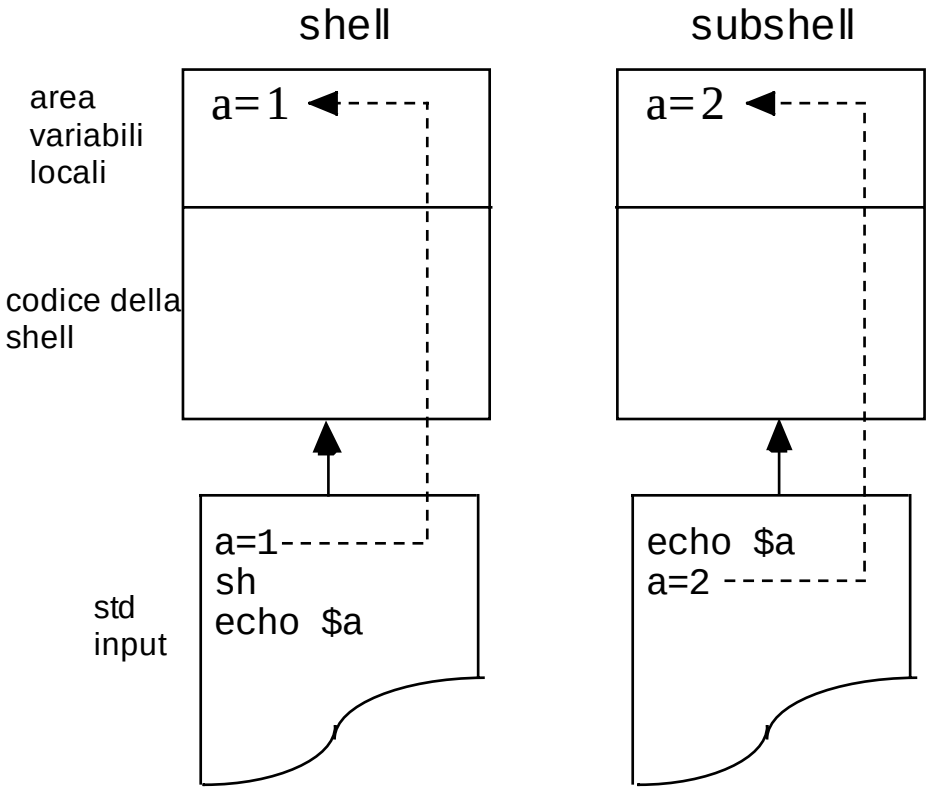
```
$ a=who
$ echo $a
who
$ $a
luigi  pts/0  May 23 15:41
robertopts/1  May 23 21:50
$ a=echo
$ b=ciao
$ c=!
$ $a $b $c
ciao !
$ $b $a $c
ciao: not found
$
```

SCOPE DELLE VARIABILI

Le variabili di shell sono **locali** alla shell corrente

Esempio:

```
$ a=1                    <---  queste due variabili hanno lo
stesso nome
$ sh                           ma sono variabili diverse
$ echo $a                <---

$ a=2
$ ^D $
$ echo $a
1
$
```



VARIABILI LOCALI: ESEMPI

Esempio 1:

```
$b=1;(echo $b;b=2;echo $b;);echo $b
 2 1
$
```

Esempio 2:

```
$cat script1
echo $a
a=2
echo $a
$ a=1; script1 ; echo $a
 2 1
$
```

Infatti, il contenuto di `( ... )` e di `script1` non è eseguito dalla stessa shell che interpreta gli altri comandi, ma da una subshell.

Nota: questa è la teoria, ma nelle mie prove (SunOs e AIX) l'esempio 1 non funziona e stampa 1 2 1 (sia su `sh` che `csh`), mentre l'esempio 2 funziona su `csh`, ma stampa 1 2 1 su `sh`

VARIABILI PREDEFINITE

Esiste un insieme di **variabili di shell predefinite** (nome e significato), che contengono informazioni utili quali:

- il nome del file che contiene lo script corrente
- i valori dei parametri di chiamata dello script
- ...

Il loro valore è assegnato direttamente dalla shell (ma a volte può essere modificato dall'utente con apposito comando)

Sono in genere diverse per ogni shell

VARIABILI PREDEFINITE: ESEMPI

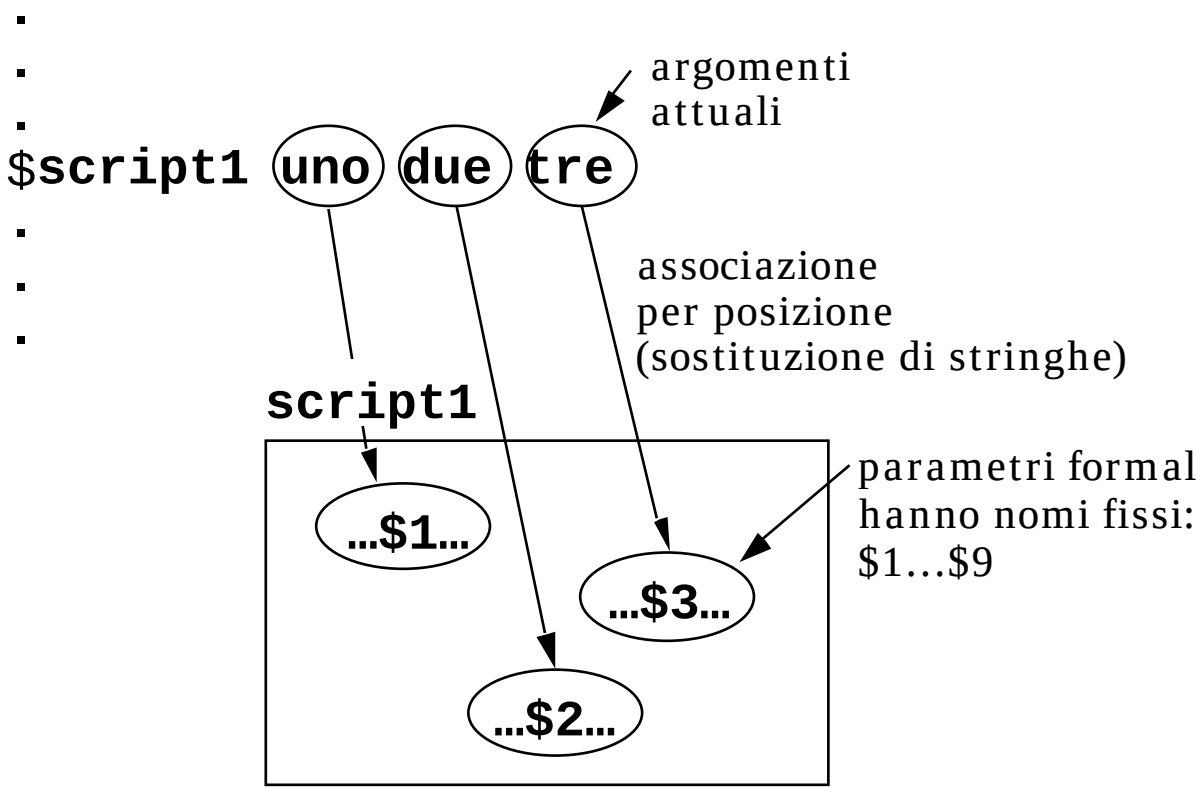
Comuni a sh, ksh, csh:

$0	nome dello script corrente
$1 ... $9	$n è il valore dell'n-esimo argomento allo script (se esiste)
$*	lista di tutti gli argomenti allo script
$$	process-id della shell corrente

In sh e ksh:

$?	exit code dell'ultimo comando eseguito in foreground
$# (posizionali)	numero dei parametri dello script
$! di	process-id dell'ultimo comando background
$-	opzioni correnti della shell

LE VARIABILI $1..$9



Esempio:

```
$saluti mario
ciao mario
$saluti
ciao
$saluti mario luigi
ciao mario
$
```

saluti

```
echo ciao $1
```

Nessun controllo di congruenza sul numero argomenti attuali/parametri formali

IL COMANDO `set` (sh)

Alle variabili **$1 $2 ...** è possibile assegnare un valore anche mediante il comando `set`

Esempio:

```
$set uno due tre
$echo $1 $2 $3
uno due tre
$echo $1Ø
unoØ
$
```

LE VARIABILI $# E $*

$# contiene il numero dei parametri posizionali (attuali) di uno script (sh)

$* contiene tutti i parametri posizionali (attuali) di uno script

Il loro valore è assegnato dalla shell, e non può essere modificato

Esempio:

```
% cat script1
echo $#
echo $*
% script1 a  b c d
4
a b c d
%
```

LA VARIABILE $0

Contiene il nome del comando in esecuzione

Esempio:

```
$ echo $0
sh
$ echo 'echo $0' > script1
$ chmod +x script1
$ script1
script1
$
```

LE VARIABILI $$ E $!

$$ process-id della shell

$! process-id dell'ultimo comando eseguito in background (sh)

Non possono essere modificati dall'utente

Esempio:

```
$ ps
  PID TTY          TIME COMD
  3643 pts/0     0:00 ps
  3634 pts/0     0:00 sh
$ echo $$
3634
$ sort file1 > out1 & sort file2 > out2
&
3644
3645
$ echo $!
3645
$
```

LA VARIABILE $? (sh)

Contiene il valore di ritorno dell'ultimo comando eseguito in foreground

Normalmente, il valore "ø" significa che il comando ha avuto successo

Non può essere modificata dall'utente

Esempio:

```
$ ls
file1  file2
$ echo $?
0
$ ls -y
ls: illegal option -- y
usage: ls -1RaAdCxmnlogrtucpFbqisfL
[files]
$ echo $?
2
$ echo $?
0
$ ls > out & ls -y 2> err &
3637
3638
$ echo $?
0
$
```

Nota: In `csh` si chiama `$status`

RICHIAMI DI C: VALORE DI RITORNO DA UN `main`

Il valore di ritorno della funzione `main` viene specificato nel programma con la funzione `exit` della C Standard Library :

```
#include <stdlib.h>
void exit(int status);
```

Esempio:

```
int main(int argc, char *argv[])
{  ...
   exit(0);
}
```

Note:

- nel `main`, `return v` equivale a `exit(v)`
- meglio usare le macro `EXIT_SUCCESS` ed `EXIT_FAILURE`, definite in `<stdlib.h>`

ESPRESSIONI CONDIZIONALI

comando1 **&&** *comando2*

Il *comando2* viene eseguito solo se *comando1* è stato eseguito con successo (cioè se `$?=0` o `$status=0`)

Esempio:

```
$rm a && echo rimosso!
rimosso!
$
```

comando1 **||** *comando2*

Il *comando2* viene eseguito solo se *comando1* è stato eseguito senza successo (cioè se `$?≠0` o `$status≠0`)

Esempio:

```
$rm a 2> out || echo errore!
errore!
$
```

ESEMPIO

```
%cc prog.c && a.out

%cc prog.c || echo errori
```

IL COMANDO `read` (sh)

read [*nomevar* ...]

Legge una linea dallo standard input della shell, e assegna successive parole alle variabili specificate

(le eventuali parole aggiuntive vengono tutte assegnate all'ultima variabile)

Esempio:

```
$cat script1
echo Please enter your name
read name
echo Your name is $name
$script1
Please enter your name
Roberto
Your name is Roberto
$script1
Please enter your name
Roberto Polillo
Your name is Roberto Polillo
$
```

IL COMANDO `readonly` (sh)

`readonly` [*`nomevar ...`*]

Le variabili specificate saranno utilizzabili in sola lettura, e non possono pertanto essere successiva-mente modificate

Se nessun argomento è specificato, elenca le variabili dichiarate `readonly`

Esempi:

```
$a=ciao
$echo $a
ciao
$readonly a
$readonly
readonly a
$a=hallo
a: is read only
$
```

16. VARIABILI DI AMBIENTE

PREMESSA

Poichè normalmente le variabili sono locali alla shell, occorre un meccanismo che consenta a questa di passare alcune informazioni ai processi da essa creati, e in particolare alle subshell

Infatti, può essere molto utile passare informazioni sull'ambiente di esecuzione: pathname della home directory, ecc.

Per questo si utilizza il concetto di **ambiente** ("environment") e di **variabili di ambiente**

Nota:
Il concetto di environment esiste in Standard C e in POSIX, ed è un concetto generale: si tratta di un meccanismo di passag-gio di informazioni a un `main` C, in aggiunta a quello solito di passaggio di parametri

RICHIAMI DI C: ENVIRONMENT

Secondo lo standard C, alla chiamata di un `main`, oltre ai parametri viene anche passata una **envi-ronment list**...

... cioè un array di stringhe di caratteri, ciascuna della forma:

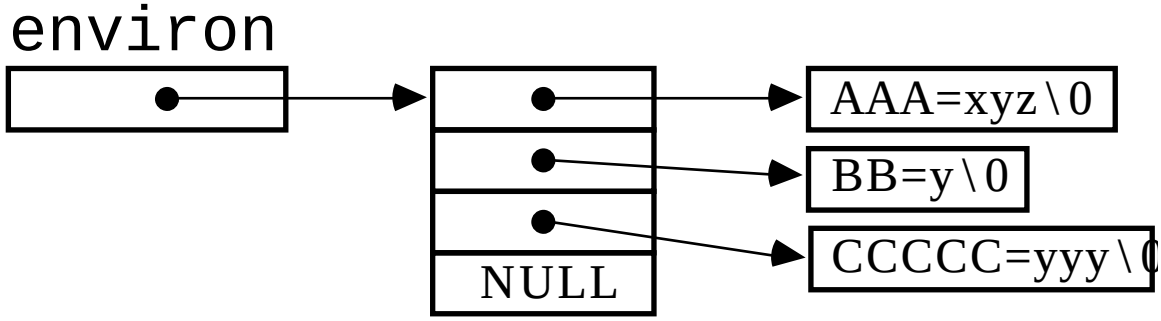
```
<nome>=<stringa di char>
```

... ed accessibile attraverso la variabile globale:

```
extern char **environ;
```

... oppure attraverso opportune funzioni standard (`getenv` e `putenv`)

Esempio:



AMBIENTE DELLA SHELL

L’ambiente della shell è una lista di coppie

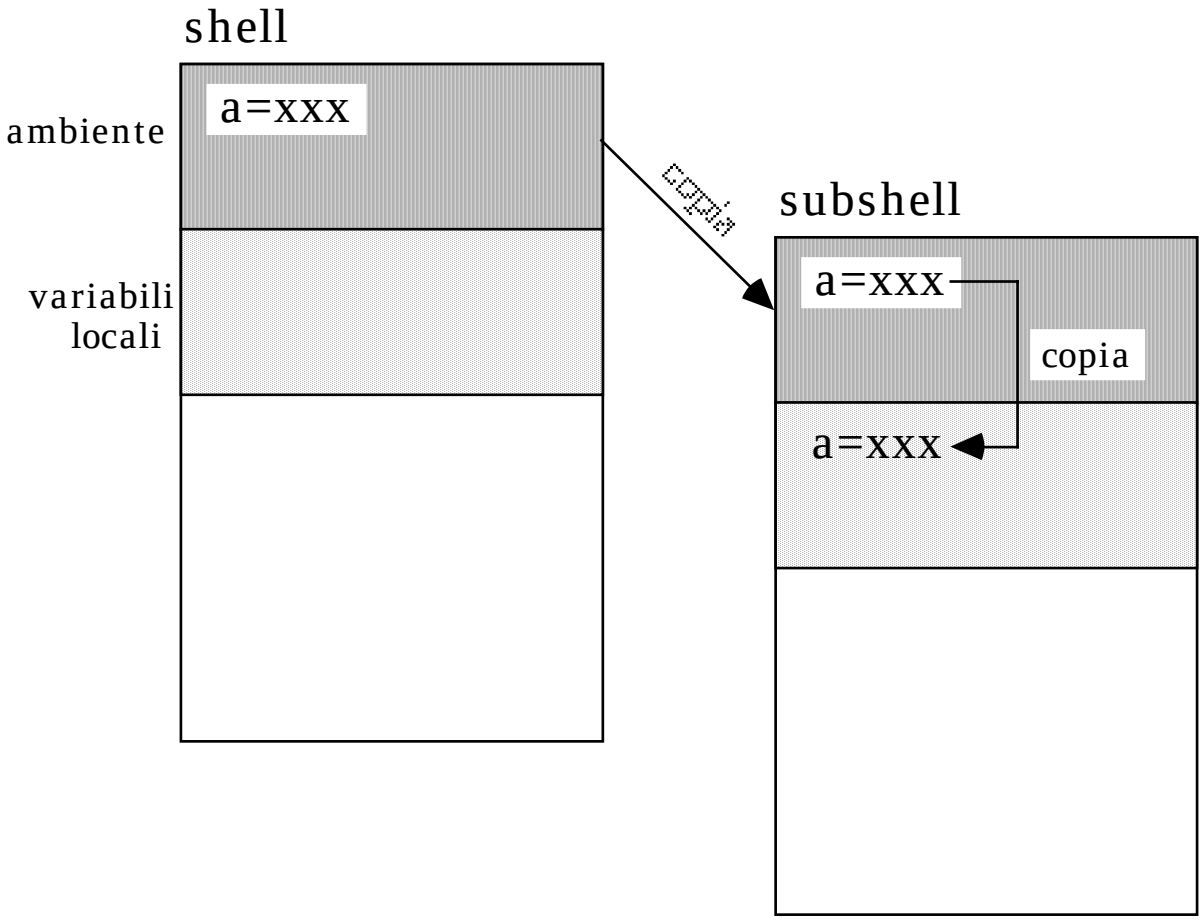
nome=valore

che viene trasmessa ad ogni processo da essa creato: sono le cosiddette variabili di ambiente

Quando viene attivata, una (sub)shell esamina l'ambiente ereditato e crea una variabile per ogni nome trovato, assegnandole il valore corrispon-dente

Queste variabili locali vengono utilizzate nel solito modo

ESEMPIO


shell
ambiente
a=xxx
variabili
locali
copia
subshell
a=xxx
copia
a=xxx

COME MODIFICARE L'AMBIENTE

Una shell può:

- inserire delle variabili locali nel proprio ambiente (per "esportarle" alle sottoshell che essa genererà)
- rimuovere delle coppie nome=valore dal proprio ambiente

Il formalismo differisce a seconda della shell usata

Nota:

- in ogni caso, un processo non può modificare l'ambiente di chi lo ha creato

IL COMANDO `export` (sh)

`export` [*`nomevar ...`*]

Le variabili indicate vengono marcate, per essere automaticamente inserite (nome e valore) nell'am-biente di tutti i processi successivamente creati

Se nessun argomento è specificato, elenca le va-riabili esportate dalla shell corrente

Nota:

In csh, per assegnare un valore a una variabile di ambiente esiste invece il comando:

`setenv` *`nome valore`*

ESEMPIO

```
$ x=A
$ export x
$ sh  #B
$ echo $x
A
$ x=B
$ sh #C
$ echo $x
A
$ x=C
$ export x
$ sh #D
$ echo $x
C
$ ^D $ echo $x   #sono tornato in C
C
$ ^D $ echo $x   #sono tornato in B
B
$ ^D $ echo $x   #sono tornato in A
A
$
```

ESEMPIO

```
$ a=a
$ export a
$ sh #2
$ b=b
$ export b
$ export
export b    #export lista solo le
            #variabili esportate
            #dalla shell corrente
$ sh #3
$ echo $a $b
a b
$
```

env

"**env**ironment"

Senza argomenti, lista l' ambiente corrente

Esempio:

```
% env
HOME=/usermail/roberto
PATH=/bin:/usr/sbin:/usr/bin:.
LOGNAME=roberto
HZ=100
TERM=vt100
TZ=MET
SHELL=/bin/csh
MAIL=/var/mail/roberto
PWD=/usermail/roberto
USER=roberto
%
```

AMBIENTE: SINTESI

L’**ambiente** di un processo è una lista di coppie nome-valore che viene trasmessa a un programma in esecuzione

Quando viene attivata, la shell esamina l'ambiente e crea una variabile per ogni nome trovato, assegnandole il valore corrispondente

I comandi eseguiti ereditano lo stesso ambiente

Se l'utente modifica il valore di queste variabili o ne crea di nuove, nessuna di queste modifica l'ambiente, a meno che non sia usato il comando `export`

L'ambiente visto da un qualunque comando è composto dalle coppie nome-valore originariamente ereditate dalla shell e non modificate, più tutte le modifiche eseguite sulle variabili esportate con `export`

VARIABILI DI AMBIENTE PREDEFINITE

Esiste un insieme di **variabili di ambiente prede-finite**, che contengono informazioni utili sull'am-biente di esecuzione

Il loro valore è in genere assegnato in un file di startup (se non lo è, la shell può usare un valore di default)

Sono in genere diverse per ogni shell

VARIABILI DI AMBIENTE PREDEFINITE: ESEMPI

Comuni a sh, ksh, csh:

$HOME	pathname della home directory
$PATH	lista di directory in cui ricercare un comando
$MAIL	pathname della mailbox dell'utente
$USER	user-id dell'utente
$SHELL	pathname della shell di login
$TERM	tipo del terminale corrente

In sh e ksh:

$IFS	elenco dei caratteri separatori
$PS1	prompt primario
$PS2	prompt secondario
$SHENV	directory in cui cercare il file di startup `.profile` (in alternativa a $HOME)

LA VARIABILE D'AMBIENTE
$HOME

Contiene il pathname della login directory dell'utente

Il suo valore viene assegnata dal processo di login, che lo preleva da `/etc/passwd`

Esempio:

```
$ grep roberto /etc/passwd
roberto:x:207:100:Roberto Polillo:/usermail/roberto:/bin/csh
$ echo $HOME
/usermail/roberto
$ HOME=/usermail
$ cd
$ pwd
/usermail
$
```

LA VARIABILE D'AMBIENTE $PATH

Contiene il "search path" per i comandi, cioè l'elenco delle directory nelle quali la shell cerca il comando da eseguire

Il suo valore di default è assegnato dalla shell

Può essere modificata dall'utente

Esempio:

```
$ echo $PATH
/bin:/usr/sbin:/usr/bin:/usr/ucb:/etc:/usr/etc:.
$ PATH=.
$ echo $PATH
.
$ ls
ls: not found
$
```

LA VARIABILE D'AMBIENTE `$IFS` (sh)

Contiene i caratteri che sh considera separatori

Il suo valore di default è assegnato da sh, ed è:
spazio, tab, newline

Può essere modificata dall'utente

Esempio:

```
$ IFS=ab
$ echoaaaaaunobbbbbdue
uno due
$ man man
m: not found
$
```

LE VARIABILI D'AMBIENTE $PS1 E $PS2 (sh)

PS1: prompt primario di sh

PS2: prompt secondario di sh (quando un comando continua su più righe)

Il loro valore di default è assegnato da sh (PS1 = $, PS2 = >)

Possono essere modificate dall'utente

Esempio:

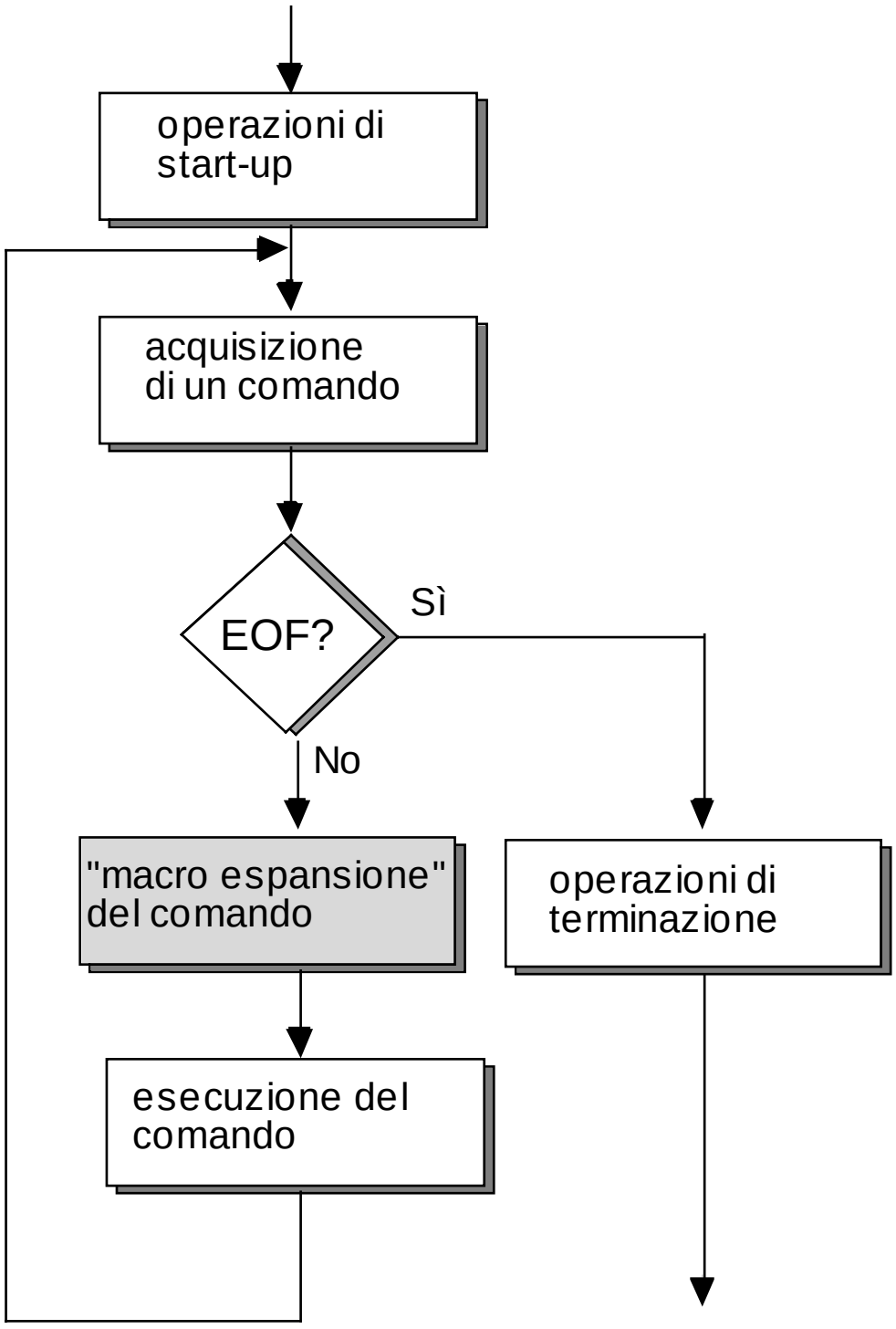
```
$ echo $PS1 $PS2
$ >
$ PS1='dimmi amore mio -->'
dimmi amore mio -->PS2='dimmi ancora  -
->'
dimmi amore mio -->PS1='O
dimmi ancora    -->O
dimmi ancora    -->O
dimmi ancora    -->OOOO>'
O
O
O
OOOO>
```

ESEMPIO: UN FILE DI STARTUP

```
$ cat /etc/profile
#ident  "@(#)profile 1.14  93/08/24 SMI" /* SVr4.0
1.3   */
# The profile that all logins get before using
their own #.profile.
trap ""  2 3
export LOGNAME PATH
if [ "$TERM" = "" ]
then
        TERM=sun
        export TERM
fi
#       Login and -su shells get /etc/profile
services.
#       -rsh is given its environment in its
.profile.
case "$0" in
-sh | -ksh | -jsh )
        if [ ! -f .hushlogin ]
        then
                /usr/sbin/quota
                # Allow the user to break the
Message-Of-             #The-Day only.
                trap "trap '' 2"  2
                /bin/cat -s /etc/motd
                trap "" 2
                /bin/mail -E
                case $? in
                0)
                        echo "You have new mail."
                        ;;
                2)
                        echo "You have mail."
                        ;;
                esac
        fi
esac
umask 022
trap  2 3
$
```

17. MACROESPANSIONI E QUOTING NELLA SHELL

CICLO DI ESECUZIONE DELLA SHELL (RICHIAMO)



SINTESI MACROESPANSIONI

"Parameter substitution"

```
%a=ciao
%echo $a    ----> echo ciao
ciao
%
```

"Command substitution"

```
%echo ‘pwd‘ ----> echo /usr/roberto
/usr/roberto
%
```

"File name generation"

```
%rm *          ----> rm file1 file2
%
```

ORDINE DELLE MACROESPANSIONI

In **sh** ciascun tipo di macroespansione viene effettuato **una sola volta**, nella sequenza:

1. parameter substitution
2. command substitution
3. riconoscimento spazi e tokenizzazione
4. file name generation

Esempio:

```
%ls
a b
%a=ciao
%echo $*     ---> <stringa vuota>

                       e non:

                ---->  echo $a b
                ---> echo ciao b
                ---> ciao b
```

QUOTING

a). Totale: ´ . . . . ´

Toglie il significato speciale a tutti i metacaratteri (tranne ´ e *newline*)

N.B. Per quotare un solo carattere, basta premettere \

b). Parziale: " . . . . "

Vengono effettuate:

- sostituzione dei parametri
- sostituzione dei comandi

ma non:

- interpretazione dei separatori
- generazione dei nomi di files

Esempi:

```
$a=ciao
$echo  ‘pwd‘    $a       *
/usermail/roberto ciao file1 file2
$echo ’‘pwd‘    $a       *’
‘pwd‘    $a       *
$echo "‘pwd‘    $a       *"
/usermail/roberto    ciao       *
$echo \* \$a
* $a
$
```

ESEMPIO

```
$touch '$a'
$ls
$a                    <--- file di nome $a
$rm *
$
```

ESERCIZIO

```
% a=pwd
% b=$pwd
% c=‘pwd‘
% d="pwd"
% e=’pwd’
% echo $a $b $c $d $e
```

qual è il risultato?

```
% $a ; $b ; $c ; $d ; $e
```

qual è il risultato?

EXECUTION TRACE

Con l'opzione **sh -x**, la shell visualizza comandi e argomenti quando li esegue

Esempio:

```
$set 'date'
$echo $1
Sun
$sh -x
$set 'date'
+date
+set Sun Jun 4 18:49:25 MET DST 1995
$echo $1
+echo Sun
Sun
$
```

18. STRUTTURE DI CONTROLLO DI SHELL

STRUTTURE DI CONTROLLO DI sh

Sono disponibili le seguenti strutture di controllo:

`;`	sequenza
`if`	selezione
`case`	alternativa
`for`	iterazione enumerativa
`while`, `until`	iterazione

LA STRUTTURA `if`

```
$if      listacomandi1
>then listacomandi2
>elif listacomandi3   (opzionale)
>else listacomandi4   (opzionale)
>fi
```

Se l'ultimo comando di *`listacomandi1`* ha successo (`$?=0`), allora esegue *`listacomandi2`*, altrimenti...

Esempio:

```
$if rm file 2> out
>then echo rimosso!
>else echo errore!
>fi
rimosso!
$
```

LA STRUTTURA case

```
$case word in
>pattern1)   listacomandi1;;
>pattern2)   listacomandi2;;
>*)          listacomandin;;
>esac
```

pattern : *word*[|*word*|...]

confronta *word* con ogni *pattern*, nell'ordine in cui questi compaiono. Se c'è corrispondenza, viene eseguita la *listacomandi* relativa, quindi la esecuzione della struttura termina.

Il caso * è il caso di default

Esempio:

```
$a=tre
$case $a in
>uno ) echo 1 ;;
>due ) echo 2 ;;
>*     ) echo altro ;;
>esac
altro
$
```

ESEMPIO

Realizzare un comando **append**, tale che

append file	aggiunge il contenuto dello standard input al contenuto di **file**
append file1 file2	aggiunge il contenuto di **file1** al contenuto di **file2**

Allora:

```
case $# in
1) cat >> $1 ;;
2) cat >> $2 < $1 ;;
*) echo usage: append [from] to ;;
esac
```

LA STRUTTURA `for`

```
$for name [in word1 word2...]
>do listacomandi
>done
```

Alla variabile *name* sono assegnati a turno i valori `word1, word2,...` ogni volta che viene eseguita in ciclo la *listacomandi*

Esempio:

```
$for a in uno due tre
>do echo $a
>done
uno
due
tre
$
```

LA STRUTTURA `for` (SEGUE)

Se `in` *`word1 word2...`* è omesso, viene assunto `in $*`

Esempio:

`rimuovi`

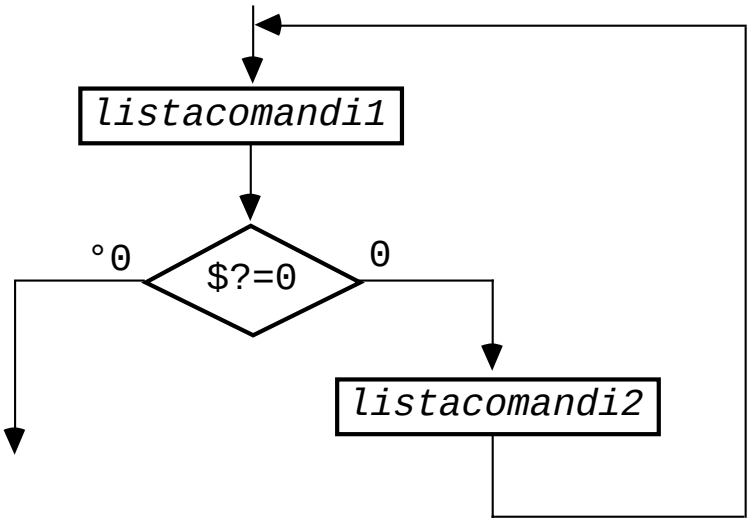
```
for a
do rm $a 2> out || echo $a non rimosso
done
```

```
$rimuovi uno due
uno non rimosso
due non rimosso
$
```

LE STRUTTURE `while` E `until`

```
$while listacomandi1
>do listacomandi2
>done
```

viene eseguita *`listacomandi1`*; se `$?=0`; viene eseguita *`listacomandi2`* e così via in ciclo




```
$until listacomandi1
>do listacomandi2
>done
```

Come *`while`*, ma la condizione di uscita è invertita

ESEMPIO

```
$compila source & aspetta obj &
$
```

aspetta:

```
until test -f $1
do sleep 60
done
echo $1 pronto
```

meglio:

```
$compila source && echo pronto &
11701
$
.....
$pronto
```

IL COMANDO `test`

`test` *`espressione`*

Valuta l’espressione e assegna all’exit code il valore zero se l'espressione è vera, non zero altrimenti

Molte possibilità per *`espressione`*

STRUTTURE DI CONTROLLO DI `csh`

Sono disponibili le seguenti strutture di controllo:

`;`	sequenza
`goto`	salto incondizionato
`if`	selezione
`switch`	alternativa
`foreach`	iterazione enumerativa
`while`, `repeat`	iterazione

19. COMANDI PER ACCESSO REMOTO

ALCUNI COMANDI DI ACCESSO REMOTO

`rlogin`	“remote login”
`telnet`	remote login per sistemi eterogenei
`rcp`	“remote copy”
`rdate`	“remote date”
`rsh`	“remote shell”
`rusers`	elenco utenti remoti
`finger`	informazioni sugli utenti della rete
`ping`	verifica se una macchina è attiva

...

N.B. Tutti gli esempi che seguono sono relativi a Unix System V R4

IL COMANDO `rlogin`

`rlogin` *`macchina`*

“remote login”

(È necessario avere un login valido sulla *`macchina`* remota, e gli opportuni diritto di accesso)

Esempio:

```
local$rlogin remote
remote$
...
remote$exit
Connection closed
local$
```

IL COMANDO `telnet`

`telnet` *`macchina`*

Simile a `rlogin`, permette di registrarsi anche in sistemi non Unix

Esempio:

```
$telnet remote
Trying 127.12.0.30
Connected to remote
Escape character is '^]'
remote login:
```

IL COMANDO `rcp`

`rcp` *`macchina1:pathname1 macchina2:pathname2`*

"remote copy"

Copia file da sistema locale a remoto, da remoto a locale, da remoto a remoto

Se *`macchina`* è omesso si intende la macchina locale

Esempio:

`local$:`**`rcp file remote:file`**

oppure, equivalentemente:

`local$:`**`rcp file remote:`**

IL COMANDO `rdate`

`rdate` *`macchina`*

“remote date”

Inizializza la data e ora locale a partire dalla data e ora di *`macchina.`*

È riservato al superuser

Esempio:

```
#rdate remote
Sat Apr 28 11:59:03 1995
#
```

(se ha successo, il comando visualizza data e ora aggiornate)

IL COMANDO `rsh`

`rsh` *`macchina`* [*`comando`*]

“remote shell”

Permette di connettersi a *`macchina`* e di eseguire lo specificato *`comando`*

(`rsh` copia il suo stdin in quello del comando remoto e copia stdout e stderr del comando remoto in stdout e stderr locali)

NB: “remote shell” e “reduced shell” si chiamano entrambi `rsh`, ma sono due comandi diversi (hanno due pathname diversi)

rsh: ESEMPI

`local$`**`rsh remote`**

come rlogin

`local$`**`rsh remote cat file`**

visualizza il `file` di `remote` in locale

`local$`**`cat temp|rsh remote wc > size`**

`temp` e `size` sono locali; `wc` è eseguito su `remote`

(ad esempio, per eseguire programmi di lunga durata su macchine scariche)

`local$`**`cat temp|rsh remote wc “>“size`**

“>“ indica che `size` è remoto

IL COMANDO `rusers`

`rusers` *`[opzioni]`* [*`macchina`*]

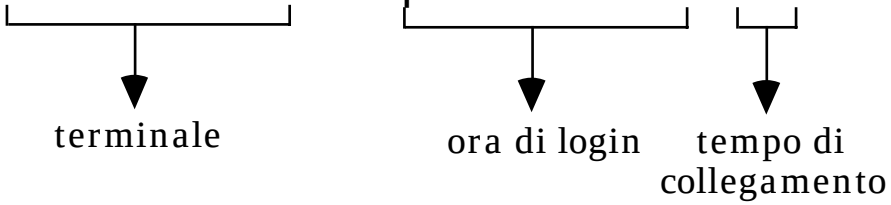
"remote users"

Dà l'elenco degli utenti collegati su *`macchina`*

(l'opzione `-a` elenca anche le macchine inattive)

Esempi:

```
local$rusers
local     giorgio
remote mario     luigi     root
local$rusers remote
remote mario  luigi    root
local$rusers -l local
giorgio local:console Apr28 10:35 35
local$
```



(Comando analogo: `rwho`)

IL COMANDO `finger`

finger [*opzioni*]
[*macchina*] [*user-id*[*@macchina*]]

Fornisce informazioni sugli utenti collegati in rete.

Per default, si riferisce alla macchina locale

Esempio:

```
local$ finger
Login   Name    TTY       Idle   When         Where
root    hgfj    console   58     Sat 10:45
mario   Mario   pts011            Sat 11:44   remote
local$
```

minuti da quando l'utente ha introdotto l'ultimo comando

nel caso di rlogin, indica la macchina da cui è stato fatto rlogin

finger: ESEMPIO

```
local$finger luigi@remote
Login name: luigi      in real life: Luigi Rossi
Directory: /home/giorgio  Shell: /usr/bin/ksh
On since Apr 28 11:44:50  on pts011
3 minutes 50 seconds Idle Time
New mail received Mon Apr 16 15:32:46 1995
unread since Sat Apr 28 11:44:19 1995
No plan
local$
```

IL COMANDO `ping`

`ping` *`macchina`*

Verifica se *`macchina`* è attiva

(le invia un messaggio: se è attiva, risponderà)

Esempio:

```
local$ping remote
remote is alive
$local
```

ACCESSO A FILE REMOTI: ALCUNI COMANDI

`share`	abilita altre macchine a montare proprie directory
`unshare`	rimuove una directory dall'elenco delle risorse condivise
`mount`	monta localmente una directory remota
`umount`	smonta una directory montata
`dfshares`	"distributed file shares" (lista le risorse disponibili a una specificata macchina)
`dfmounts`	"distributed file mounts" (lista le macchine che hanno montato le proprie risorse condivise)

IL COMANDO share

share [*opzioni*] *directory nome*

Abilita altre macchine a montare proprie directory.

È riservato al superuser

Esempi:

#**share -o ro /export mieifile**

dichiara condivisibile la directory/export, con nome mieifile, in sola lettura (only read-only)

#**share -o ro,rw=remote /export mieifile**

come sopra, ma la macchina remote ha accesso in read-write

#**share** elenca le directory condivise della macchina corrente

IL COMANDO unshare

unshare *directory*

Rimuove la *directory* dall'elenco delle risorse condivise.

È riservato al superuser

Esempio:

#**unshare /export**

(chi sta usando la risorsa da altre macchine riceverà un messaggio di errore)

IL COMANDO mount

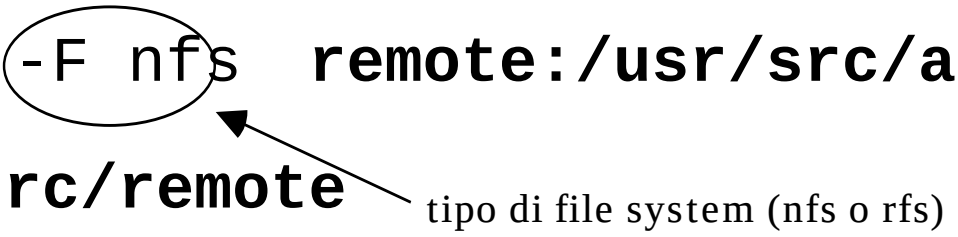
mount [*opzioni*] *macchina:directory directory*

Monta una directory remota su una directory locale vuota (“punto di mount”)

È riservato al superuser

Esempio:

```
#mkdir /usr/src/remote
#mount -F nfs  remote:/usr/src/acc
/usr/src/remote
#
```



IL COMANDO `umount`

`umount` *`directory`*

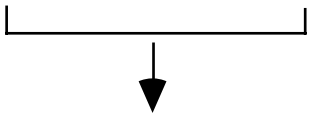
Smonta la *`directory`*

(Le directory montate rimangono disponibili fino ad esecuzione di `umount`, o fino a chiusura del sistema)

È riservato al superuser

Esempio:

#**`umount /usr/src/remote`**



pathname locale

IL COMANDO `dfshares`

`dfshares` [*`macchina`*]

“distributed file shares”

Elenca le risorse disponibili a *`macchina`*
(default: la macchina corrente)

Esempio:

```
local$dfshares
RESOURCE                    SERVER     ACCESS
                            TRANSPORT
local:/export               local      rw        tcp
remote:/usr/src/access      remote     ro        tcp
local$
```

dfmounts

“distributed file mounts”

Fornisce l’elenco delle macchine che hanno montato le proprie risorse condivise

(consente di conoscere l’utilizzazione delle proprie risorse, prima di revocarne la condivisione con unshare)

AUTOMOUNT

È un demone che:

- sorveglia le richieste di open per file compresi in un elenco di directory predefinito
- quando rileva un tentativo del genere, consulta un database per determinare come montare quel file, quindi
- esegue il comando **`mount`**

(solo in NFS, non in RFS)

2O. LA FILOSOFIA DI UNIX

I lucidi che seguono sono tratti da:

Mike Gancarz,

The Unix Philosophy,

Butterworth-Heinemann, 1995

“Un sistema operativo è una entità vivente.

Un sistema operativo incarna la filosofia dei suoi creatori.

Tutta la flosofia di Unix si basa sull’idea che l’utente sa che cosa sta facendo.”

Op.Cit.

I PRINCIPI

1. Piccolo è bello
2. Fai in modo che ogni programma faccia bene una sola cosa
3. Costruisci un prototipo appena possibile
4. Preferisci la portabilità all'efficienza
5. Memorizza i dati numerici in file ASCII
6. Usa il potere del software a tuo vantaggio
7. Usa script di shell per aumentare potere e portabilità
8. Evita le "captive user interfaces"
9. Fai di ogni programma un filtro

1. PICCOLO È BELLO

- I programmi piccoli sono facili da comprendere
- I programmi piccoli sono facili da manutenere
- I programmi piccoli consumano meno risorse di sistema
- I programmi piccoli sono più facili da combinare con altri strumenti

ESEMPIO

COPIA IL FILA A NEL FILE B

1. Chiedi all’utente il nome del file sorgente
2. Controlla che il file sorgente esista
3. Se non esiste, avverti l’utente
4. Chiedi all’utente il nome del file destinazione
5. Controlla se il file destinazione esiste
6. Se esiste, chiedi all’utente se desidera rimpiazzarlo
7. Apri il file sorgente
8. Informa l’utente se il file sorgente è vuoto, e in tal caso esci
9. Apri il file destinazione
10. **Copia i dati dal file sorgente al file destinazione**
11. Chiudi il file sorgente
12. Chiudi il file destinazione

2. FAI IN MODO CHE OGNI PROGRAMMA FACCIA BENE UNA SOLA COSA

Esempio:

```
$ls
aaaa        bbbb        cccc
dddd        eeee        ffff
gggg        hhhh
$
```

È corretto stampare l'elenco su più colonne ?

- Che cosa succede se il terminale ha più di 80 colonne ?
- Che cosa succede se il terminale stampa in caratteri proporzionali ?
- Che cosa succede se invece che 4 colonne ne preferisco 2 ?

3. COSTRUISCI UN PROTOTIPO APPENA POSSIBILE

- La prototipazione è un processo di apprendimento

- Costruire prototipi presto riduce i rischi

4. PREFERISCI LA PORTABILITÀ ALL'EFFICIENZA

- Fra qualche tempo l'hardware sarà più veloce
- Non perdere troppo tempo a migliorare la efficienza di un programma
- La soluzione più efficiente raramente è portabile
- Il software portabile riduce anche le esigenze di addestrare gli utenti
- I buoni programmi non muoiono mai - vengono portati su nuove piattaforme

5. MEMORIZZA I DATI NUMERICI IN FILE ASCII

- L’ ASCII è un formato di scambio comune
- Testi ASCII sono facili da leggere e da editare
- I file dati ASCII facilitano l’uso degli strumenti di elaborazione testi disponibili in Unix
- Una migliore portabilità compensa grandemente la perdita di efficienza
- I problemi di efficienza saranno comunque superati con la prossima versione dell’hardware

6. USA IL POTERE DEL SOFTWARE A TUO VANTAGGIO

- I buoni programmatori scrivono del buon codice; i programmatori eccellenti “prendono in prestito” del buon codice
- Evita la sindrome del NIH (“not invented here”)
- Incoraggia gli altri a usare il tuo codice per migliorare il proprio lavoro
- Automatizza tutto

7. USA SCRIPT DI SHELL PER AUMENTARE POTERE E PORTABILITÀ

- Gli script di shell permettono di potenziare enormemente il proprio lavoro
- Gli script di shell fanno anche risparmiare tempo
- Gli script di shell sono più portabili del C
- Resisti alla tentazione di riscrivere in C gli script di shell

8. EVITA “CAPTIVE USER INTERFACES”

- Le CUI assumono che l’utente sia umano
- Nelle CUI, i parser dei comandi sono spesso grandi e complessi da scrivere
- Le CUI tendono ad indurre un approccio “grande è bello”
- Programmi con CUI sono difficili da combinare con altri programmi
- Le CUI non sono scalabili
- Le CUI non permettono di trarre vantaggio dal software esistente

9. FAI DI OGNI PROGRAMMA UN FILTRO

- Ogni programma scritto dall'inizio dell'era dei calcolatori è un filtro
- I programmi non creano dati, le persone sì
- I computer convertono i dati da una forma all'altra

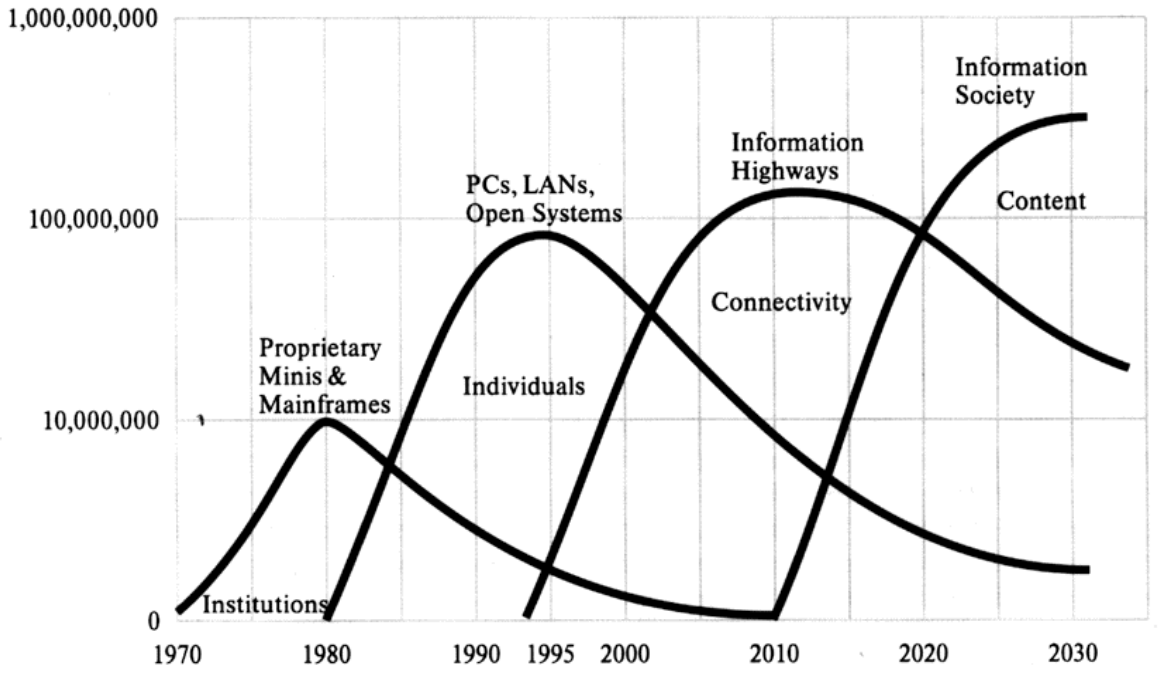
ALTRI PRINCIPI MINORI

1. Permetti all’utente di personalizzare il proprio ambiente
2. Costruisci kernel piccoli e leggeri
3. Usa le minuscole, e sii conciso
4. Risparmia gli alberi
5. Il silenzio è d’oro
6. Pensa in modo concorrente
7. La somma delle parti è più grande del tutto
8. Cerca la soluzione al 90%
9. Peggio è meglio
10. Pensa in modo gerarchico

“La filosofia di un secolo è il senso comune del successivo”

Proverbio su un dolcetto cinese

EVOLUZIONE DELL'INDUSTRIA DELL'INFORMATION TECHNOLOGY



fonte Rapporto EITO, 1995