Corso di Fondamenti di Informatica 2
CdL Ingegneria Informatica
Ing. Franco Zambonelli

IL SISTEMA OPERATIVO UNIX: IL FILE SYSTEM E LO SHELL

Lucidi Realizzati in Collaborazione con:

Prof. Letizia Leonardi
Università di Modena

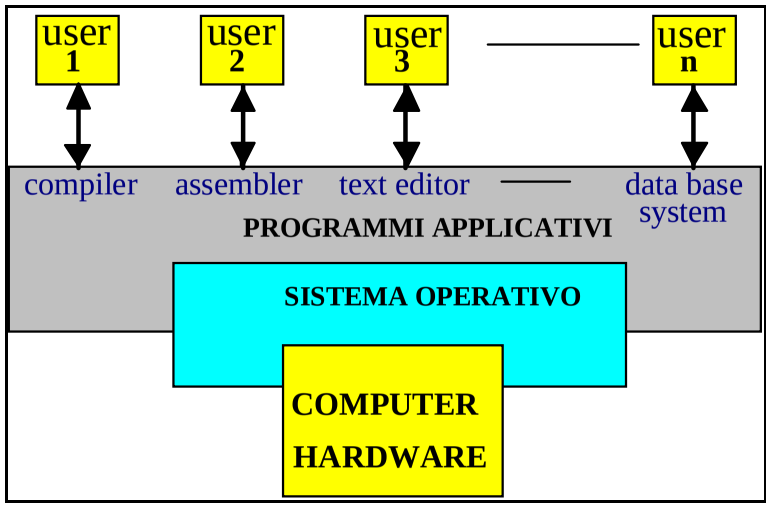
Prof. Antonio Corradi
Università di Bologna

Prof. Cesare Stefanelli
Università di Ferrara

IL SISTEMA OPERATIVO

Sistema Operativo come **gestore risorse** (per rendere efficiente l'uso delle risorse sia hardware che software)

Sistema Operativo come **macchina virtuale** (per semplificare l'uso del sistema di calcolo da parte degli utenti)



Gestione delle Risorse

Assegnazione delle risorse del sistema di calcolo ai programmi di utente in base al tipo di richiesta e agli obiettivi da raggiungere ***(uso efficiente delle risorse)***

Risorse Hw e Sw: tempo di CPU, memoria, I/O, etc..

* risoluzione di **conflitti** nell'uso delle risorse
* scelta dei **criteri** con cui assegnare una risorsa

Disponibilità di **apposite operazioni *(system call*)** per la gestione delle risorse

Macchina virtuale

Astrazione per semplificare l'uso delle risorse e nascondere i dettagli implementativi.

Esempio**: Controllore di un floppy disk**

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

⇓

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

Interfacciamento col Sistema

Il sistema operativo deve rendere disponibile all'utente un modo di visualizzare lo stato delle risorse del sistema, ed interagire con il sistema stesso e le sue risorse.

MS-DOS, UNIX: Tutte le risorse del sistema (contenuto delle memorie di massa e periferiche) vengono viste in termini di ***file***, e l'interazione utente-macchina virtuale avviene attraverso ***comandi*** sui file

Esempi:

- comando UNIX per cancellare un file

`RM PIPPO` (il comando RM ordina di cancellare il file di nome PIPPO)

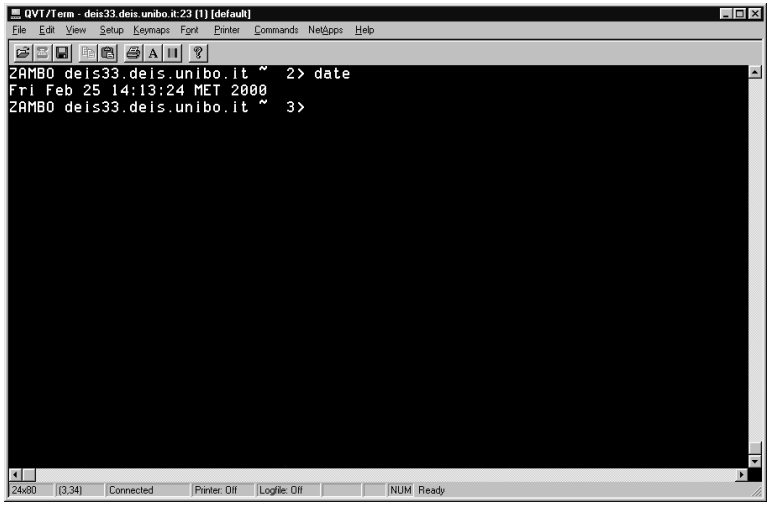
- comando per eseguire un programma applicativo di nome realizzato dall'utente (ad esempio, "prog_1.exe" e` il file prodotto dopo compilazione e linking):

`PROG_1` (esegui il file/programma di nome PROG_1)

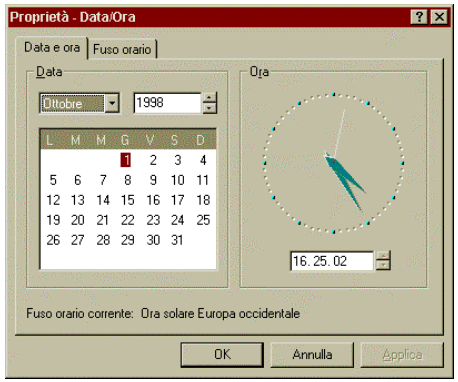
WINDOWS, MAC, OS-2: Rappresentazione grafica del sistema e delle sue risorse (concetto di ***scrivania "desktop" virtuale***). Le finestre e le icone rappresentano risorse o contenitori di risorse, come se fossero appoggiati sulla scrivania virtuale che è lo schermo. Attraverso il mouse e la tastiera, agiamo sulle risorse del sistema.

Esempio: VEDERE LA DATA

In UNIX:



In Windows:



UNIX: Linguaggio comandi

Le utilità del sistema operativo UNIX sono disponibili attraverso un linguaggio comandi direttamente interpretato da un **processore dei comandi**. Legge quello che noi scriviamo, interpretandolo come comando e, se il comando è corretto, lo esegue

I processori dei comandi sono evoluti da semplici riconoscitori di comandi a supporti per ambienti più complessi, per l'esecuzione di veri e propri linguaggi di programmazione.

Il Concetto di Prompt

- In un ambiente interattivo non grafic il **prompt** è il simbolo (ad esempio, una sequenza di caratteri) che viene mostrato all'utente per indicare che si attende l'immissione di un comando.

IN MSDOS:

```
nomedrive:\>
```

IN UNIX

```
$
```

oppure

```
nomedirettorio>
```


Evoluzione di UNIX e Personal Systems

- **1965 Multics** (Bell Telephone)
- **1969 UNIX** per PDP (Thompson & Ritchie, Turing Award)
- **1973** Definizione **Linguaggio C** e Riscrittura UNIX in C
- **1977 UNIX** Disponibile per Piattaforme NON PDP
- **1980, MS-DOS 1.0**:La prima versione del sistema operativo MS-DOS era ispirata a UNIX una delle prime realizzazioni di sistema operativo per microcomputer.
- **1984: Apple Mac –** Introduzione Interfaccia grafica e sistema a finestre
- **1988: Windows:** nasce come interfaccia grafica di supporto al sistema operativo MS-DOS
- **1993: Windows NT:** integra la gestione della multi-utenza
- **1999: Windows 98 e 2000:** realizza il concetto di sistema operativo inteso come strumento per la navigazione in uno spazio di dati, comunque distribuito –> **INTERNET**

E il Futuro ?

- Esecuzione **applicazioni distribuite** e recupero **on-demand** di software (adesso Java applet)
- Micro-sistemi operativi per personal digital systems
- Network Television, Networked House
- Interfaccia vocale e "touch screen".

UNIX E FILES

Tutto il finzionamento del sistema operativo UNIX è CENTRATO SUL FILE SYSTEM

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

- FILE SYSTEM **gerarchico**
  ALBERO di sottodirettori

- OMOGENEITÀ **dispositivi** e **file**
  TUTTO è file

FILE

astrazione unificante del **sistema operativo**

file ordinari

file direttori, accesso ad altri file

file speciali (dispositivi fisici), contenuti nel direttorio /dev

ORGANIZZAZIONE del FILE SYSTEM

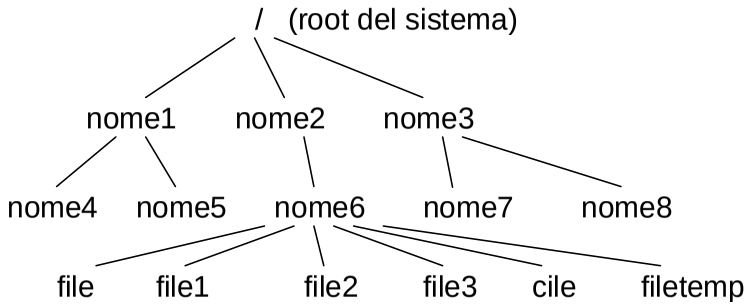
NOMI di file
- **ASSOLUTI**: dalla radice
  /nome2/nome6/file
- **RELATIVI**: dal direttorio corrente
  nome6/file

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

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

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

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



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

Accesso al Sistema Operativo Unix

Ingresso (LOGIN)

Username: franco
Password: **********

L'accesso viene verificato al login (autenticazione utente)

Uscita **(LOGOUT)**

L'uscita avviene solo al logout, tramite uno dei comandi:

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

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

La shell non è unica: un sistema può metterne a disposizione varie (e altre possono essere aggiunte)

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

La shell associata a ogni utente è specificata all'interno del file **/etc/passwd**, che costituisce il "database" degli utenti registrati su quel sistema (con tutte le loro caratteristiche).

FORMATO DEL FILE /etc/passwd

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

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

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

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

commento ➔ campo di commento

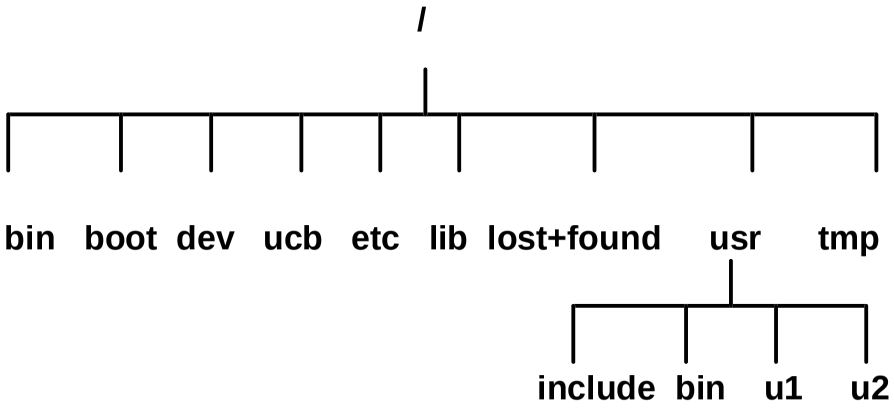
direttorio ➔ direttorio iniziale in cui si trova l'utente al login (home directory)

comando ➔ comando che viene eseguito automaticamente all'atto del login **(in genere è una shell)**

ESEMPIO:

root:.XPc4HKe0nPQA:0:1:Operator:/:/bin/sh
franco:rTIW65BOuQ9ng:230:30:Franco Zambonelli:/home/franco:/bin/sh

STRUTTURA DEL FILE SYSTEM



bin comandi principali di sistema

dev i dispositivi

etc, file significativi per il sistema,
ad es. **passwd** che memorizza gli utenti autorizzati

lib, le librerie di sistema

tmp, file temporanei

usr, sottodirettorio per utenti

Esiste un utente, **root**, privilegiato rispetto agli altri:
il gestore del sistema (detto anche super-user) che sfugge alle regole di sicurezza

PROTEZIONE dei File

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

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

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

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

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

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

i primi 9 ==> triple di permessi

SIGNIFICATO DEI 12 BIT

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

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

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

dodicesimo bit
SUID (Set-User-ID) (identificatore di utente effettivo)
Si applica a un file di **programma eseguibile**
Se vale 1, fa sì che *l'utente che sta eseguendo* quel programma venga considerato *il proprietario* di quel file (solo per la durata della esecuzione)

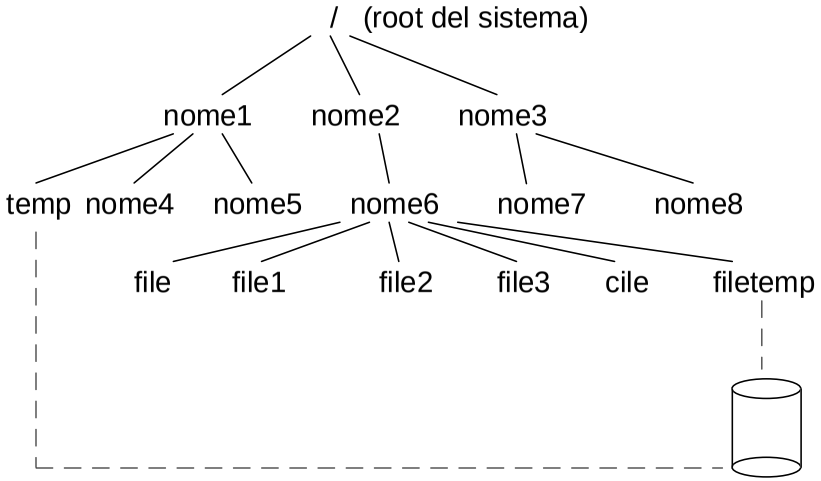
Questo bit è necessario per consentire, durante l'esecuzione di quel programma, operazioni di lettura/scrittura su file di sistema, che l'utente non avrebbe il diritto di leggere / modificare.

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

SGID bit: come SUID bit, per il gruppo
sticky bit
il sistema cerca di mantenere in memoria l'immagine del programma, anche se non è in esecuzione

Collegamenti (Link)

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



Il sistema considera e tratta il tutto:
- se un file viene cancellato, le informazioni sono veramente eliminate solo se non ci sono altri link a esso
- Il link cambia i diritti? ➔ Meglio di no

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

comando: ln [-s]

ln /nome2/nome6/filetemp /nome1/temp

Si vedano i link di ogni direttorio

SHELL ovvero
il **PROCESSORE COMANDI**

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

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

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

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

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

COMANDI RELATIVI AL FILE SYSTEM

Sintassi:

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

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

CREAZIONE / GESTIONE DI DIRETTORI

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

TRATTAMENTO FILE

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

Esempi di comandi:

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

PROTEZIONE e DIRITTI SUI FILE

Per variare i bit di protezione:

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

I permessi possono essere concessi o negati dal
proprietario del file

Esempio di variazione dei bit di protezione:

chmod 0755 /usr/dir/file

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

Ad esempio:
chmod u-w fileimportante

Altri comandi:
chown <nomeutente> <nomefile>
chgrp <nomegruppo> <nomefile>

Esempio, il comando ls per listare il contenuto di una directory

Varie opzioni:

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

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

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

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

ls -d [<nomedir>]
i direttori sono visualizzati come i file, senza entrare nel contenuto e listarlo

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

ls -i [<nomedir>]
lista gli i-number dei file con le altre informazioni

ls -r [<nomedir>]
lista i file in ordine opposto al normale ordine alfabetico

ls -t [<nomedir>]
lista i file in ordine di ultima modifica, dai più recenti, fino ai meno recenti

ALTRI COMANDI DI SISTEMA

- I *file* sono considerati insiemi di *linee*, fatte da *parole*
- Le *parole* sono sequenze di *caratteri*, separate da spazi

more <nomefile> *una pagina per volta*
sort <nomefile> *ordina alfabeticamente*
sort [**-r**] <nomefile> *(r = in ordine inverso)*
Molte opzioni:
- uscita su file opzione **-o** <nomefileout>

diff <file1> <file2> *mostra solo le righe diverse*
find <direttorio> **-name** <nomefile> **-print**
cerca nomefile *in* direttorio
wc [**-lwc**] [<nomefile>]
*conta le linee (opzione **l**), o le parole (opzione **w**), o i caratteri (opzione **c**) dello standard input o del file*

COMANDI di STATO del sistema

date *data e ora attuale*
time *cronometra l'esecuzione di un comando*
who *mostra gli utenti attualmente collegati*
ps *mostra i processi correnti nel sistema*
sleep *sospende il processo quando specificato*
man <comando> *mostra l'help (guida) del comando*

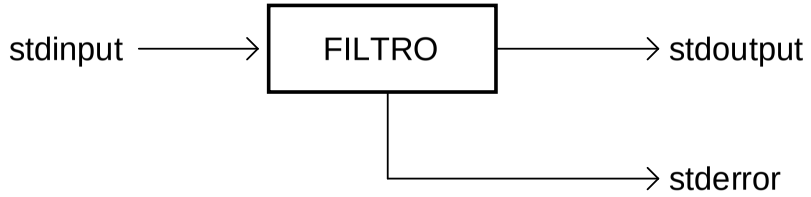
ESEMPIO: date; time; who; ps

RIDIREZIONE DELL'I/O

Tutti i comandi di UNIX sono **filtri**

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

standard input ➔ console
standard output ➔ console
standard error ➔ console



I filtri possono operare sull'input considerandolo a linee

more *stampa una pagina per volta*
sort *ordina alfabeticamente le linee*
grep <stringa> *cerca la* stringa *nel file*
rev *ribalta ogni singola linea*
tee <file> *copia l'input sia sull'output sia sul* file
head [-numerolinee] *filtra le prime linee del file*
tail [-numerolinee] *filtra le ultime linee del file*
awk *trasforma le stringhe di un file secondo certe regole*

Concetto di filtraggio e ridirezione FONDAMENTALE per la **composizione** di programmi di trattamento dati in UNIX.

Ogni comando può essere ridiretto su un file diverso **senza cambiare il comando**

- **completa OMOGENEITÀ fra dispositivi e files**

ridirezione dell'input

<comando> **<** <fileinput>

ridirezione output

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

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

Alcuni esempi di ridirezione:

ls -lga > file

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

who > temp
wc -l temp

ps > file

RIDIREZIONE (estensioni)

Ogni comando trova aperti

stdin, stdout, stderr

RIDIREZIONE MULTIPLA

In caso di ridirezione, il file di ingresso è aperto in lettura, il file di output aperto in scrittura ***(cioè creato vuoto)***

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

EFFETTI COLLATERALI distruzione dei file file1 e file3

ALTRI FILE

In Bourne Shell è possibile anche:

- usare altri file in ridirezione, specificandoli alla invocazione del comando
- agganciare più output ad uno stesso file

comando **2>** file2 **3>** file3 **5>** file5
Si richiede la apertura del file2 con chiave 2, del file 3 con chiave 3, etc.

comando &> 2
L'uscita del comando viene forzata sullo std error

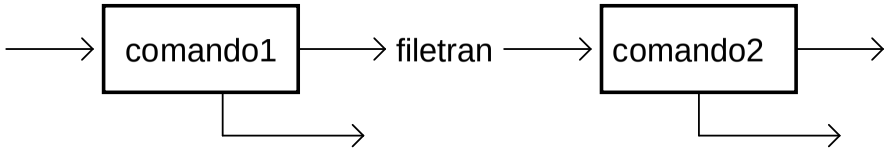
Si noti che la numerazione prosegue dai file standard:
stdin = 0, stdout = 1, stderr = 2, ...

PIPE

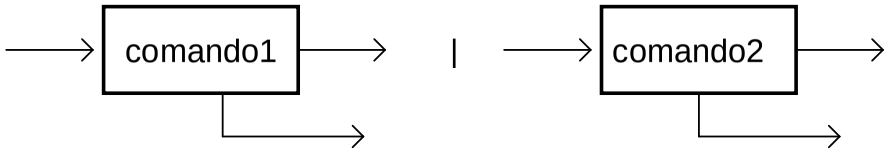
COLLEGAMENTO AUTOMATICO di comandi

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

SCHEMA IMPLEMENTATIVO



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



<comando1> **|** <comando2>

In **DOS** ➔ REALIZZAZIONE con **FILE TEMPORANEI**

In **UNIX** ➔ PIPE come **COSTRUTTO PARALLELO**

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

I comandi della pipe procedono in parallelo tra loro
➔ NON si creano file temporanei
Mentre un comando produce l'output, l'altro lo consuma

Esempi di piping:

who | wc -l *conta gli utenti collegati*

ls -R | more

rev < file1 | sort | rev | more

provare le differenze tra le pipe in Unix e in DOS

Definizione di processo

Informalmente, il termine processo viene usato per indicare un **programma in esecuzione**

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

Differenza tra processo e programma

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

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

In un sistema multitasking **più** processi possono essere in esecuzione ***"concorrentemente"***

RIENTRANZA DELLO SHELL

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

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

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

Le invocazioni attivano un processo che esegue lo shell

Gli shell sono RIENTRANTI:

più processi possono condividere il codice senza errori ed interferenze

```
sh
sh
csh
ps      # quanti processi si vedono?
```

METACARATTERI

Lo SHELL riconosce caratteri speciali (WILD CARD)

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

? un qualunque carattere in un nome di file

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

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

\# commento fino alla fine della linea
**** escape (segnala di *non* interpretare il carattere successivo come speciale)

Il comando **echo** scrive la stringa successiva
echo * *stampa tutti i nomi di file del direttorio corrente*
echo * *stampa il carattere asterisco '*'*

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

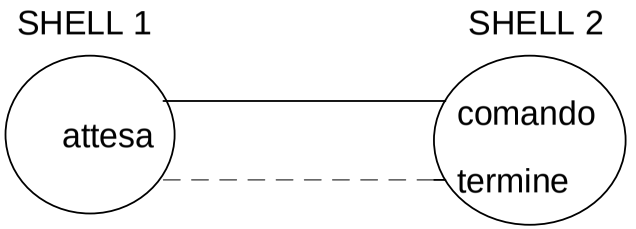
ESECUZIONE di COMANDI in SHELL

In UNIX ogni comando è eseguito da una nuova shell

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

La **seconda shell**
- esegue le sostituzioni dei metacaratteri e dei parametri
- cerca il comando
- esegue il comando

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



AMBIENTE DI SHELL (environment):
insieme di variabili di shell, per esempio:

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

SCHEMA di un PROCESSORE COMANDI

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

Per abortire il comando corrente: CTRL-C

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

Variabili nella shell

Ogni shell definisce:

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

nomevariabile=$nomevariabile
l-value *r-value*

Esempi:

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

Sostituzioni della shell (parsing)

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

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

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

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

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

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

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

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

Sostituzioni della shell

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

Riassunto fasi

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

Comandi relativi al controllo dell'espansione:

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

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

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

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

Sostituzioni della shell: Esempi

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

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

Esempi:
$ **es='??'**
$ **$es**
tt: execute permission denied
$

shell esegue fasi 1, 2 (sostituzione di es con ??), 3 (sostituzione di ?? con il file del dir corrente tt) e prova quindi ad eseguire tt che non ha pero' i diritti di esecuzione

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

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

ATTENZIONE:

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

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

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

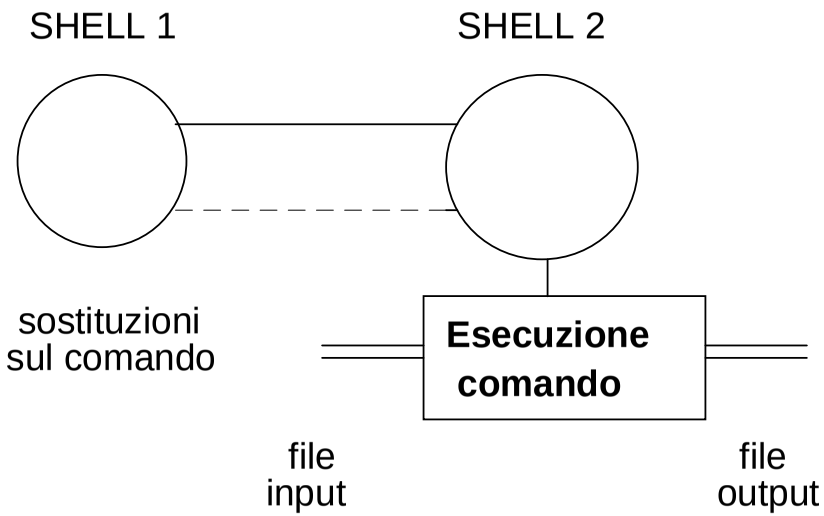
```
eval echo $x     # stampa 3 (perché valuta $y)
```

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

Esempio:

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

Esecuzione di un COMANDO



Il comando **esegue** avendo collegato
- il proprio input al *file di input*
- il proprio output al *file di output*

specificati dalla **shell di lancio**

ESECUZIONE IN PARALLELO &

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

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

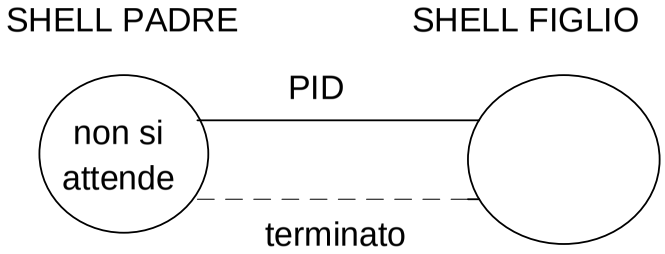
Lo shell invocante è immediatamente attivo

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

I processi in background si eliminano con
kill <PID>

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

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



PROGRAMMAZIONE NELLO SHELL

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

Linguaggio Comandi

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

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

SHELL DI BOURNE (/bin/sh)

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

Lo shell è **case sensitive**

Per fare eseguire un file comandi (myFile):

- o si rende eseguibile il file e poi lo si lancia:

  chmod +x myFile ; myFile

- oppure si può invocare direttamente una shell che lo esegua:

  sh myFile

VARIABILI

Sono disponibili **variabili** che contengono stringhe
NON è necessaria la definizione delle variabili

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

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

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

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

Assegnamento

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

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

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

Le variabili sono trattate come stringhe di caratteri

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

PASSAGGIO PARAMETRI

comando argomento1 argomento2 ... **argomentoN**

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

- **$0** rappresenta il comando stesso
- **$1** rappresenta il primo argomento
- ....

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

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

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

COMANDI CORRELATI
È possibile far scorrere tutti gli argomenti ➔ **shift**
- $0 non va perso, solo gli altri sono spostati ($1 perso)

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

È possibile riassegnare gli argomenti ➔ **set**

set exp1 exp2 exp3 ...

- gli argomenti sono assegnati secondo la posizione

ALTRE VARIABILI

Oltre agli argomenti di invocazione del comando

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

INPUT/OUTPUT

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

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

ATTENZIONE:
- la shell NON supporta direttamente **array**
- **ma consente di simularli** "in qualche modo"

INPUT:

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

OUTPUT:

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

STRUTTURE DI CONTROLLO

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

Tale valore di uscita è posto nella variabile ?

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

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

ESEMPIO 1

cp a.com b.com

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

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

cp: cannot access a.com

$ **echo $?**

2

ESEMPIO 2

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

---

test

Comando per la **valutazione di una espressione**

test -<opzioni> <nomefile>

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

ATTENZIONE: convenzione opposta rispetto al C!

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

TIPI DI TEST POSSIBILI

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

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

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

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

TIPI DI TEST / segue

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

Espressioni booleane

!	not
-a	and
-o	or

---

expr

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

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

ESEMPIO:

expr 24 - 12 + 65 # *fornisce 77*

i=12; j=$i+1; echo $j ➔ 12 + 1
expr $j; ➔ 13
expr $i + 1 -16 ➔ -3

STRUTTURE DI CONTROLLO

ALTERNATIVA

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

ATTENZIONE:

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

Esempio

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

Esempio di invocazione:

fileinutile si 12 ➔ *stampa* si

Esempio (controllo numero argomenti)

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

Esempio

file leggiemostra (uso: leggiemostra filename)

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


ALTERNATIVA MULTIPLA

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

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

ESEMPI:

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

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

```
#Ancora controllo dei parametri
case $# in
  0) echo Usage is: $0 file etc
     exit 2;; # si esce
  *) ;;
esac
```

RIPETIZIONI ENUMERATIVE

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

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

ESEMPIO

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

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

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

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

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

NB: il comando >file crea il file “file” di 0 byte

RIPETIZIONI NON ENUMERATIVE

while <lista-comandi>
  do
    <comandi>
  done

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

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

until <lista-comandi>
  do
    <comandi>
  done

Come while, ma inverte la condizione

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

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

AMBIENTE di un FILE COMANDI

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

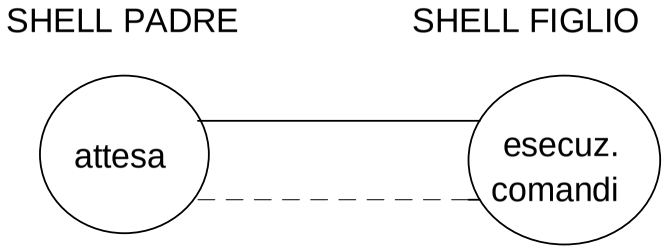
L'ambiente è accessibile alle ***shell figlie***, che possono accrescere il tutto e aggiungere nuove variabili

- **viene passato solo l'ambiente predefinito iniziale**
- **sono aggiunti i valori delle sole variabili esportate**

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

L'ambiente padre è preservato

OGNI COMANDO è ESEGUITO da una NUOVA SHELL, distinta dalla shell-padre
La shell-figlio ottiene una copia (parziale) dell'ambiente del padre *ma non può modificarla*



IL TRATTAMENTO DEI SEGNALI

Un programma Shell tiene anche conto di EVENTI ASICRONI

I segnali UNIX sono **eventi** a cui si può **associare un gestore**
Lo statement shell
trap comandi numerosegnale

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

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

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

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

Per INVIARE un segnale a un processo:
KILL –numero_del_segnale pid_del_processo

Le azioni conseguenti sono

ignorare

```
    trap ' ' 1              # o anche
```

azione utente

```
    trap 'ls; exit' 15    # da kill
    trap 'ls' 15
```

default: TERMINAZIONE

```
    trap 2
```

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

d=`pwd`
for i in *
do
  if test -d $d/$i
    then
      cd $d/$i
      while echo  $i: direttorio
        trap exit 2 # si esce dal comando se interrupt
        read x      # ad es. CTRL-D per uscire dal while
      do
        trap '  ' 2   # ignorati
        eval $x
      done
      scan          # chiamata ricorsiva
  fi
done
```


ESEMPI

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

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

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

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

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

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

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

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

FILE COMANDI

la parte di controllo degli argomenti è fondamentale

È necessario **verificare gli argomenti**
- devono essere **innanzitutto *nel numero giusto***
- **poi *del tipo richiesto***

```
# invocazione  di comando per ...
case $# in
0|1|2|3|4)    echo Errore. Almeno 4 argomenti ... &> 2
              exit 1;;
esac
#
# in caso di argomenti corretti:
# primo argomento: dev'essere un direttorio (o file)
      if test ! -d $1
        then echo argomento sbagliato: $1 direttorio; exit 2
      fi;;
#
# primo argomento: se è un nome di file assoluto
case $1 in
      /*)  if test ! -f $1
                then echo argomento sbagliato: $1 file &> 2; exit 2
           fi;;
       * ) echo argomento sbagliato: $1 assoluto; exit 3;;
 esac
#
# primo argomento: se è un nome di file relativo
case $1 in
    */*) echo argomento sbagliato: $1 nome relativo; exit 3;;
     * ) ;;
esac
```

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

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

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

Si possono eliminare alcuni argomenti per comodità di scansione: si usi lo shift, dopo avere salvato gli argomenti che vengono eliminati

```
   salva1=$1  # salvataggio di $1
   shift
   salva2=$1  # salvataggio di $2
   shift
   # cosa vale adesso $*?
   for i
   do
   # il ciclo è fatto per tutti gli argomenti esclusi quelli tolti
   done
# gli argomenti iniziali sono dati da $salva1 $salva2 $*
```