SISTEMA OPERATIVO UNIX

Sistema Operativo progettato e sviluppato presso i **Bell Laboratories dell'AT&T**

1969 ⇨ Ken **Thompson** implementa ***UNICS*** (UNiplexed Information and Computing Service) su un minicomputer **PDP-7**, interamente in *ASSEMBLER*

➔ il S.O. era ***monoprogrammato (inizialmente!)***

➔ in seguito UNICS viene chiamato **UNIX**

Brian **Kernighan** e Dennis **Ritche** si uniscono a Thompson

➔ UNIX viene portato su vari modelli di **PDP-11**

⇨ la maggior parte del codice viene riscritto inventando il **linguaggio C** ➔ questo aumenta la sua PORTABILITÀ

➔ viene introdotta la ***multiprogrammazione***

Dall'inizio degli anni '80, esistono due versioni di UNIX:

✶ una sviluppata dallo UNIX Support Group della AT&T
➔ **UNIX System V** (1983)

✶ una sviluppata dalla Università di Berkeley
➔ **UNIX BSD** (Berkeley Software Distribution) (1978)

➲ tentativo di *STANDARDIZZAZIONE*: **POSIX** (Portable Operating System unIX)

CARATTERISTICHE:

- sistema operativo ***multiutente, multiprogrammato*** e ***multiprocesso*** (multitasking)
- memoria virtuale

ORGANIZZAZIONE di UNIX: livelli ➔ poco stratificato

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

ACCESSO A UNIX

⇨ sistema multi-utente

L'accesso al sistema per una sessione interattiva viene verificato durante la fase di **LOGIN (LOG-IN)**

Username: pippo ← nome utente
Password: ******** ← parola-chiave (non viene fatto l'eco di nessun carattere, oppure viene fatto l'eco di un carattere standard, ad esempio *)

Il sistema di sicurezza si realizza mediante l'uso di password
➔ **fase di autenticazione**

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

La fine di una sessione interattiva avviene durante la fase di **LOGOUT (LOG-OUT)** mediante **il comando exit o premendo contemporaneamente CTRL-D** (eof-tastiera!)

In una sessione interattiva, una **shell (interfaccia comandi)** rimane in attesa dei comandi immessi dall'utente (e quindi non termina fino a che l'utente non decide di fare il logout): dato che esistono normalmente in UNIX diverse shell che possono essere eseguite, quale mettere in esecuzione per uno specifico utente è indicato all'interno del file di sistema **/etc/passwd** *(vedi nel seguito)*

Nel caso dei laboratori didattici (LICA e INFOMEC), la fase di autenticazione coinvolge un server LDAP che gestisce in modo centralizzato gli username e le password degli studenti e dei docenti
Lo username che uno studente deve usare è il numero presente nell'indirizzo di posta d'Ateneo e la password è quella della posta e di ESSE3!

FILE SYSTEM

- FILE COME **STREAM DI BYTE**
  ➔ NON sono introdotte organizzazioni logiche o accessi a record

- FILE SYSTEM **gerarchico con possibilità di link**
  ➔ GRAFO di sottodirectory

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

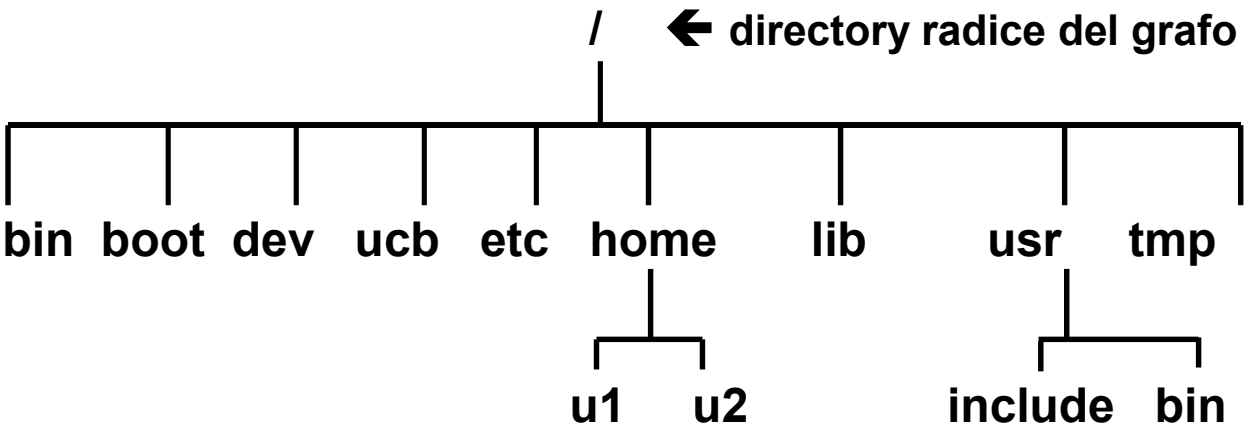
FILE
⇨ astrazione unificante del sistema operativo

- file ordinari

- file directory
  accesso ad altri file

- file speciali (dispositivi fisici)
  contenuti nella directory **/dev**

STRUTTURA TIPICA DEL FILE SYSTEM



bin, comandi principali del sistema
per esempio, il comando **cat** che serve per visualizzare il contenuto di un file di testo

dev, i dispositivi

etc, file significativi per il sistema,
per esempio, il file **passwd** che memorizza gli utenti autorizzati (quindi /etc/passwd)

home, directory degli utenti

lib, le librerie di sistema

tmp, file temporanei

usr, per librerie o comandi specifici per gli utenti

Riprendiamo a questo punto la **fase di autenticazione** per capire che controlli vengono fatti a livello sistemistico …

FORMATO DEL FILE /etc/passwd

utente:password:UID:GID:commento:directory:comando

1) **utente** ➔ nome che bisogna dare al login
2) **password** ➔ password che occorre dare al login
3) **UID** ➔ **User IDentifier**
   numero che identifica in modo univoco l'utente nel sistema
4) **GID** ➔ **Group IDentifier**
   numero che identifica il gruppo di cui fa parte l'utente
5) **commento** ➔ campo di commento
6) **directory** ➔ directory iniziale (in forma assoluta) in cui si trova l'utente al login (***home directory***)
7) **comando** ➔ comando che viene eseguito automaticamente all'atto del login
   N. B.: in genere è una shell

ESEMPIO: nelle vecchie versioni di UNIX la password era memorizzata in forma crittografata nel file /etc/passwd (che è visualizzabile con il comando **cat** da tutti gli utenti)

root:.XPc4HKe0nPQA:0:1:Operator:/:/bin/sh
letizia:rTIW65BOuQ9ng:120:20:Letizia Leonardi:/home/letizia:/bin/csh
pippo::121:20:Utente Generico:/home/p:/bin/ksh

Nota bene: di solito il nome relativo semplice della home directory è uguale al nome di utente, ma questo non è un vincolo: ad esempio, l'utente pippo ha come nome relativo semplice della sua home p (e non pippo!) ⇨ rende solo più "leggibile" la struttura delle sottodirectory di utente

FILE /etc/passwd (segue)

Nelle versioni più recenti di Unix/Linux, le password (sempre in forma crittografata) sono memorizzate in un file diverso (**/etc/shadow**) che non è leggibile dai normali utenti
➔ questo approccio garantisce una maggiore sicurezza

Il comando (dato in forma assoluta, che si trova nell'ultimo campo) che viene eseguito come ultima azione al termine dell'autenticazione NON è detto che sia una shell
➔ si può anche avere il caso che il comando indicato e quindi eseguito invece termini ➔ per verificarlo …

Prova di login con username "particolare":
il sistema, al termine della fase di autenticazione, esegue (invece che una shell) un programma che stampa un messaggio di saluto e poi termina!

IMPORTANTE:
Esiste un utente, con username **root**, privilegiato rispetto agli altri: corrisponde al gestore del sistema che "sfugge" alle regole di protezione ⇨ ***SUPERUTENTE***

Ad esempio, tale utente può visualizzare (con il comando cat) il file /etc/shadow

*Alcune distribuzioni Linux (come UBUNTU, DEBIAN e derivate), non consentono l'accesso diretto con lo username root, ma usano il comando **sudo** (SUPERUSER DO) per poter fare eseguire da alcuni utenti privilegiati (detti **sudoers**) i comandi come superutente!*

OSSERVAZIONE IMPORTANTISSIMA: tutto quello che segue fa riferimento all'uso di UNIX/Linux utilizzando una ***interfaccia testuale*** (che chiameremo ***terminale***); nel caso di uso di una interfaccia grafica bisognerà applicare i comandi illustrati in una specifica finestra (di solito indicata con il nome TERMINALE!)

SHELL (PROCESSORE COMANDI)

In condizioni normali, se la fase di autenticazione ha successo, l'utente può iniziare una sessione di lavoro tramite l'uso di una **shell**

La shell è un **processore comandi**, che:
1) **accetta** un comando* richiesto dall'utente
2) lo **interpreta**
3) e (*se l'interpretazione è riuscita*), esegue le sostituzioni, ricerca il comando, realizza ridirezioni/piping e lo **esegue**

I **comandi** sono:
- inseriti da un terminale (al *prompt* dei comandi) e quindi la shell legge la linea appena l'utente ha premuto il tasto INVIO/RETURN (detto anche **CR** cioè Carriage Return, ritorno a capo del cursore del video).
- oppure sono prelevati da una linea di un file (detto appunto file comandi o script)

La shell esegue le fasi 1-2-3) in un **ciclo** che ha termine solo o all'esecuzione del comando **exit** (o CTRL-D) se da terminale o fino alla fine del file comandi

IMPORTANTE: La shell presenta una importante particolarità rispetto ad un semplice esecutore di un comando alla volta ➔ **creazione** (*a parte eccezioni sotto indicate*) di un **processo** (sotto-shell) per la fase 3)

ECCEZIONI: alcuni comandi sono eseguiti direttamente dalla shell ➔ sono i comandi interni (detti anche **built-in**)
Esempi (vedi lucido 47): **exit** ➔ per terminare
cd dir ➔ per cambiare directory

N.B. Per abortire l'esecuzione di un comando, usare la combinazione di tasti **CTRL-C**!

* **NOTA BENE**: per comando si intende sia un comando definito dal sistema, sia un programma eseguibile sviluppato da un utente e sia uno script (file comandi)

SHELL (segue)

In generale la sintassi di un comando è (**case sensitive!**):
comando [-opzioni] [argomenti] <CR>*
Possiamo anche inserire più comandi sulla stessa linea, ma vanno separati con il metacarattere **;**

IMPORTANTE: esistono diverse shell che gli utenti possono utilizzare ➔ Bourne Shell (SH praticamente la prima shell di UNIX), **Bourne Again Shell** (BASH), **C shell, Korn shell**, **Almquist shell** (ASH), **Debian Almquist Shell** (DASH) ➔ spiegheremo la **Bourne shell** perché più semplice!

Le diverse shell si differenziano per aspetti visibili all'utente e altri non visibili ➔ In particolare, a livello visibile le shell si differenziano per la *sintassi* che sono in grado di interpretare (in particolare nei file comandi/script)

Il sistemista di un sistema UNIX/Linux assegna ad ogni utente una delle shell disponibili, ma se all'utente non piace e ne vuole usare un'altra basta che la invochi come un normale comando (senza indicare alcun parametro)

PRECISAZIONE: se ad una shell viene passato un file comandi, la shell esegue il file comandi e poi termina

OPZIONE IMPORTANTE PER LE SHELL: -x xtrace

Write each command to standard error (preceded by a '+ ') before it is executed. Useful for debugging

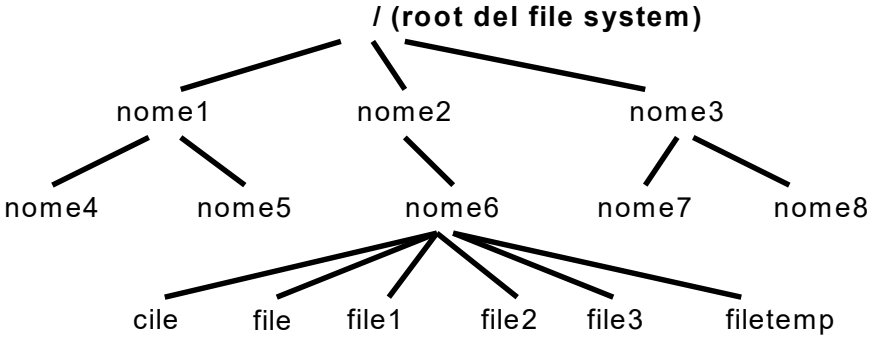
ESEMPIO:

```
$ sh
$ bash
$ ps          ➔ quanti processi si vedono?
```

N.B. 1) **$** è il prompt dei comandi (che può anche essere personalizzato); 2) il comando **ps** mostra le informazioni sui processi correnti associati al terminale; 3) per conoscere cosa fa esattamente un comando e quali opzioni prevede si può usare il comando **man** (che sta per *manual*)

* Le parentesi quadre indicano elementi che, a livello sintattico, sono opzionali! Gli argomenti sono detti anche parametri

ORGANIZZAZIONE del FILE SYSTEM



NOMI di file/directory

- **ASSOLUTI**: dalla radice
  ES: file: /nome2/nome6/file
- **RELATIVI**: dalla directory corrente
  ES: dir. corrente: /nome2 file: nome6/file
- **RELATIVI SEMPLICI**: dalla directory corrente
  ES: dir. corrente: /nome2/nome6 file: file

I nomi sono **case-sensitive** e possono essere lunghi a piacere (in genere il limite è 255 caratteri)

Directory corrente identificata da **.**
Directory padre della directory corrente identificata da **..**

Ogni utente ha una directory di default, in cui l'utente si trova all'ingresso nel sistema (***home directory***)

Si possono usare abbreviazioni nei nomi usando dei caratteri speciali ➔ detti anche ***WILDCARD*** che sono dei **metacaratteri** ➔ I principali, trattati nella fase 3-sostituzione, sono (nel seguito vedremo anche il caso di [ ]):

***** fa match con qualunque stringa (anche vuota)
? fa match con un qualunque carattere (ma almeno uno)

Ad esempio:

`$ echo file*` ➔ file, file1, file2, file3, filetemp
`$ echo file?` ➔ file1, file2, file3
`$ echo *ile` ➔ cile, file

N.B. il comando **echo** stampa le stringhe corrispondenti ai nomi dei file sullo standard output ➔ provare ad usare anche il comando **ls**

COMANDO ls

Il comando **ls** (*list*) mostra le informazioni relative a file o directory

Se **ls** è invocato senza parametri, mostra le informazioni sui file e sulle directory della directory corrente (di default in ordine alfabetico), se invece è invocato con nomi di file mostra solo le informazioni su quei file, se è invocato con nomi di directory (a parte se si usa l'opzione –d) mostra le informazioni sui file e sulle directory delle directory specificate

Vediamo ora il significato di alcune opzioni del comando **ls**:

-l lista, oltre al nome, tutte le informazioni associate ai file, cioè tipo del file, permessi, numero link, proprietario, gruppo, dimensione in byte, ora ultima modifica

-a lista anche i nomi dei file "nascosti", cioè il cui nome inizia con '.'

-A come -a, escludendo però **.** e **..**

-F lista i nomi dei file visualizzando i file eseguibili con suffisso *, le directory con suffisso */*

-d nomedir lista le informazioni associate alla directory considerata come file, senza listarne il contenuto

-R lista ricorsiva dei file contenuti nella gerarchia

-i lista gli i-number dei file oltre al loro nome

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

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

NOTA BENE: le opzioni si possono combinare assieme
ES: ls -la
lista, oltre al nome, tutte le informazioni associate a tutti i file (anche quelli "nascosti")

PROTEZIONE

In un Sistema Operativo multi-utente, vi è la necessità di regolare gli ***ACCESSI*** alle informazioni

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

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

Ogni ***utente*** viene identificato all'interno del sistema (nella fase di LOGIN) con:
- un identificatore (**user ID**)
- e un gruppo (**group ID**)

Inoltre, ad ogni file è associato:
- user ID del proprietario
- group ID del proprietario
- un insieme di **9 bit di protezione**

I 9 bit di protezione rappresentano i **diritti di accesso** (detti anche **permessi**): i diritti indicano se la specifica operazione (**lettura r, scrittura w ed esecuzione x**) è consentita o meno per il proprietario, gli appartenenti al gruppo del proprietario e gli altri ➔ il S.O. ogni volta che un utente accede (in r, w oppure x) ad un file deve verificare se quell'utente ha i diritti corretti per farlo!

ESEMPIO:

9	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*
1	1	1	1	0	0	1	0	0
r	w	x	r	w	x	r	w	x
	USER			**G**ROUP			**O**THERS	

NOTA BENE: Esiste però un utente privilegiato: il gestore del sistema ➔ **SUPERUTENTE**

PROTEZIONE (segue)

Significato dei diritti rwx per le directory:

- Senza **r** NON si può visualizzare il contenuto della directory, ma si possono cancellare file e crearli
- Senza **w** NON si possono cancellare file nè crearli
- Senza **x** NON si può fare cd (e non si può neanche creare un file)

NOTA BENE: Per visualizzare i diritti di accesso (permessi) associati ad un file si deve usare il comando **ls –l nomefile**; per visualizzare i diritti di accesso (permessi) associati ad una directory si deve usare il comando ls **–ld** nomedir

COMANDI RELATIVI ALLA PROTEZIONE

chmod [u g o a] [+ -] [rwx] nomefile
chmod dirittiInOttale nomefile

I permessi possono essere concessi o negati ***SOLO*** dal proprietario del file ➔ comando **chmod** (*change mode*)

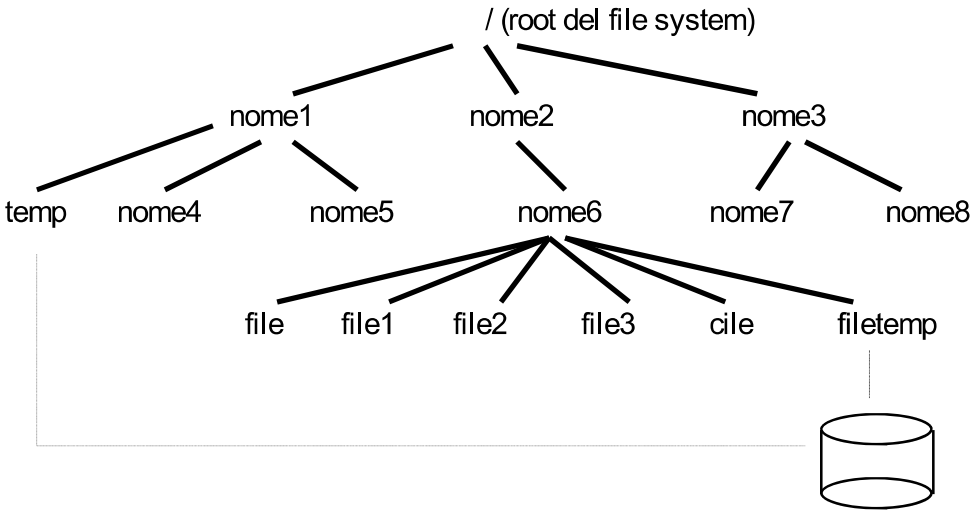
<u>Esempio di variazione dei diritti di accesso:</u>
chmod u+rwx,g+rx,o+rx /home/dir/file
chmod 0755 /home/dir/file

9	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*
1	1	1	1	0	1	1	0	1
r	w	x	r	w	x	r	w	x
	USER			**G**ROUP			**O**THERS	

ATTENZIONE: i comandi
chown nomeutente nomefile
che cambia il proprietario di un file/directory
chgrp nomegruppo nomefile
che cambia il gruppo di un file/directory
possono essere usati solo dal super utente ===> ***PERCHÈ***?

Linking di file e directory

Le stesse informazioni contenute in un file possono essere **visibili tramite NOMI diversi** ➔ quindi l'albero gerarchico di directory in realtà è un **GRAFO ACICLICO**



ln /nome2/nome6/filetemp /nome1/temp

➔ Utilizzando il comando **ln** (*link*), le stesse informazioni sono accessibili tramite DUE percorsi diversi (due nomi assoluti diversi):

- /nome2/nome6/filetemp
- /nome1/temp

Di default (quindi usato senza opzioni), il comando **ln** crea, nel file system, un **link HARDWARE** ➔ incremento numero link (nl)

NOTA BENE: il concetto di link hardware vale (implicitamente) anche per le directory

Se viene richiesta la **CANCELLAZIONE** di un file usando il comando **rm** (*remove*), le informazioni non sono eliminate se ci sono altri link (HW) ➔ viene decrementato nl ➔ le informazioni sono eliminate solo quando nl è uguale a 0!

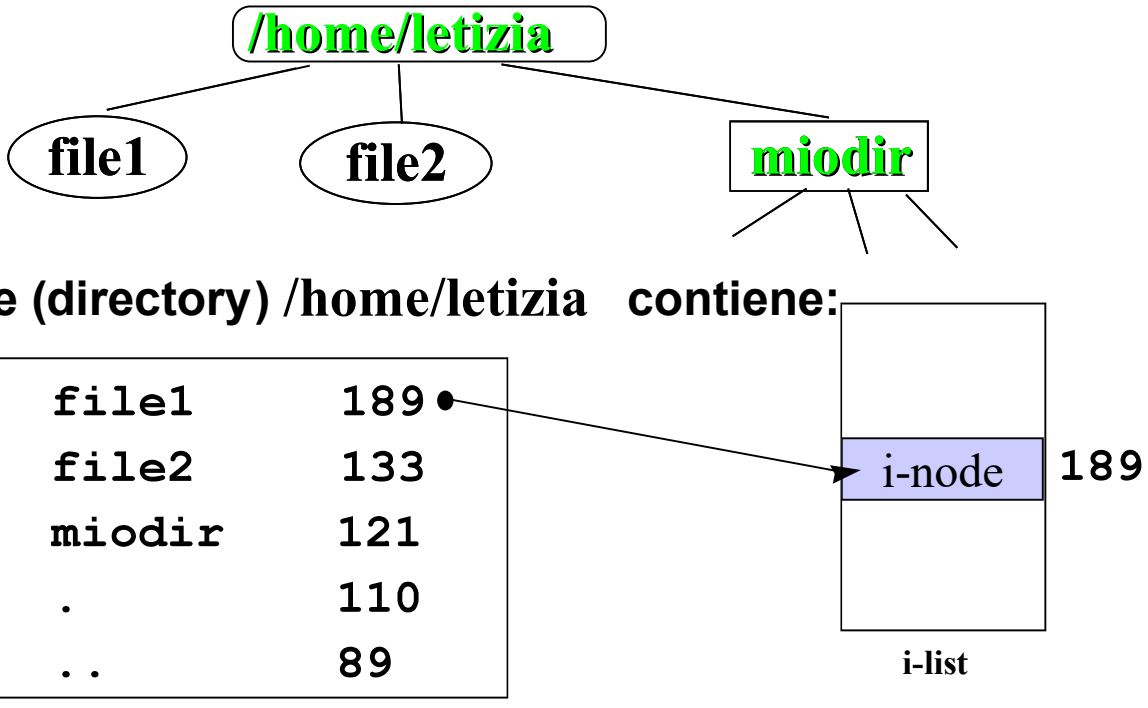
Il comando **mv** (*move*) che consente di 'spostare' un file nel grafo (e nel caso più semplice, consente di rinominare un file) corrisponde all'uso coordinato del comando **ln** e del comando **rm**

Struttura interna di una Directory

Per capire come vengono implementati i **link hardware** bisogna analizzare la struttura interna di una directory: per ogni file, la directory contiene

nomefile (relativo semplice)	**i-number**

Dove l'i-number consente di identificare in modo univoco l'i-node del file (che contiene tutte le informazioni di dettaglio sul file, a parte il nome, vedi lucido seguente!)



Quindi quando si usa il comando **ln** per creare un link hw, viene copiato l'elemento della directory in un'altra directory e viene (di solito) cambiato il nome ➔ entrambi gli elementi quindi contengono lo stesso i-number e quindi fanno riferimento allo stesso i-node ➔ all'interno dell'i-node viene incrementato il numero di link ➔ **si verifichi ciò prima e dopo la creazione di un link hw con il comando ls –li**

Anche i nomi relativi **.** e **..** sono trattati in modo uniforme e rappresentano **link**, rispettivamente, alla **directory stessa** e alla **directory padre**

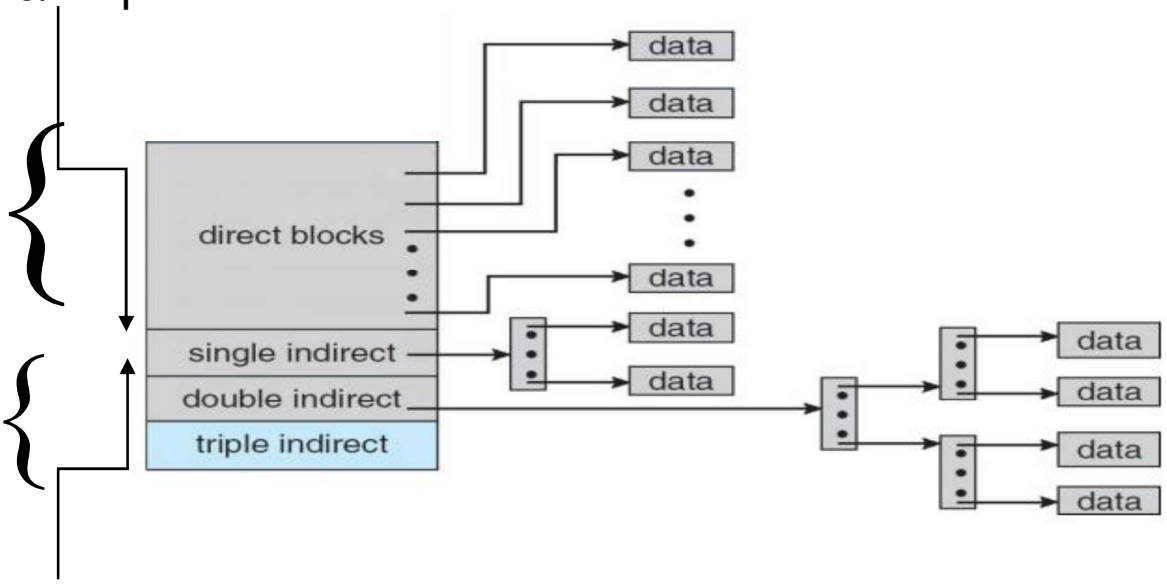
I-NODE

Il contenuto di I-Node per file *normali/directory* è

- ***tipo*** del file: (ordinario, directory o special file);
- i ***bit*** SUID, SGID, e 'sticky'; ➔ *3 bit speciali!*
- ***diritti*** read/write/execute per utente, gruppo e altri;
- numero dei ***link*** del file; ➔ *nl*
- ***identificatori*** user, group (UID e GID);
- ***dimensione*** del file ➔ in byte
- ***tempi di accesso*** (lettura, scrittura file e I-node)
- ***indirizzi*** di tredici/quindici blocchi per recuperare i blocchi di dati

PER RITROVARE i blocchi fisici del FILE:

10/12 puntatori diretti a blocchi di dati



3 puntatori a blocchi indice di primo, secondo, terzo livello

Esempio: `soELab@Lica02$ ls -li provalink`

```
15073694 -rw-r----- 2 soELab users 43 Mar  5  2017 provalink
```

Esiste anche la possibilità di creare **link simbolici o software** (ln -s) che però si comportano in modo diverso dai link hardware! ➔ verificare con il comando ls –li e verificare cosa succede se si cancella il file originale o se invece si cancella il link!

COMANDI RELATIVI AL FILE SYSTEM

DIRECTORY

mkdir nomedir ➔ (*make dir*) crea una directory
rmdir nomedir ➔ (*remove dir*) cancella una directory
N.B. deve essere VUOTA!
cd nomedir ➔ (*change dir*) modifica la directory corrente
pwd ➔ (*print working dir*) scrive su standard output la directory corrente
ls [nomedir/nomefile] ➔ (*list*) *comando già visto*

FILE

cp filesorg dest ➔ (*copy*) copia il filesorg nel file di nome dest, se dest è una directory copia il file con nome filesorg in quella directory
ln nomefile1 nomefile2 ➔ (*link*) *comando già visto*
mv nomefile1 nomefile2 ➔ (*move*) *comando già visto*
rm nomefile ➔ (*remove*) *comando già visto*
ma approfondiamo: elimina il link nomefile andando a decrementare il numero di link nell'i-node; se il numero di link è uguale a 0 allora cancella il file
N.B. utilizzare l'opzione –i (*interactive*) per fare richiedere una conferma (soprattutto nel caso per esempio di **rm** *), dato che un file cancellato NON può essere recuperato!
cat nomefile ➔ (*catenate*) *comando già visto*

Il nome *catenate* deriva dalla possibilità del comando cat di **concatenare** il contenuto di più file: infatti se si usa
cat nomefile1 nomefile2 … nomefilen
il contenuto dei file indicati viene riportato sullo standard output uno di seguito all'altro (concatenati!)

2 DEI 3 BIT SPECIALI DELL'I-NODE

Nell'i-node, oltre i 9 bit per i permessi, ci sono altri 3 bit che sono **bit speciali**: in particolare 2 di questi 3 bit hanno significato solo per i file che contengono **codice eseguibile**

Vediamone il significato:
SUID bit (set-user-id bit)
Se è a 1 in un file (con proprietario l'utente U) che contiene un programma eseguibile, il processo dell'utente U1 che lo manda in esecuzione viene considerato il proprietario U per la durata della esecuzione ➔ *Senza suid settato, un programma potrebbe causare errori, durante l'esecuzione da parte di un utente U1 a causa di operazioni di lettura/scrittura su file di U su cui l'utente U1 potrebbe non avere i diritti relativi*

ES: il comando **passwd** ha il SUID settato dato che deve modificare il file /etc/shadow (del superutente), altrimenti non modificabile direttamente da utenti diversi dal superutente

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

Tramite il SUID (e il SGID) si riesce ad implementare una protezione tipo astrazione di dato: solo tramite le operazioni (cioè i file eseguibili) messe a disposizione è possibile accedere ad informazioni altrimenti protette
⇨ si basa sulla presenza, per ogni processo, di:

- **identificatori** d'utente e di gruppo **reali** che rimangono fissati e corrispondono a quelli presenti nel file passwd
- **identificatori** d'utente e di gruppo **effettivi** che inizialmente sono uguali ai reali, ma possono cambiare per effetto della esecuzione di un file con SUID e/o SGID settati

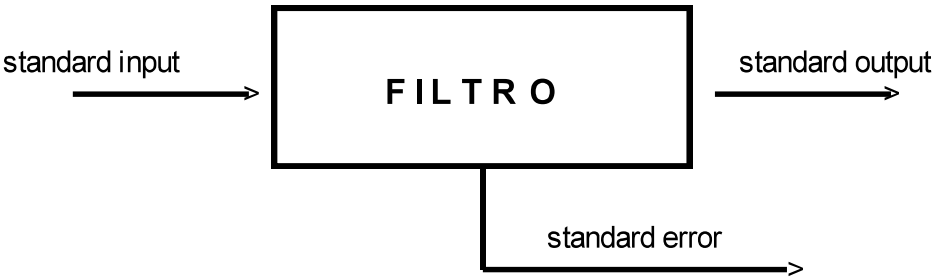
UNIX per autorizzare gli accessi alle informazioni verifica gli identificatori effettivi di utente e gruppo

RIDIREZIONE DELL'I/O

Alcuni comandi di UNIX sono filtri completi (li chiameremo comandi-filtro), mentre altri comandi sono filtri parziali

Un ***filtro*** è un programma che riceve in ingresso dati dallo standard input e produce risultati in uscita sullo standard output; eventuali errori vengono segnalati sullo standard error

ASSOCIAZIONI DI DEFAULT:
standard input --> tastiera del terminale
standard output --> video del terminale
standard error --> video del terminale



Ad esempio, un comando filtro parziale è il comando **ls** dato che produce un risultato che viene scritto su standard output e in caso di errore su standard error

Vediamo alcuni esempi di comandi-filtri che operano sullo standard input considerandolo a linee (di tutti questi esiste anche la normale versione comando!):
cat ➔ versione filtro del comando cat
more ➔ simile al cat, ma con paginazione dell'ouput
sort ➔ ordina le linee dello standard input
grep stringa ➔ cerca la stringa nello standard input
rev ➔ rovescia le linee dello standard input
wc ➔ conta caratteri, parole e linee dello stand. input
head [-numerolinee]
tail [-numerolinee]
➔ filtrano le prime/ultime linee dello standard input

OMOGENEITÀ dispositivi e file

Per ciascuno dei comandi-filtro (o anche dei comandi filtri parziali) lo standard input e lo standard output può essere ridiretto

ridirezione dello standard input

comando-filtro **<** fileinput

ridirezione dello standard output (la seconda in append)

comando-filtro **>** fileoutput

comando-filtro **>>** fileoutput

NOTA BENE:

Cosa producono i comandi seguenti?

1) **> filequalunque**
2) **comando > /dev/null**

Esempi di ridirezione:

ls -la > file1

cat file2 > file3

cat < file2

ls > file4
sort < file4 > file5

rev < file6 > file7
more < file7

wc -l < file8

ps > file9

pwd > file10
ls >> file10
cat < file10

RIDIREZIONE (segue)

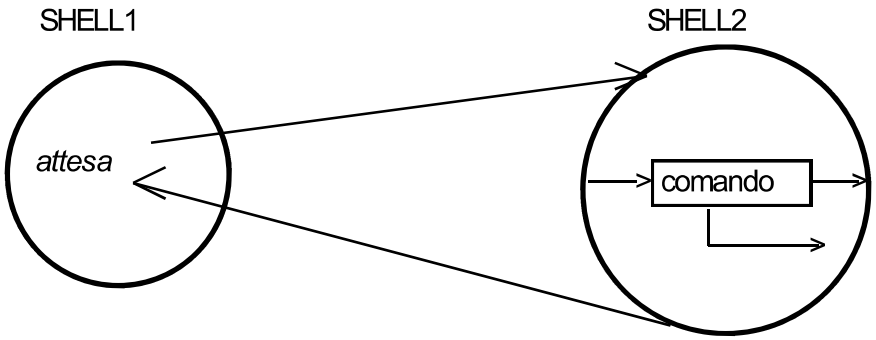
Ogni comando trova aperti

stdin, stdout, stderr

In caso di ridirezione,

- il file specificato come standard input è aperto in lettura
- il file specificato come standard output è creato e quindi aperto in scrittura *(cioè file nuovo o comunque con il contenuto cancellato)*
- in caso di append, il file specificato come standard output è aperto in scrittura e in append

I metacaratteri per la ridirezione sono riconosciuti e trattati prima del lancio del comando dal processo creato dallo shell (SHELL2) ➔ nella fase 3 (vedi lucido 7)



Il processo SHELL2 **esegue** il comando avendo prima collegato

il proprio standard input al *file di input*

il proprio standard output al *file di output*

specificati nella **shell di lancio** (SHELL1)

OSSERVAZIONE: nel caso si usi uno dei comandi-filtro utilizzando come standard input la tastiera (collegamento di default) ricordarsi che per concludere l'immissione di caratteri si deve usare la combinazione di tasti CTRL-D che rappresenta l'eof dato da tastiera!

RIDIREZIONE STANDARD ERROR

Supponiamo di essere in una directory che contiene solo i file di nome q, q1 e q2: vediamo il risultato dei comandi:

1) ls c* q*
c not found* ← scritto su standard error
q q1 q2 ← scritto su standard output
2) ls c* q* **>** file-output ⇨ si ridirige solo lo standard output
c not found*
Il file file-output contiene:
q q1 q2
3) ls c* q* **2>** file-error ⇨ si ridirige solo lo standard error
q q1 q2
Il file file-error contiene:
c not found*
4) ls c* q* **>** file-totale **2>&1**
⇨ si ridirigono sia output che error
Il file file-totale contiene (N.B. nessuna visualizzazione sul video):
c not found*
q q1 q2
5) ls c* q* **2>&1 >** file-o
⇨ si ridirige solo output (come in 2)
c not found*
Il file file-o contiene:
q q1 q2
NOTA BENE: l'ordine è importante poiché il collegamento viene fatto dinamicamente

OSSERVAZIONI:
1) comando **>& 2**
L'output del comando viene forzata sul canale di chiave 2, cioè l'assegnamento corrente di stderror
2) comando **> /dev/null >&1**
L'output e l'error del comando vengono entrambi forzati su /dev/null

RIDIREZIONE MULTIPLA

comando **>** *file1* **<** *file2* **>** *file3* **< file4 > file5**
In caso di ridirezione multipla, è solo l'ultimo collegamento che ha effetto quindi in questo caso il comando esegue con stdin da file4 e stdout su file5
NOTA BENE: EFFETTI COLLATERALI ➔ perdita del contenuto dei file file1 e file3

RIDIREZIONI AVANZATE

Nella shell, utilizzando una notazione specifica nella ridirezione (che si deve comunque usare nel caso dello standard error), è possibile alla invocazione del comando utilizzare anche altre 'chiavi'*

comando **3>** file3 **4>** file4 **5>** file5
Si richiede la apertura del file3 con chiave 3, del file4 con chiave 4 e del file 5 con chiave 5

Si noti che la numerazione prosegue dai file standard:
stdin, 0< ➔ non importa scrivere lo 0 (è il default)
stdout, 1> o 1>> ➔ non importa scrivere l'1 (è il default)
stderr, 2> o 2>> ➔ dal 2 in poi deve essere scritto

Per ogni numero maggiore di 2 si deve usare il simbolo <, > o >> a seconda di cosa richiede il comando che si sta usando

Nota bene: nessun comando di UNIX utilizza implicitamente numeri diversi da 0, 1 e 2, ma un programmatore può definire un programma che fa uso implicitamente di numeri maggiori di 2 e la shell garantisce, a seconda dei casi, apertura in lettura, apertura in scrittura o creazione per questi!

* A livello di programmazione, invece che il termine chiave si userà il termine file descriptor!

PIPE

In una shell, la **PIPE** (metacarattere |) **collega** automaticamente
lo **standard output** di un comando con
lo **standard input** del comando successivo

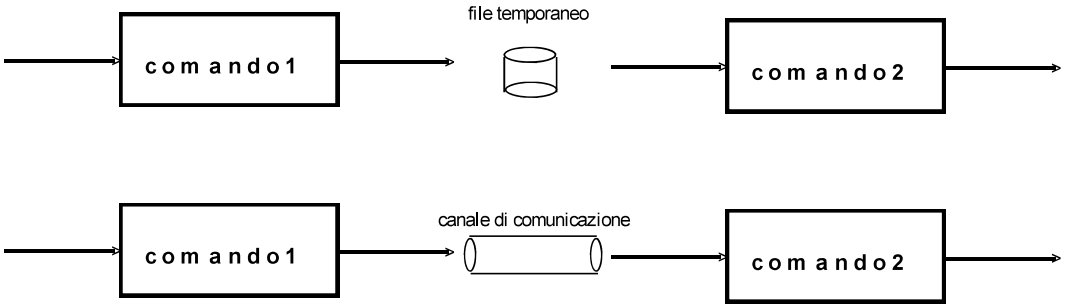
comando1 **|** comando2

Una stessa linea di comando può contenere più metacaratteri |

comando1 **|** comando2 **|** comando3 **|** comando4

Da un punto di vista di principio sono possibili DUE POSSIBILI SCHEMI IMPLEMENTATIVI:

1) Con file temporaneo ➔ DOS
2) Con canale di comunicazione fra processi ➔ UNIX



A noi chiaramente interessa come si comporta UNIX ➔ PIPE come **CANALE DI COMUNICAZIONE!**

Ogni comando che viene specificato a destra e a sinistra del metacarattere di pipe (|) viene eseguito da **un processo separato**: ogni coppia di processi risulta collegata con **uno specifico canale di comunicazione** ➔ ne esiste uno diverso per ogni pipe!

ATTENZIONE: non usare piping di comandi se NON serve!

PIPE (segue)

Ad esempio, se la shell esegue
$ comando1 | comando2

Vengono creati **due processi shell**, uno per eseguire **comando1** e uno per eseguire **comando2**
Inoltre viene creata una **pipe**, cioè un **canale di comunicazione** in modo che:

- il processo che esegue comando1, scrivendo sullo standard output, in realtà scrive sulla pipe
- il processo che esegue comando2, leggendo dallo standard input, in realtà legge dalla pipe e quindi riceve le informazioni inviate dal processo che esegue comando1

NOTA BENE: il tutto avviene in ***concorrenza*** e quindi a mano a mano che il processo che esegue comando1 scrive sulla pipe, il processo che legge dalla pipe riceve le informazioni e quindi può operare in concorrenza e non in modo sequenziale!

ESEMPI DI PIPING:

who | wc -l

ls | head -1

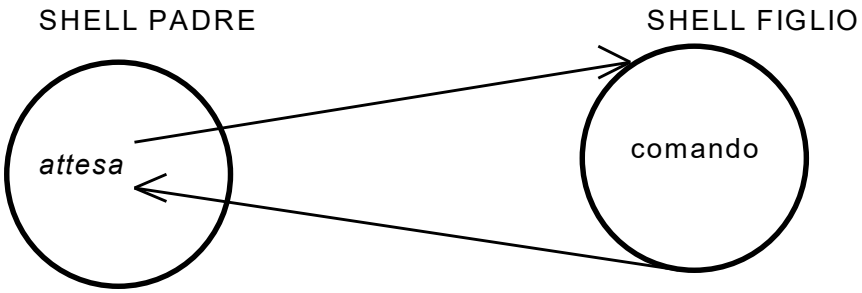
cd /tmp
ls | sort | rev | more

rev < file1 | sort | **tee** tmp | rev | more

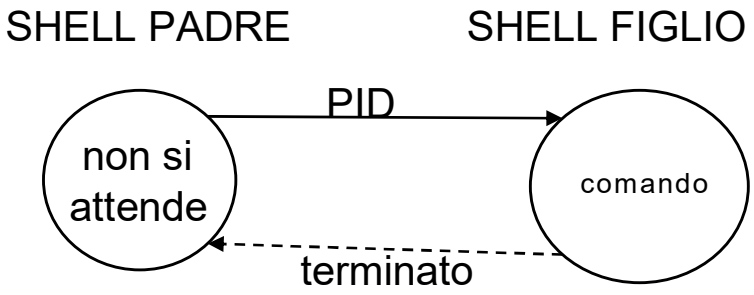
NOTA BENE: il comando **tee** è un comando-filtro che però ha bisogno di un parametro obbligatorio (un file, es. tmp) ➔ passa il contenuto dello standard input sullo standard output scrivendolo anche sul file (che viene creato/sovrascritto)!

ESECUZIONE IN PARALLELO

Normalmente, quando si manda in esecuzione un comando, la shell padre aspetta il completamento del processo shell figlio ➔ *esecuzione in* ***foreground*** **(sincrona)**



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



In questo caso, la shell invocante è immediatamente attiva (viene *SUBITO* visualizzato il prompt dei comandi):

```
$ comando [argomenti] &
[n] PID
$
```

dove PID è l'**identificatore del processo** eseguito in background e **n** è il numero di processi attivati in background

Quando termina il processo in background, la shell padre viene avvisata

Si possono avere **vari** processi in background, ma **un** solo processo in foreground per il terminale corrente

ESECUZIONE IN PARALLELO (segue)

Nel caso un processo in background non serva più o sia bloccato, lo si può uccidere con il comando **kill** PID

Un processo in background condivide lo stesso dispositivo fisico (il terminale) con tutti gli altri eventuali processi di backgroud, la shell padre e con l'eventuale processo in foregroud; quindi se:

1) scrive su standard output si mescolano le informazioni
   ⇨ la soluzione è utilizzare la ***ridirezione in uscita***

2) legge da standard input bisogna utilizzare la ***ridirezione in ingresso***, poichè lo standard input altrimenti, ***di default***, risulta chiuso per un processo in background

Quindi la tipica richiesta di esecuzione in parallelo è in realtà:

```
$ comando [argomenti] [< f1] > f2 &
[n] PID
$
```

Ad esempio:

```
$ ls -l / > tmpls &
```

ALTRI COMANDI

date
fornisce data ed orario corrente del sistema

who
w
elenca gli utenti collegati allo stesso sistema (con alcune differenze nelle informazioni riportate)

ps ➔ (*process status*) *comando già visto*
elenca i processi attivi nel sistema

man comando ➔ (*manual*) *comando già visto*
help del comando

which comando
riporta il nome assoluto del comando (simile al comando **whereis** che fornisce ulteriori informazioni, ad esempio su dove è il manuale del comando)

diff nomefile1 nomefile2
riporta le linee che presentano delle differenze nei due file

find direttory -name nomefile
riporta tutti i nomi assoluti del file di cui viene fornito il nome a partire alla directory specificata

Rivediamo alcune opzione di comandi già visti (presentati precedentemente come comandi-filtri)

sort [-r] nomefile [nomefile]
r sta per reverse. Molte opzioni:
solo il merge (opzione -m)
uscita su file (opzione -o nomefileout)

wc [-lwc] [nomefile]
si contano le linee (opzione l), le parole (opzione w) e i caratteri (opzione c) dello standard input o del file specificato

PROGRAMMAZIONE NELLO SHELL

Una shell ha un proprio linguaggio che può essere usato come un **linguaggio di programmazione interpretato**

➔ Linguaggio Comandi **del BOURNE SHELL*** (/bin/sh)

➔ Usato come linguaggio prototipale di sistema per il **rapid prototyping (sviluppo rapido)** delle applicazioni

Tramite questo linguaggio si possono scrivere ***file comandi (detti anche script)***, che usano comandi e inoltre:
- variabili
- passaggio dei parametri
- costrutti per il controllo di flusso

Per fare eseguire un ***file comandi*** di nome F.sh+ si può procedere in due modi:

- rendere eseguibile il file comandi e lanciarlo

```
$ chmod +x  F.sh
$ F.sh
```

➔ se non si ha la directory corrente nel PATH bisogna usare la notazione `./F.sh`

OSSERVAZIONE: la prima linea di un file comandi specifica la shell che deve essere usata per l'interpretazione

`#!/bin/sh` ➔ *shabang*

- invocare direttamente uno shell che lo esegua

```
$ sh F.sh
```

(N.B. Il secondo modo può essere utilizzato anche se il file comandi è stato reso eseguibile!)

Opzioni di **sh** utili per il *debugging* di programmi shell:

sh -x stampa i comandi e gli argomenti come sono eseguiti

sh -v stampa le linee del file comandi come sono lette dallo shell

---

* Useremo la sintassi definita dalla **BOURNE SHELL (perché è quella più semplice)**, che però viene accettata anche dalla **BASH** e dalla **DASH**!

+ Per convenzione i nomi dei file comandi li faremo sempre terminare con **.sh**!

VARIABILI in SHELL*

In una shell possiamo usare delle **variabili** ➔ le variabili sono rappresentate da una **coppia nome-valore**

Dato che siamo in un ambiente interpretato ***NON*** esiste la fase di definizione o dichiarazione di una variabile

➔ Una variabile esiste appena compare in una espressione di assegnamento (metacarattere **=)**

➔ **ATTENZIONE**: **NON** deve essere presente alcuno spazio bianco prima o dopo del simbolo di assegnamento

Esempi:

1) `$ a=    # la variabile a serve per …`
   Dopo questo assegnamento esiste una variabile di nome **a** che non ha alcun valore
2) `$ a=10 # la variabile a vale 10`
   Dopo questo assegnamento il valore della variabile di nome **a** è cambiato ed è diventato **"10"**

NOTA BENE: per inserire commenti
commento fino alla fine della linea

IMPORTANTE: il valore di una variabile è sempre trattato come una **stringa** ➔ la variabile **a** ha come valore la stringa costituita dai caratteri **1** e **0** e non il numero 10!

Il valore ad una variabile può essere cambiato a piacimento

A livello sintattico, il **valore di una variabile** viene indicato dal nome della variabile preceduto dal metacarattere **$**

riferimento sin. riferimento destro
nomevariabile1=$nomevariabile2

* In particolare ci riferiamo alla **BOURNE SHELL**, ma la sintassi che useremo da qui in poi può essere usata anche nella **BASH** e nella **DASH**!

AMBIENTE in SHELL

Ogni processo shell ha un **AMBIENTE di esecuzione** che è costituito da un insieme di **variabili** (con *nome* e *valore*) dette **VARIABILI DI AMBIENTE** ➔ variabili di shell speciali

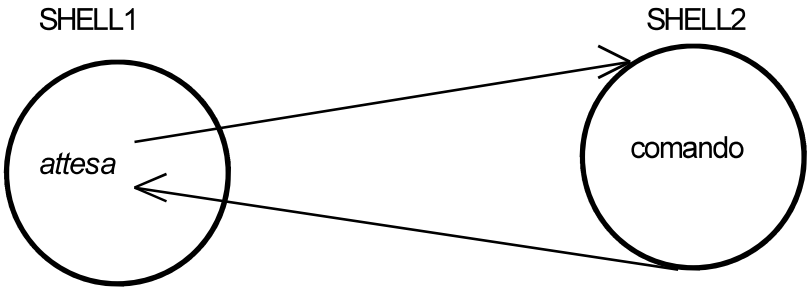
Per esempio, una variabile di ambiente memorizza la **directory corrente**, mentre la variabile di ambiente **HOME** indica la directory di accesso iniziale
Inoltre, un'altra variabile di ambiente importante è la variabile **PATH** che contiene le directory in cui la shell deve ricercare ogni comando da eseguire

Il comando **env** (*environment*) mostra l'ambiente corrente di ogni processo shell

Per visualizzare invece il valore di una singola variabile di ambiente si può usare il comando **echo** indicando però il nome della variabile preceduto dal metacarattere **$**:
Es: `echo $HOME; echo $PATH`

Come abbiamo già visto, in UNIX, **ogni comando** viene (in genere) **eseguito da un nuovo processo shell** (generato/creato dal processo shell che mostra il prompt dei comandi)

Il processo **shell padre:**
- attende la fine dell'esecuzione del sottoshell ➔ esecuzione in **foreground** (***sincrona***)



- ***NON*** attende la fine dell'esecuzione del sottoshell, ma prosegue ➔ *esecuzione in* ***background*** **(asincrona)**

AMBIENTE in SHELL (segue)

Quando viene creato un nuovo processo di shell per eseguire un comando, questo processo riceve una **COPIA** dell'ambiente del processo padre

- ➔ quindi tutte le variabili di ambiente del padre sono copiate nell'ambiente del figlio
- ➔ il processo shell figlio può usare le stesse variabili di ambiente del padre (con gli stessi valori)
- ➔ se il processo shell figlio e/o il processo shell padre modificano il valore di una variabile di ambiente, tale modifica VALE solo per il processo shell che la effettua!

Le variabili di shell che non fanno parte dell'ambiente NON vengono copiate nel processo shell figlio

Quindi riassumendo, in un processo shell esistono:

- **variabili di ambiente (ereditate dai processi figli per copia)**
- **variabili di shell (NON ereditate dai processi figli)**

Esiste la possibilità con uno specifico comando di far diventare variabile di ambiente una semplice variabile di shell ➔ COMANDO **EXPORT**

SINTASSI: export nome-var

Dopo aver usato il comando **export**, una variabile di shell diventa parte dell'ambiente

Nota bene: in Bourne Shell, qualunque modifica ad una variabile di ambiente per avere effetto sull'ambiente **deve** essere seguita dal comando **export** per la variabile modificata

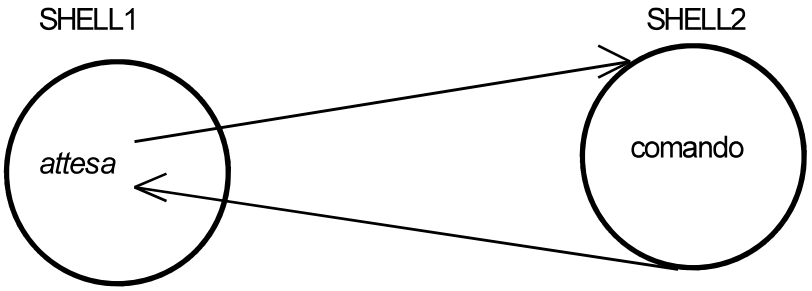
Le shell più recenti potrebbero non avere questa necessità, ma **noi useremo sempre l'export** anche se potrebbe non essere necessario

SOSTITUZIONI (ESPANSIONI)

Come abbiamo già visto, la shell è un **processore comandi**, che (dal lucido 7):
1) **accetta** un comando* richiesto dall'utente
2) lo **interpreta**
3) e (*se l'interpretazione è riuscita*), esegue le sostituzioni, ricerca il comando, realizza ridirezioni/piping e lo **esegue**

Quindi **ogni comando** viene (in genere) **eseguito da un nuovo processo shell** (detta sottoshell, generato/creato dal processo shell che mostra il prompt dei comandi)

Il processo **shell padre** (sulla base della fase di interpretazione che comporta *anche* verificare o meno **la presenza del metacarattere &** prima del <CR>):
- attende la fine dell'esecuzione del sottoshell ➔ esecuzione in **foreground** (***sincrona***)



- ***NON*** attende la fine dell'esecuzione del sottoshell, ma prosegue ➔ *esecuzione in **background*** **(asincrona)**

Prima della esecuzione, la sottoshell deve eseguire le sostituzioni trattando specifici metacaratteri ➔ le sostituzioni comportano l'espansione in base alla presenza nella linea di comando (che viene scandita effettuando un parsing), di caratteri speciali

In particolare vengono eseguite 3 sostituzioni il cui ORDINE è fissato in modo rigoroso … Vediamolo

* **RICORDARSI**: per comando si intende sia un comando definito dal sistema, sia un programma eseguibile sviluppato da un utente e sia uno script (file comandi)

SOSTITUZIONI (segue)

Le sostituzioni vengono espanse in questo ordine:

1) **Sostituzione delle variabili/parametri**
   Ogni riferimento al valore di una variabile (**$nome**) o di un parametro viene espanso nel valore corrispondente
   ES: x=12 **#ATTENZIONE: no bianchi**
   echo $x # produce 12

2) **Sostituzione dei comandi**
   I comandi che sono indicati tra ` ` (*backquote* o apici rovesci) sono eseguiti e ne viene espanso il risultato
   ES: echo `pwd`/F # stampa nome assoluto del file F (supposto presente nella directory corrente)

3) **Sostituzione dei nomi di file**
   Le wildcard *****, **?**, **[ ]** sono espanse ai nomi di file nel file system secondo un algoritmo di **pattern matching**

RICORDARSI CHE:

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

? fa *match* con un qualunque carattere in un nome di file

INOLTRE:

[c1c2c3] (o anche **[c1,c2,c3]**) fa *match* con un qualunque carattere, in un nome di file, compreso tra quelli nell'insieme. Anche **range** di valori: **[c1-cn]**; si può usare anche il metacarattere **!** che nega il pattern seguente

Ancora sulla fase 3-sostituzioni:

**** non interpreta come speciale il carattere successivo

➔ **\ viene detto carattere di ESCAPE**

Per esempio:

```
ls *\**
```

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

ESEMPI DI SOSTITUZIONI DI TIPO 3

echo *
visualizza tutti i nomi di file della directory corrente

echo * ← **controesempio del precedente**
esegue l'echo del carattere *

ls [q-s]*
lista i nomi dei file che iniziano con almeno un carattere compreso tra q e s.

ls *[!0-9]
lista i nomi dei file che terminano con caratteri **non** numerici

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

ls * lista i file della directory corrente (entrando nelle directory e listandoli a sua volta)

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

ls /*/*/* lista tutti i file delle directory di secondo livello a partire dalla root (entra nelle directory a sua volta)

ls -d /*/*/* lista tutti i file delle directory di secondo livello a partire dalla root (le directory sono trattate come file)

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

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

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

VARIABILI (ancora)

Come abbiamo visto:
- NON è presente alcuna fase di definizione delle variabili
- Il nome delle variabili è libero (alcune però sono predefinite)
- Il valore delle variabili viene recuperato con il metacarattere **$**
- Per cambiare valore ad un variabile si deve usare l'assegnamento ➔ **variabile=valore** # niente spazi!!!
- Il valore di una variabile è sempre considerato una **STRINGA**

ESEMPIO:

```
$ i=ciao
$ echo i $i          # visualizza i ciao
$ i=12
$ echo i $i          # visualizza i 12
$ j=$i+1             # niente spazi prima e dopo il car. +
$ echo j $j          # visualizza j 12+1
```

Le variabili sono trattate come stringhe di caratteri

⇨ per valutare espressioni aritmetiche serve il comando **expr**

➔ attenzione che bisogna usare anche la sostituzione 2) per fare espandere al valore della espressione

ESEMPIO:

```
$ j=`expr $i + 1`    # ATTENZIONE ORA CI VOGLIONO gli spazi prima e dopo il car. +
$ echo j $j          # visualizza j 13
```

Il comando expr consente di effettuare tutte le normali operazioni aritmetiche: +, -, *, /, %

N.B. per la moltiplicazione bisogna usare l'escape (\)!

ESEMPIO:

```
$ j=`expr $i * 2`          #expr: syntax error
$ j=`expr $i \* 2`
```

INIBIZIONE DELLE SOSTITUZIONI

Ci sono dei casi in cui non si vuole che le sostituzioni precedentemente indicate (lucido 33) siano eseguite

Una particolare sintassi consente di inibire (cioè non svolgere) le sostituzioni (tutte o solo alcune)

1) **'...'** (quote, singolo apice) ➔ **nessuna espansione** non vengono operate le sostituzioni 1, 2, 3 ➔ inibizione totale
2) **"..."** (doublequote, doppi apici) ➔ **solo sostituzioni 1 e 2** (ma non la 3) ➔ non vengono espansi i wildcard ➔ inibizione parziale

ESEMPIO:

```
y = 3
echo ' $y  e `pwd` e * ' # produce $y e `pwd` e *
echo " $y  e `pwd` e * " # produce 3 e /home/… e *
```

OSSERVAZIONE IMPORTANTE:

Il processore comandi esegue, nell'ordine visto, una 'passata' di sostituzioni

➔ Nel caso l'espansione risultante richieda nuove sostituzioni, bisogna richiedere esplicitamente alla shell di effettuarle! ⇨ uso del comando **eval**

ESEMPIO:

```
$ x='ls -l $z'
$ z=nomefile
$ $x            ← produce un errore
$ eval $x       ← corretto (supponendo esista nomefile)
```

ERRORE: Il processo figlio sottoshell esegue la sostituzione 1), quindi il valore di x viene espanso e sostituisce `$x` con `ls -l $z` , quindi non essendo presente nessuna sostituzione 2 e 3), viene ricercato il comando **ls** e quindi mandato in esecuzione ➔ **ls** dà errore dato che (molto presumibilmente) NON esiste alcun file che si chiama **$z**

PASSAGGIO ARGOMENTI

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

Gli argomenti passati all'invocazione di un comando costituiscono i valori dei **parametri**

➔ I parametri all'interno di un file comandi (script) sono indicati con ***variabili posizionali*** rispetto alla linea di comando

$1 è il primo argomento

$2 è il secondo argomento, etc

La shell consente di recuperare anche il nome del comando stesso con la notazione **$0**

OSSERVAZIONE: Non esiste la possibilità di modificare direttamente il valore di un parametro, ma …:

1) COMANDO **shift**
   ⇨ opera una traslazione del valore dei parametri verso il basso
   si perde il primo argomento e gli altri sono spostati
   (il vecchio $2 è diventato $1, etc. ⇨ nessun effetto su $0)
2) COMANDO **set**
   ⇨ È possibile riassegnare i parametri
   set exp1 exp2 exp3 ...
   gli argomenti del comando set sono assegnati secondo la posizione ai parametri

ALTRE VARIABILI

$* l'insieme di tutte le variabili posizionali, che corrispondono agli argomenti del comando: $1, $2, ecc.

$# il numero di argomenti passati al comando (***$0 chiaramente escluso***)

$? il return code (valore di ritorno) dell'ultimo comando eseguito

$$ il numero del processo in esecuzione

COSTRUTTI PER CONTROLLO FLUSSO

Anche la shell, come un qualunque linguaggio di programmazione imperativo, fornisce i principali costrutti per il **controllo di flusso**: alternativa semplice, alternativa multipla e cicli

Nella shell, questi costrutti sono basati, invece che su una espressione booleana, sul valore di ritorno (return code) dei comandi

Infatti, come abbiamo visto, ogni comando fornisce in uscita un valore di ritorno (nella variabile $?), che può essere utilizzato per renderlo parte di espressioni

In genere, i comandi di UNIX hanno come valore di ritorno:
- **valore zero (0)** ➔ per indicare che il comando ha avuto SUCCESSO
- **valore positivo (qualunque > 0)** ➔ per indicare che il comando ha avuto INSUCCESSO

ESEMPI:

1) **cp a.txt b.txt**

se il comando non riesce ➔ ***stato insuccesso*** (valore > 0)
p.e. a.txt non esiste ➔ il ritorno è un codice di errore
altrimenti ➔ ***stato successo*** (valore 0)

2) **ls F**

stato successo se viene trovato il file di nome F

3) **grep abc F**

stato successo se viene trovata la stringa abc nel file di nome F

OSSERVAZIONE: se i comandi degli esempi 2 e 3) vengono usati in un costrutto per il controllo di flusso si dovrà usare **ridirezione dell'output su /dev/null**!

ALTERNATIVA SEMPLICE

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

PRIMO ESEMPIO: file if1.sh

```
#!/bin/sh
if grep $1 $2
then echo TROVATO STRINGA $1 NEL FILE $2
else echo NON TROVATO STRINGA $1 NEL FILE $2
fi
```

Come detto prima, grep scrive su standard ouput (ed eventualmente su standard error) e quindi meglio passare al

SECONDO ESEMPIO: file if2.sh

```
#!/bin/sh
if grep $1 $2 > /dev/null 2>&1
then echo TROVATO STRINGA $1 NEL FILE $2
else echo NON TROVATO STRINGA $1 NEL FILE $2
fi
```

OSSERVAZIONE:
La sintassi del costrutto if (così come dei seguenti) deve essere rispettata rigorosamente come è stata indicata e cioè

if lista-comandi ← **a capo!**
then comandi ← **then** va a linea nuova
[**else** comandi] ← **else** va a linea nuova
fi ← **fi** va a linea nuova

Se si preferisce scrivere le parole-chiave then, else e fi senza andare a capo bisogna usare il metacarattere **;**
Ad esempio:
if lista-comandi ; **then** comandi
[**else** comandi]
fi

COMANDO TEST

Il comando **test** è usato in particolare nel costrutto **if**

Comunque il comando **test**, in generale, serve per la valutazione di una espressione che ha come valore di ritorno 0 in caso di successo, altrimenti un valore di ritorno diverso da zero in caso di insuccesso

Vediamo le forme più utilizzate del comando test:

1) **test -opzioni nome**

test	**-f**	nomefile	esistenza file nomefile
	-d	nomedir	esistenza directory nomedir
	-r	nomefile/dir	diritto di lettura su file/dir (**-w** e **-x**)

...

2) **test stringa1 = stringa2**
valuta se due stringhe sono uguali
NOTA BENE: in questo caso ci deve essere uno spazio prima e uno dopo il metacarattere di =
test stringa1 != stringa2
valuta se due stringhe sono diverse

3) **test -z stringa1** valuta se la stringa è nulla
test stringa1 valuta se la stringa non è nulla

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

OSSERVAZIONE: test stringa1 = stringa e
test numero1 –eq numero2
a livello di semantica sono diversi

Espressioni booleane

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

ESEMPI:

```
if test $NC -gt 0 -a $NC -le $2
```
➔ 0 < $NC <= $2

```
if test ! -d $1 -o ! -x $1
```
➔ se $1 non è un dir o non è traversabile …

COMANDO READ

In un file comandi, oltre che lo standard output, si può usare anche lo standard input usando il comando **read**

read var1 var2 var3 #input

le stringhe fornite in ingresso dall'utente (o prelevate da un file in ridirezione) vengono attribuite alle variabili secondo la corrispondenza posizionale

COMANDO ECHO (ancora)

Nel caso si voglia essere sicuri che una stringa venga riportata sul terminale corrente (indipendentemente che alla invocazione del file comandi venga utilizzata la ridirezione) si deve forzare l'output del comando echo su **/dev/tty** che rappresenta il terminale corrente!

ESEMPIO: file `LeggiEMostra.sh`

```
#!/bin/sh
#file comandi leggi e mostra
if test -z $1
then    echo Errore: voglio un parametro
        exit 1
fi
if test ! -f $1 -o ! -r $1
then    echo Errore: $1 non file oppure non leggibile
        exit 2
fi
echo "vuoi visualizzare il file $1 (si/no)?" > /dev/tty
read var1
if test $var1 = si
then     ls -la $1; cat $1
else echo niente stampa di $1
fi
```

NOTA BENE: I file comandi sono trattati in modo **OMOGENEO** ai comandi di sistema, quindi all'invocazione si può usare ridirezione in ingresso e in uscita, piping e anche qualunque sostituzione

ALTERNATIVA MULTIPLA

Alternativa multipla, secondo il valore della variabile var

case $var **in**
pattern-1**)** comandi **;;**
...
pattern-i **|** pattern-j **|** pattern-k**)** comandi **;;**
...
pattern-n**)** comandi **;;**
esac

Il **case** consente di avere maggiori flessibilità sia nel controllo del valore di una stringa e sia nel controllo sul numero di parametri (stretto o lasco)

ESEMPI:

1) readCase.sh

```
#!/bin/sh
echo "Fornire una risposta (affermativa ===> Si, si, Yes, yes)"
read risposta
case $risposta in
S* | s* | Y* | y*) echo OK;;
*)                 echo NO;;
esac
```

2) nroPar.sh

```
#!/bin/sh
# controllo nro parametri (stretto)
case $# in
0|1|2)  echo "Pochi parametri $# ==> $* "
        echo Usage is: $0 file1 file2 file3
        exit 1;;
3)      echo "Numero giusto di parametri $# ==> $* " ;;
*)      echo "Troppi parametri $# ==> $* "
        echo Usage is: $0 file1 file2 file3
        exit 2;;
esac
echo "Qui si procede con i $# parametri ==> $* e
si possono fare altri controlli sul loro tipo"
```

RIPETIZIONI ENUMERATIVE

for var [**in** list]
do
comandi
done

Il costrutto di controllo **for** effettua un ciclo basandosi sulla scansione della lista list: la variabile var assume, ad ogni iterazione, via via i valori elencati

NOTA BENE: se **in list** MANCA, allora di default la lista è quella dei parametri e quindi è come se venisse scritto **in $***

ESEMPI:

1) `for i in *` # la lista corrisponde all'espansione di *
#esegue per tutti i file e directory della directory corrente

2) `for i in d*` # la lista corrisponde all'espansione di *
#esegue per tutti i file e directory della directory corrente il cui nome inizia per d

3) **for** i **in** `cat filetemp` # la lista corrisponde all'espansione del comando cat filetemp e quindi ogni stringa contenuta in filetemp costituisce un elemento della lista

OSSERVAZIONE: Chiaramente il **nome della variabile** che viene utilizzata in un **for** può essere scelto a piacere: i, j, ma anche nomefile, nomedir, var a seconda del significato della lista utilizzata dopo la parola-chiave **in**

ESEMPIO: creazione veloce di un insieme di file vuoti

```
#file crea.sh
for i           # cioé in $*
do > $i         # ridirezione dello standard output su $i
done
```

RIPETIZIONI NON ENUMERATIVE

while lista-comandi
do
comandi
done

Con il costrutto **while** si ripete fintanto che il valore di ritorno dell'ultimo comando della lista è uguale a zero, cioè fino a che ha successo

ESEMPIO: file `ce.sh`

```
while test ! -f $1
do sleep 10
done
```

until lista-comandi #duale di while
do
comandi
done

Con il costrutto **until** si ripete fino a che l'ultimo comando produce insuccesso

ESEMPIO: file `ceutente.sh`

```
until who | grep $1
do sleep 10
done
```

Uscite anomale:
1) **in caso di errore si deve usare exit** [status] (default 0)
2) in caso di voler uscire da un ciclo sono presenti le parole-chiave analoghe al C: continue e break

NOTA BENE:

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

ESEMPIO DI FILE COMANDI RICORSIVO

PROBLEMA: vogliamo cercare un **file** (*nome relativo semplice*) in una **sottogerarchia** che viene identificata da un *nome assoluto di directory*, se specificato, altrimenti si intende quella a partire dalla directory corrente

Suddividiamo il problema in due sottoproblemi:
1) Controllo parametri di invocazione, settaggio PATH e invocazione ricorsiva, eventuali azioni conclusive
2) File comandi ricorsivo che effettua la ricerca

SOLUZIONE PRIMO SOTTOPROBLEMA:

```
#!/bin/sh
#file comandi Beginc.sh
case $# in                    ← controllo numero parametri!
0) echo "Usage is: $0 [directory] file"
   exit 1;;
1) d=`pwd`; f=$1;;
2) d=$1; f=$2;;
*) echo "Usage is: $0 [directory] file"
   exit 1;;
esac
#controllo che il primo parametro sia dato in
forma assoluta
case $d in                    ← controllo primo parametro!
/*) ;;
*) echo Errore: $d non in forma assoluta
   exit 2;;                   ← valori diversi per maggiore usabilità!
esac
#controllo che il secondo parametro sia dato in
forma relativa semplice
case $f in                    ← controllo secondo parametro!
*/*) echo Errore: $f non in forma relativa semplice
    exit 3;;
*) ;;
esac
```

segue ESEMPIO DI FILE COMANDI RICORSIVO

```
#controllo che il primo parametro sia una directory e che sia traversabile
if test ! -d $d -o ! -x $d   ← controllo dir e traversabile!
then echo Errore: $d non directory; exit 4
fi

PATH=`pwd`:$PATH             ← settaggio plausibile!
export PATH
echo Beginc.sh: Stiamo per esplorare la directory $d
Cercafile.sh $d $f           ← invocazione file comandi ricorsivo!
echo HO FINITO TUTTO         ← azioni conclusive!
```

SOLUZIONE SECONDO SOTTOPROBLEMA:

```
#!/bin/sh
#file comandi Cercafile.sh
#ricerca in breadth-first
cd $1
if test -f $2
then
echo il file $2 si trova in `pwd`
fi

for i in *
do
if test -d $i -a -x $i
then
  echo Stiamo per esplorare la directory `pwd`/$i
  Cercafile.sh `pwd`/$i $2
fi
done
```

OSSERVAZIONI:

1) In quale directory ci troviamo al termine della esecuzione del primo file comandi?
2) Come si deve modificare il file comandi ricorsivo per avere la ricerca in depth-first?

Ora si possono svolgere tutte le parti shell dei compiti di esame!

COMANDI INTERNI (built-in)

Esistono dei comandi classificati come interni che non mettono in esecuzione un nuovo processo SHELL

Alcuni di questi sono:

.
break
continue
cd
eval
exit
export
read
set
shift
trap
umask

NOTA BENE: Vedi manuale di sh (sul sito) per un elenco completo (alla voce Special commands)

ESEMPIO:

. file

Legge ed esegue i comandi contenuti in file: non viene creato un sottoshell e non serve il diritto di esecuzione su file

TRATTAMENTO DEI SEGNALI

Un programma Shell può tenere anche conto di EVENTI ASICRONI

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

Il comando
trap comandi numerosegnale

associa al segnale, indicato da numerosegnale, i comandi specificati
All'arrivo del segnale, quindi, si eseguono i comandi specificati

Esempi di numeri di segnali:
- **0** fine del file
- **2** CTRL-C: interrupt da tastiera
- **3** CTRL-Z: stop da tastiera

ESEMPIO:
trap 'rm /tmp/*; exit' 2
associa al segnale 2 la ripulitura della directory tmp (dove si inseriscono i file temporanei) e quindi la terminazione della sessione di lavoro

Altre azioni possono essere

Ignorare il segnale
trap ' ' 2

default: TERMINAZIONE
trap 2

PROVARE ad ignorare il CTRL-C e poi lanciare un comando e provare ad abortirlo!
Quindi, ripristinare l'azione di terminazione