

Introduzione a UNIX shell e file comandi

Shell

Programma che permette di far *interagire l'utente (interfaccia testuale) con SO tramite comandi*

- resta in attesa di un comando...
- ... mandandolo in esecuzione alla pressione di <ENTER>

In realtà (lo vedremo ampiamente) **shell è un interprete comandi evoluto**

- potente *linguaggio di scripting*
- interpreta ed esegue comandi da *standard input* o da *file comandi*

Differenti shell

Ciclo di esecuzione della shell

- La shell non è unica, un sistema può metterne a disposizione varie
 - **Bourne shell** (standard), C shell, Korn shell, ...
 - L'implementazione della **bourne shell in Linux** è **bash** (/bin/bash)
- Ogni utente può indicare la shell preferita
 - La scelta viene memorizzata in /etc/passwd, un file contenente le informazioni di tutti gli utenti del sistema
- La shell di login è quella che richiede inizialmente i dati di accesso all'utente
 - Per **ogni utente connesso** viene generato un **processo dedicato** (che esegue la shell)

```
loop forever
  <LOGIN>
  do
    <ricevi comando da file di input>
    <interpreta comando>
    <esegui comando>
  while (! <EOF>)
  <LOGOUT>
end loop
```

Accesso al sistema: login

Per accedere al sistema bisogna possedere una coppia **username e password**

- NOTA: UNIX è case-sensitive

SO verifica le credenziali dell'utente e manda in esecuzione la sua **shell di preferenza**, posizionandolo in un **direttorio di partenza**

- Entrambe le informazioni si trovano in `/etc/passwd`

Comando `passwd`

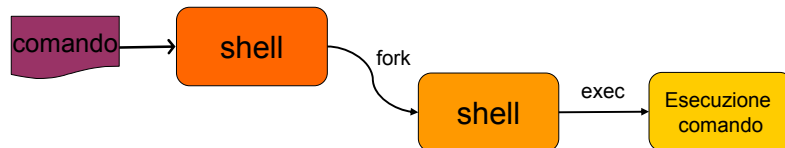
- È possibile **cambiare la propria password** di utente, mediante il comando `passwd`
- Se ci si dimentica della password, bisogna chiedere all'amministratore di sistema (utente `root`)

Uscita dal sistema: logout

- Per uscire da una shell qualsiasi si può utilizzare il comando **exit** (che invoca la system call `exit()` per quel processo)
- Per uscire dalla shell di login
 - `logout`
 - `CTRL+D` (che corrisponde al carattere <EOF>)
 - `CTRL+C`
- Per rientrare nel sistema bisogna effettuare un nuovo login

Esecuzione di un comando

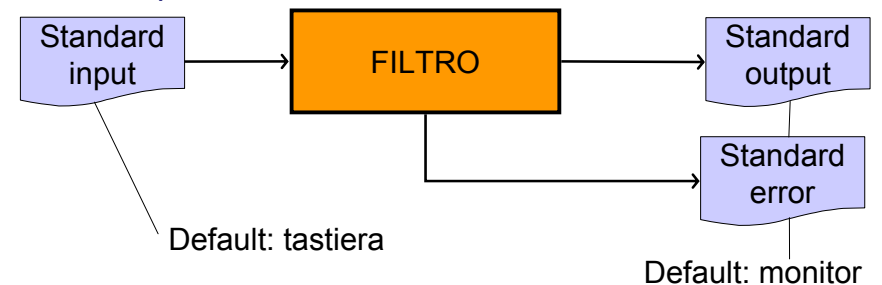
- Ogni comando richiede al SO l'esecuzione di una particolare azione
- I **comandi principali** del sistema si trovano nella directory `/bin`
- Possibilità di **realizzare nuovi comandi (scripting)**
- Per ogni comando, shell **genera un processo figlio dedicato alla sua esecuzione**
 - Il processo padre **attende la terminazione del comando** (foreground) **o prosegue in parallelo** (background)



Comandi e input/output

I comandi UNIX si comportano come FILTRI

- un filtro è un programma che riceve un ingresso da un input e produce il risultato su uno o più output



Manuale

esiste un **manuale on-line** (`man`), consultabile per informazioni su ogni comando Linux. Indica:

- ▢ **formato del comando (input) e risultato atteso (output)**
- ▢ **descrizione delle opzioni**
- ▢ possibili restrizioni
- ▢ file di sistema interessati dal comando
- ▢ comandi correlati
- ▢ eventuali bug

per uscire dal manuale, digitare `:q` (*quit* per editor tipo vi)

Formato dei comandi

tipicamente: ***nome -opzioni argomenti***

esempio: `ls -l temp.txt`

convenzione nella rappresentazione della sintassi comandi:

- ▢ se un'opzione o un argomento possono essere omessi, si indicano tra quadre **[opzione]**
- ▢ se due opzioni/argomenti sono mutuamente esclusivi, vengono separati da | **arg1 | arg2**
- ▢ quando un arg può essere ripetuto n volte, si aggiungono dei puntini **arg...**

Cenni pratici introduttivi all'utilizzo del file system Linux

File

File come **risorsa logica** costituita da **sequenza di bit**, a cui viene dato un nome

Astrazione molto potente che consente di **trattare allo stesso modo entità fisicamente diverse** come file di testo, dischi rigidi, stampanti, direttori, tastiera, video, ...

- **Ordinari**
 - ▢ archivi di dati, comandi, programmi sorgente, eseguibili, ...
- **Directory**
 - ▢ gestiti direttamente solo da SO, contengono riferimenti a file
- **Speciali**
 - ▢ dispositivi hardware, memoria centrale, hard disk, ...

In aggiunta, anche:

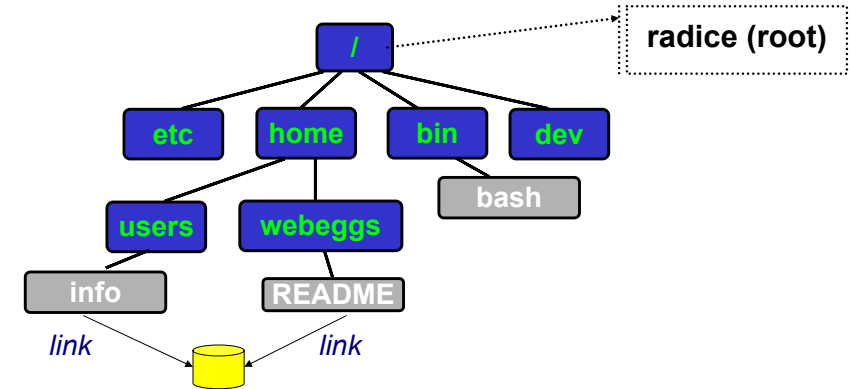
- FIFO (pipe) - file per la comunicazione tra processi
- soft link - riferimenti (puntatori) ad altri file o direttori

File: nomi

- È possibile nominare un file con una **qualsiasi sequenza di caratteri (max 255)**, a eccezione di '.' e '..'
- È sconsigliabile utilizzare per il nome di file dei caratteri speciali, ad es. **metacaratteri e segni di punteggiatura**
- ad ogni file possono essere associati **uno o più nomi simbolici (link)** ma ad ogni file è associato **uno e un solo descrittore (i-node)** identificato da un intero (i-number)

directory

File system Linux è organizzato come un grafo diretto aciclico (DAG)



Gerarchie di directory

- All'atto del login, l'utente può cominciare a operare all'interno di una specifica directory (**home**). In seguito è possibile cambiare directory
- È possibile visualizzare il percorso completo attraverso il **comando pwd** (print working directory)
- Essendo i file organizzati in **gerarchie di directory**, SO mette a disposizione dei comandi per muoversi all'interno di essi

Nomi relativi/assoluti

Ogni utente può specificare un file attraverso

- **nome relativo**: è riferito alla posizione dell'utente nel file system (direttorio corrente)
- **nome assoluto**: è riferito alla radice della gerarchia /

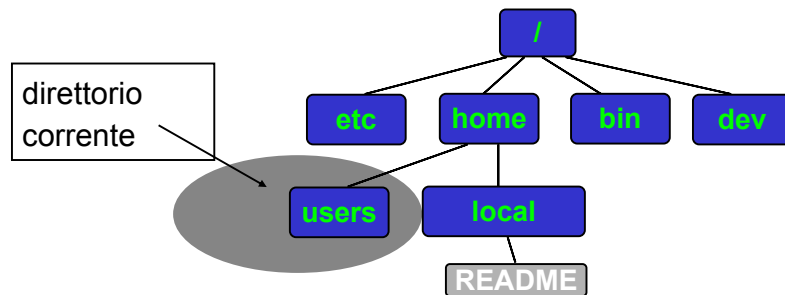
Nomi particolari

- . è il direttorio corrente (visualizzato da **pwd**)
- .. è il direttorio 'padre'
- ~ è la propria home utente

Il comando **cd** **permette di spostarsi all'interno del file system**, utilizzando sia nomi relativi che assoluti

- **cd** senza parametri porta alla home dell'utente

Nomi relativi/assoluti: esempio



nome assoluto: /home/local/README

nome relativo: ../local/README

Link

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

- SO considera e gestisce la molteplicità possibile di riferimenti:
 - 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]`

Gestione file: comando `ls`

consente di **visualizzare nomi di file**

- varie opzioni: esempio `ls -l` per avere più informazioni (non solo il nome del file)
- possibilità di usare **metacaratteri (wildcard)**
 - Per es. se esistono i file `f1`, `f2`, `f3`, `f4`
 - ci si può riferire a essi scrivendo: `f*`
 - o più precisamente `f [1-4]`

Più avanti studieremo meglio i metacaratteri e le modalità con cui vengono gestiti esattamente dalla shell

opzioni del comando `ls`...

□ `ls [-opzioni...] [file...]`

Alcune opzioni

- **l** (long format): per ogni file una linea che contiene **diritti**, **numero di link**, **proprietario** del file, **gruppo** del proprietario, **occupazione di disco** (blocchi), **data e ora** dell'ultima modifica o dell'ultimo accesso e **nome**
- **t** (time): la lista è **ordinata per data** dell'ultima modifica
- **u**: la lista è ordinata per data dell'ultimo accesso
- **r** (reverse order): inverte l'ordine
- **a** (all files): fornisce una **lista completa** (normalmente i file il cui nome comincia con il punto non vengono visualizzati)
- **F** (classify): indica anche il tipo di file (eseguibile: `*`, directory: `/`, link simbolico: `@`, FIFO: `|`, socket: `=`, niente per file regolari)

Comandi vari di gestione

- **Creazione/gestione di directory**
 - **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*

Comando shell ps

Un processo utente in genere viene attivato a partire da un comando (da cui prende il nome). Ad es., dopo aver mandato in esecuzione il comando **hw**, verrà visualizzato un processo dal nome **hw**.

Tramite ps si può vedere la lista dei processi attivi

```
pbellavis@lab3-linux:~$ ps
  PID TTY STAT TIME COMMAND
 4837 p2  S  0:00 -bash
 6945 p2  S  0:00 sleep 5s
 6948 p2  R  0:00 ps
```

Comando **ps** molto utile quando si lancia **l'esecuzione di programmi di sistema con errori di programmazione** (guardare su man le varie opzioni **ps**)

Terminazione forzata di un processo

È possibile 'terminare forzatamente' un processo tramite il comando **kill**

Ad esempio:

- **kill -9 <PID>** provoca l'invio di un segnale **SIGKILL** (forza la terminazione del processo che lo riceve e non può essere ignorato) **al processo identificato da PID**
 - Esempio: **kill -9 6944**

per conoscere il PID di un determinato processo, si può utilizzare il comando **ps**

monitor dei processi: comando top

```
 4:20pm up 3 days, 6:48, 1 user, load average: 0.00, 0.00, 0.00
84 processes: 83 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  0.7% user,  0.3% system,  0.0% nice, 99.0% idle
Mem: 127840K av, 122660K used,  5180K free,  37088K shrd, 10024K buff
Swap: 130404K av,  16K used, 130388K free, 100040K cached

  PID USER   PRI  NI  SIZE  RSS  SHARE STAT   LIB  %CPU  %MEM  TIME COMMAND
 28704 ptorroni 12   0   748   748   568 R    0  0.9  0.5  0:02 top
 28208 root     2   0  1084  1084   732 S    0  0.1  0.8  0:00 sshd
    1 root     0   0   432   432   356 S    0  0.0  0.3  0:01 init
    2 root     0   0     0     0     0 SW   0  0.0  0.0  0:00 kflushd
    3 root    -12 -12     0     0     0 SW<  0  0.0  0.0  0:00 kswapd
    4 root     0   0     0     0     0 SW   0  0.0  0.0  0:00 md_thread
    5 root     0   0     0     0     0 SW   0  0.0  0.0  0:00 md_thread
   495 root     0   0  4660  4660  1612 S    0  0.0  3.6  0:41 X
   621 root     0   0   320   320   264 S    0  0.0  0.2  0:00 mingetty
 28748 ptorroni  8   0   280   280   236 S    0  0.0  0.2  0:00 sleep
    31 root     5   0   376   376   324 S    0  0.0  0.2  0:00 kerneld
   194 root     0   0   596   596   496 S    0  0.0  0.4  0:01 syslogd
   203 root     0   0   536   536   336 S    0  0.0  0.4  0:00 klogd
   214 daemon   0   0   416   416   340 S    0  0.0  0.3  0:00 atd
   225 root     1   0   484   472   440 S    0  0.0  0.3  0:00 crond
   226 root     0   0   416   416   326 S    0  0.0  0.3  0:13 cron
```


Protezione dei file

- Molti utenti
 - Necessità di **regolare gli accessi** alle informazioni
- Per un file, esistono 3 tipi di utilizzatori:
 - proprietario, **user**
 - 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)** (per una directory significa list del contenuto)
- Ogni file è marcato con
 - **User-ID e Group-ID del proprietario**
 - **12 bit di protezione**

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			Group			Others		
PERMESSI											

Sticky bit

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

SUID e SGID

- SUID (Set User ID) (identificatore di utente effettivo)
 - Si applica a un file di **programma eseguibile solamente**
 - 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)**
 - È necessario per consentire 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 SUID lo rende possibile
- SGID bit: come SUID bit, per il gruppo

Protezione e diritti su file

Per variare i bit di protezione:

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

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

Esempi 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			Group			Others		

- `chmod u-w fileimportante`

Altri comandi:

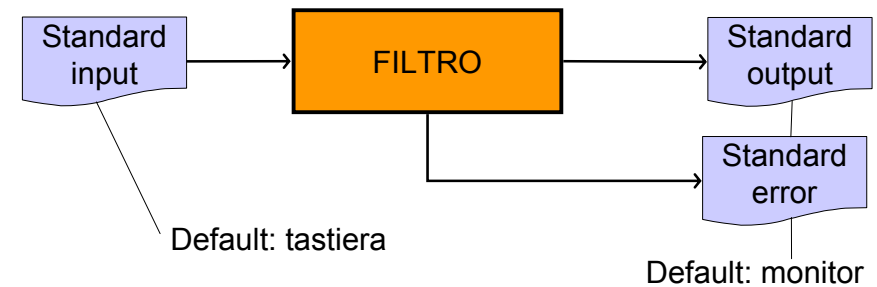
- `chown <nomeutente> <nomefile>`
- `chgrp <nomegruppo> <nomefile>`

Comandi, piping e ridirezione

Comandi e input/output

I **comandi UNIX** si comportano come **FILTRI**

- filtro è un programma che riceve il suo **ingresso da standard input** e produce il **risultato su standard output** (trasformazione di dati)



Comandi shell Linux: filtri

Alcuni esempi:

- `grep <testo> [<file>...]`
Ricerca di testo. Input: (lista di) file. Output: video
- `tee <file>`
Scrive l'input sia su file, sia sul canale di output
- `sort [<file>...]`
Ordina alfabeticamente le linee. Input: (lista di) file.
Output: video
- `rev <file>`
Inverte l'ordine delle linee di file. Output: video
- `cut [-options] <file>`
Seleziona colonne da file. Output: video

Ridirezione di input e output

Possibile ridirigere input e/o output di un comando facendo sì che non si legga da stdin (e/o non si scriva su stdout) **ma da file**

- **senza cambiare il comando**
- **completa omogeneità tra dispositivi e file**
- Ridirezione dell'input
 - `comando < file_input` Aperto in lettura
- Ridirezione dell'output
 - `comando > file_output` Aperto in scrittura (nuovo o sovrascritto)
 - `comando >> file_output` Scrittura in append

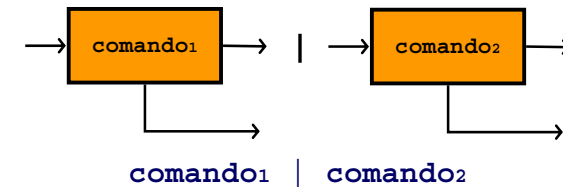
Esempi

- `ls -l > file`
File conterrà il risultato di `ls -l`
- `sort < file > file2`
Ordina il contenuto di `file` scrivendo il risultato su `file2`
- Cosa succede con `> file ?`

piping

L'output di un comando può esser diretto a diventare l'input di un altro comando (piping)

- In DOS: *realizzazione con file temporanei* (primo comando scrive sul file temporaneo, secondo legge da questo)
- In UNIX: *pipe come costruito parallelo* (l'output del primo comando viene reso disponibile al secondo e consumato appena possibile, non ci sono file temporanei)
- Si realizza con il carattere speciale '|'



Esempi di piping

- `who | wc -l`
 - Conta gli utenti collegati
- `ls -l | grep ^d | rev | cut -d' ' -f1 | rev`
 - Che cosa fa? Semplicemente mostra i nomi dei sottodirettori della directory corrente
 - `ls -l` lista i file del direttorio corrente
 - `grep` filtra le righe che cominciano con la lettera d (pattern `^d`, vedere il `man`)
 - ovvero le directory (il primo carattere rappresenta il tipo di file)
 - `rev` rovescia l'output di `grep`
 - `cut` taglia la prima colonna dell'output passato da `rev`, considerando lo spazio come delimitatore (vedi `man`)
 - quindi, poiché `rev` ha rovesciato righe prodotte da `ls -l`, estrae il nome dei direttori 'al contrario'
 - `rev` raddrizza i nomi dei direttori

Suggerimento: aggiungere i comandi uno alla volta (per vedere cosa viene prodotto in output da ogni pezzo della pipe)

Metacaratteri ed espansione

Metacaratteri

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
- [zfc] un qualunque carattere, in un nome di file, compreso tra quelli nell'insieme. Anche **range** di valori: [a-d]
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)

Esempi con metacaratteri

```
ls [a-p,1-7] * [c, f, d] ?
```

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

```
ls *\**
```

- Elenca i file che contengono, in qualunque posizione, il carattere *

Variabili nella shell

In ogni shell è possibile **definire un insieme di variabili** (trattate come stringhe) con **nome e valore**

- i riferimenti ai **valori delle variabili** si fanno con il **carattere speciale \$** (\$nomevariabile)
- si possono fare **assegnamenti**
nomevariabile=\$nomevariabile
l-value r-value

Esempi

- X=2
- echo \$X (visualizza 2)
- echo \$PATH (mostra il contenuto della variabile PATH)
- PATH=/usr/local/bin:\$PATH (aggiunge la directory /usr/local/bin alle directory del path di default)

Ambiente di esecuzione

Ogni comando esegue **nell'ambiente associato (insieme di variabili di ambiente definite) alla shell** che esegue il comando

- ogni shell **eredita l'ambiente dalla shell** che l'ha creata
- nell'ambiente ci sono **variabili** alle quali il comando può fare riferimento:
 - **variabili con significato standard:** PATH, USER, TERM, ...)
 - **variabili user-defined**

Variabili

Per vedere tutte le variabili di ambiente e i valori loro associati si può utilizzare il comando **set**:

```
BASH=/usr/bin/bash
HOME=/space/home/wwwliah/ww
PATH=/usr/local/bin:/usr/bin:/bin
PPID=7497
PWD=/home/Staff/PaoloBellavista
SHELL=/usr/bin/bash
TERM=xterm
UID=1015
USER=pbellavis
```

Espressioni

Le **variabili shell sono stringhe**. È comunque possibile **forzare l'interpretazione numerica** di stringhe che contengono la codifica di valori numerici

□ comando **expr**:

```
expr 1 + 3
```

Esempio:

```
echo risultato: var+1
```

var+1 è il risultato della corrispondente espressione?

```
echo risultato: `expr $var + 1`
```

a che cosa serve?

Esempio

```
#!/bin/bash
A=5
B=8
echo A=$A, B=$B
C=`expr $A + $B`
echo C=$C
```

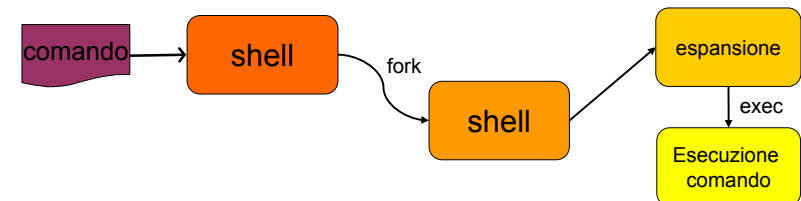
file *somma*

```
bash-2.05$ somma
A=5, B=8
C=13
```

invocazione

output

Espansione



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

- La shell **prima prepara i comandi come filtri**: ridirezione e piping di ingresso uscita
- Nelle successive scansioni, se shell trova altri caratteri speciali, **produce delle sostituzioni (passo di espansione)**

Passi di sostituzione

Sequenza dei passi di sostituzione

1) Sostituzione dei comandi

- comandi contenuti tra `` `` (**backquote**) sono eseguiti e sostituiti dal risultato prodotto

2) Sostituzione delle variabili e dei parametri

- **nomi delle variabili** (`$nome`) sono espansi nei valori corrispondenti

3) Sostituzione dei metacaratteri in nomi di file

- metacaratteri `*` `?` `[]` sono espansi nei **nomi di file** secondo un meccanismo di **pattern matching**

Inibizione dell'espansione

In alcuni casi è necessario **privare i caratteri speciali del loro significato**, considerandoli come caratteri normali

- `\` carattere successivo è considerato come un normale carattere
- `` `` (apici): proteggono da qualsiasi tipo di espansione
- `" "` (doppi apici) proteggono dalle espansioni con l'eccezione di `$` `\` `` `` (**backquote**)

Esempi sull'espansione

- `rm `*$var`*`
 - Rimuove i file che cominciano con `*$var`
- `rm `"$var"`*`
 - Rimuove i file che cominciano con `*<contenuto della variabile var>`
- `host203-31:~ paolo$ echo "<`pwd`>"`
`</Users/paolo>`
- `host203-31:~ paolo$ echo '`pwd`>'`
`<`pwd`>`
- `A=1+2 B=`expr 1 + 2``
 - In A viene memorizzata la stringa `1+2`, in B la stringa `3` (`expr` forza la valutazione aritmetica della stringa passata come argomento)

Riassumendo: passi successivi del parsing della shell

R ridirezione dell'input/output

```
echo hello > file1 # crea file1 e  
# collega a file1 lo stdout di echo
```

1. sostituzione dei comandi (backquote)

```
`pwd` → /temp
```

2. sostituzione di variabili e parametri

```
$HOME → /home/staff/pbellavis
```

3. sostituzione di metacaratteri

```
plu?o* → plutone
```

Scripting: realizzazione file comandi

File comandi

Shell è un **processore comandi** in grado di interpretare **file sorgenti in formato testo e contenenti comandi** → **file comandi (script)**

Linguaggio comandi (vero e proprio linguaggio programmazione)

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

NB:

- **quali statement** sono disponibili dipende da **quale shell** si utilizza
- file comandi viene **interpretato** (non esiste una fase di compilazione)
- file **comandi deve essere eseguibile** (usare `chmod`)

Scelta della shell

File comandi

La prima riga di un file comandi deve specificare **quale shell si vuole utilizzare**: `#! <shell voluta>`

- Es: `#!/bin/bash`
- `#` è visto dalla shell come un commento ma...
- `#!` è visto da SO come identificatore di un file di script
 - SO capisce così che l'interprete per questo script sarà `/bin/bash`
- Se questa riga è assente viene scelta la shell di preferenza dell'utente

È possibile memorizzare **sequenze di comandi all'interno di file eseguibili**:

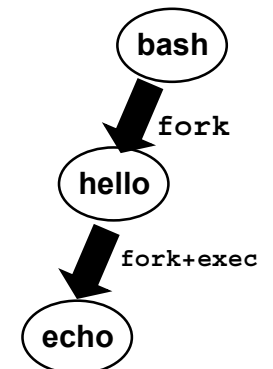
file comandi (script)

Ad esempio:

```
#!/bin/bash  
echo hello world!
```

file hello

```
bash-2.05$ hello  
hello world!
```



Passaggio parametri

```
./nomefilecomandi arg1 arg2 ... argN
```

Gli argomenti sono **variabili posizionali** nella linea di invocazione contenute nell'ambiente della shell

- **\$0** rappresenta il comando stesso
- **\$1** rappresenta il primo argomento ...
- è possibile far scorrere tutti gli argomenti verso sinistra
→ **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 informazioni utili

Oltre agli argomenti di invocazione del comando

- **\$*** insieme di **tutte le variabili posizionali**, che corrispondono arg del comando: \$1, \$2, ecc.
- **\$#** **numero di argomenti** passati (**\$0 escluso**)
- **\$?** valore (int) restituito dall'ultimo comando eseguito
- **\$\$** id numerico del processo in esecuzione (pid)

Semplici forme di 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

Strutture di controllo

Ogni comando in uscita restituisce un **valore di stato**, che indica il suo **completamento o fallimento**

Tale valore di uscita è posto nella variabile ?

- **\$?** può essere riutilizzato in espressioni o per controllo di flusso successivo

Stato vale usualmente:

- zero: comando OK
- valore positivo: errore

Esempio

```
host203-31:~ paolo$ cp a.com b.com
cp: cannot access a.com
host203-31:~ paolo$ echo $?
2
```

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 linguaggio C!

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

Alcuni tipi di test

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 a** `=` (o `!=`) sono *necessari*
- `stringa1` e `stringa2` possono contenere metacaratteri (attenzione alle espansioni)
- `test -z <stringa>` vero se *stringa nulla*
- `test <stringa>` vero se *stringa non nulla*

Strutture di controllo: alternativa

```
if <lista-comandi>
then
    <comandi>
[elif <lista_comandi>
then <comandi>]
[else <comandi>]
fi
```

ATTENZIONE:

- le parole chiave (`do`, `then`, `fi`, ...) 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 "sì" e un numero
# < 24
if test $1 = sì -a $2 -le 24
then echo sì
else echo no
fi

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

Alternativa multipla

```
# alternativa multipla sul valore di var
case <var> in
    <pattern-1>
        <comandi>;;
    ...
    <pattern-i> | <pattern-j> | <pattern-k>
        <comandi>;;
    ...
    <pattern-n>
        <comandi> ;;
esac
```

Importante: nell'alternativa multipla si possono usare metacaratteri per fare **pattern-matching** (non sono i "soliti" metacaratteri su nome di file)

Esempi

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

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

Cicli enumerativi

```
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**
- scrivendo solo **for i** si itera con valori di **i in \$***

Esempi

- **for i in ***
 - esegue per tutti i file nel direttorio corrente
- **for i in `ls s*`**

```
do <comandi>
done
```
- **for i in `cat file1`**

```
do <comandi per ogni parola del file file1>
done
```
- **for i in 0 1 2 3 4 5 6**

```
do
  echo $i
done
```

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)

```
until <lista-comandi>
do
  <comandi>
done
```

Come while, ma inverte la condizione

Uscite anomale

- vedi C: **continue**, **break** e **return**
- **exit [status]**: system call di UNIX, anche comando di shell

Esempi di file comandi (1)

```
echo 1 > loop.$$tmp

while :
do
  sleep 5s
  if [ `ls $1 | wc -w` -ne `cat loop.$$tmp` ]
  then
    ls $1 | wc -w > loop.$$tmp
    echo in $1 sono presenti `cat
      loop.$$tmp` file
  fi
done
```

Esempi di file comandi (1)

```
echo `pwd` > "f1>"
# R: crea il file di nome f1>, poi stdout_echo= f1>; echo `pwd`
# 1:      echo /usr/bin
# 2:      nessuna operazione ulteriore di parsing
# 3:      nessuna operazione ulteriore di parsing

test -f `pwd`/$2 -a -d "$HOME/dir?"
# R:      nessuna operazione di parsing
# 1:      test -f /temp/$2 -a -d "$HOME/dir?"
# 2:      test -f /temp/pluto -a -d "/home/staff/
pbellavis/dir?"
# 3:      nessuna operazione ulteriore di parsing
test -f /temp/pluto -a -d /home/staff/
pbellavis/dir?
```

Esempi di file comandi (2)

Esercizio da svolgere in lab (o a casa):

- scrivere un file comandi che ogni 5 secondi controlli se sono stati **creati o eliminati file in una directory**. In caso di cambiamento, si deve visualizzare un messaggio su stdout (quanti file sono presenti nella directory)
- il file comandi deve poter essere invocato con **uno e un solo parametro**, la directory da porre sotto osservazione (→ fare opportuno controllo dei parametri)

Suggerimento: uso di un file temporaneo, in cui tenere traccia del numero di file presenti al controllo precedente