La Shell
Clueless at point

Sergio Ballestrero Leandro Noferini

18 giugno 2002

Indice

1 Motivazioni

1.1 Perché cominciare dalla linea di comando

Chi di voi è abituato ad utilizzare un sistema Windows, e sa che anche con Linux esiste una interfaccia grafica, probabilmente si sta chiedendo che ragione ci sia, in pieno 2000, con le moderne CPU, per utilizzare qualcosa di così retrò, complicato e poco intuitivo come la linea di comando, la modalità testo. Ho alcune risposte per voi:

1. **potenza**. I comandi Unix hanno quasi 30 anni di storia alle spalle. Centinaia di persone hanno lavorato per migliorarli - non c'è praticamente nulla che non si possa fare.

   Provate ad aprire in Word un file di 100 MB per cercare una parola; poi provate:

   ```
   grep parola file
   ```

2. **semplicità**. Ogni comando esegue operazioni semplici - non dovete combattere per trovare dei menù nascosti chissà dove fra altri diecimila , o delle opzioni di configurazione magari inaccessibili se non cambiando chissà quale file .ini o voce del Register. È sufficiente che applichiate una sola regola: *RTFM - Read The Fucking Manual*. Basta `man comando` (vedi la sezione 3.13).

3. **flessibilità**. Fra i programmi GUI - Graphical User Interface - e i programmi da linea di comando, c'è la stessa differenza che c'è fra un trapano multifunzione da hobbista, e l'attrezzatura di frese, troncatrici, pialle elettriche di un professionista. Sì, probabilmente riuscite a rabberciare qualcosa anche con il primo, ma se volete fare sul serio, velocemente e bene, servono le cose serie.

   Avete tanti strumenti e li potete combinare come volete, e fare quello che vi serve - non quello che un programmatore ha un giorno pensato che vi sarebbe servito.

4. **velocità**. Se non siete completamente inabili alla tastiera, scrivere pochi caratteri è sicuramente più veloce che trascinare un mouse fra infiniti menu e finestre di dialogo. E questo è il meno.

   Sono soprattutto le operazioni ripetitive che diventano molto più veloci, tanto più quando comincerete ad apprezzare e imparare la possibilità di programmare la linea di comando.

   Per di più, rinunciare all'interfaccia grafica significa che basta poca memoria, poca velocità di CPU. Sotto Linux, un 486 con 8MB può fare un ottimo servizio come server Web, email, file server e un sacco di altre cose. Se invece provate a far partire l'interfaccia grafica, diventerà un sistema inutilizzabile...

5. **accessibilità**. Provate a eseguire un programma su un PC a 10.000 Km di distanza, attraverso un'Internet affollata come sempre.

   O semplicemente ad accedere al vostro account su un server, da un PC di qualcun'altro su cui non avete quel bel programmino di accesso da remoto. Il telnet è (quasi) sempre disponibile (e anche quando non lo fosse ci sono sostituti assai migliori).

6. **sicurezza**. Bene, cominciamo così: installate una nuovo programma su Windows. Riavviate. Crash. E adesso? Scheda grafica preistorica e sconosciuta - niente driver video. E adesso?

Quando nient'altro funziona, la riga di comando è sempre (beh, quasi sempre) disponibile per risolvere i vostri problemi. Al limite potete far partire il sistema con uno o due floppy[1] sui quali troverete un sistema sicuramente testuale.

7. **disponibilità di programmi**. In effetti per Linux, e anche per tutti gli altri Unix, usando l'interfaccia testuale è possibile fare tutto, ma veramente tutto: leggere la posta elettronica e i newsgroups, visualizzare siti web, usare l'ftp, ascoltare cd, gli mp3, giocare. Tutto ma veramente tutto.

   Inoltre tutti questi programmi hanno uno sviluppo notevole anche adesso, nell'era delle interfacce grafiche, e quindi sono spesso allo stato dell'arte.

1.2 Filosofia

Il sistema operativo Unix nasce (dal 1969 ai Bell Labs della AT&T) per riutilizzare vecchi e piccoli computer (DEC PDP-7 e PDP-11). Per questo nasce con uno spirito minimalista - questo spirito, che certo non si vede immediatamente nella complicazione degli attuali sistemi Unix (nella configurazione, nelle applicazioni X11), è però evidente lavorando dalla linea di comando (terminale testuale).

Molti di voi probabilmente conosceranno la linea di comando dell'MS/DOS, quindi alcune cose vi suoneranno familiari, o addirittura scontate. Beh, quando fu introdotto Unix, queste cose erano rivoluzionarie, e l'MS/DOS ha pescato a piene mani nelle idee base di Unix: sotto tanti punti di vista, lo si può considerare il parente povero di Unix.

Sono quattro i punti fondamentali della filosofia Unix:

1. **comandi semplici**. Mmolti comandi, molto specializzati

   ```
   [sb@pcsash sb]$ wc /etc/passwd
   ```

2. **pipeline**. L'output di un comando può essere utilizzato come input di un altro.

   Questo sistema si chiama pipeline: un *tubo* attraverso cui passa un flusso di dati, che viene modificato dai diversi comandi che si trovano lungo il tubo.

   ```
   cat /etc/passwd|grep o|sort
   ```

3. **opzioni omogenee**. Le opzioni, specificate con `-x` o `--xxxx`, cercano di essere il più possibile omogenee, cioè di avere lo stesso significato, fra i diversi comandi:

   - `cat --help`
   - `grep --help`

4. **file omogenei**. Tutti i file sono uguali. Questo non è niente di nuovo se siete abituati al DOS, ma qualcuno di voi ricorderà la gestione file sui dischi del Commodore64 - o forse no? :-). E soprattutto, questo è molto diverso rispetto a come erano i sistemi precedenti.

   Unix, poi, fa un passo oltre: non solo i file sono uguali, ma anche tutte le periferiche sono file, e quindi uguali. Si può scrivere allo stesso modo sullo schermo, su un file, sui settori fisici di un floppy, sul modem o la stampante - potete persino provare a scrivere su uno scanner o sulla scheda audio - ma il fatto che una cosa si possa fare, non significa necessariamente che abbia un senso farla... *e neanche che non sia rischioso!*

[1] Ad esempio usando una delle tante distribuzioni pensate appositamente per casi d'emergenza come questo. Queste distribuzioni sono Linux assolutamente completi sui quali però trovate solo i programmi che possono aiutare a recuperare dischi guasti o errori di formattazione.

2 Primi passi

Iniziamo allora a prendere confidenza con la shell.

2.1 Login

Quando entrate nel sistema a linea di comando, potreste trovarvi di fronte ad un bello schermo nero, stile terminale anni 70, a cui rispondere, con un risultato del tipo:

```
Red Hat Linux release 5.1 (Manhattan)
Kernel 2.0.35 on an i586
login: sb
Password: Cappuccino
Last login: Sat Nov 21 10:12:48 on tty2
Il System Manager ti saluta
You have mail.
[sb@pcsash sb]$
```

Questo significa e che vi identificate e autenticate con il sistema con un dialogo del tipo: «chi va là?», «sono quello che tu chiami sb», «parola d'ordine?» «Cappuccino»[2]

Il sistema (in realtà un ben preciso programma che, guarda il caso, si chiama login) controlla che esista un utente con il nome dato (username, non nome effettivo), e che la password corrisponda a quella registrata nel file `/etc/passwd`.

In verità le cose sono leggermente più sofisticate perché questo sistema aveva gravi problemi di sicurezza. Da molto tempo ormai è in uso il sistema delle password shadow, con il quale semplicemente le password di tutti gli utenti vengono registrate in un file differente da `/etc/passwd` (che poi sarebbe `/etc/shadow`). Come si vede dall'esempio, il secondo campo, quello dopo il nome dell'utente, viene sostituito da una x.

Le cose poi possono essere ancora più complicate nel caso di uso della rete ma l'argomento esula ampiamente dallo scopo di questa lezione. Esempio dal mio portatile:

```
[leandro: ~]$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:100:sync:/bin:/bin/sync
games:x:5:100:games:/usr/games:/bin/sh
man:x:6:100:man:/var/catman:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/spool/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
majordom:x:30:31:Majordomo:/usr/lib/majordomo:/bin/sh
postgres:x:31:32:postgres:/var/postgres:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
msql:x:36:36:Mini SQL Database Manager:/var/lib/msql:/bin/sh
operator:x:37:37:Operator:/var:/bin/sh
```

[2] In verità la password non apparirà mentre la scriverete, anche questo per questioni di sicurezza, per evitare il classico tipo che sbircia da sopra le spalle di chi scrive.

```
list:x:38:38:SmartList:/var/list:/bin/sh
irc:x:39:39:ircd:/var:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats/gnats-db:/bin/sh
alias:x:70:65534:qmail alias:/var/qmail/alias:/bin/sh
qmaild:x:71:65534:qmail daemon:/var/qmail:/bin/sh
qmails:x:72:70:qmail send:/var/qmail:/bin/sh
qmailr:x:73:70:qmail remote:/var/qmail:/bin/sh
qmailq:x:74:70:qmail queue:/var/qmail:/bin/sh
qmaill:x:75:65534:qmail log:/var/qmail:/bin/sh
qmailp:x:76:65534:qmail pw:/var/qmail:/bin/sh
nobody:x:65534:65534:nobody:/home:/bin/sh
leandro:x:1000:100:Leandro su nerone,,,:/opt/leandro:/bin/bash
ospite:x:1002:100:Ospite di Nerone,,,:/home/ospite:/bin/bash
bbs:x:1003:100:Utente BBs,,,:/opt/daydream:/bin/bash
postfix:x:100:65534::/etc/postfix/:/bin/false
ftn:x:64000:64000:Ifmail su nerone,,,:/etc/ifmail:/bin/bash
cgabriel:x:1001:100:Christopher su Nerone,,,:/home/cgabriel:/bin/bash
mac:x:1004:100:Utente Mac,,,:/home/mac:/bin/bash
gdm:x:101:101::/var/gdm/:/bin/false
[leandro: ~]$
```

A questo punto, avete una *identità* [3] : lo username, a cui corrisponde uno userid (o uid) numerico; e appartenente ad alcuni gruppi, a cui corrispondono dei groupid (o gid) numerici.

```
[sb@pcsash sb]$ whoami
sb
[sb@pcsash sb]$ id
uid=601(sb) gid=601(sb) groups=601(sb),100(users),10(wheel),503(tape)
```

L'idea di identità sarà forse familiare a chi di voi usa una rete, e certo a chi usa Internet: molto probabilmente il vostro indirizzo di email è un account (con diritti anche molto limitati) su un server Unix. Per la strada ne scopriremo il significato.

2.2 Prompt e shell

Consideriamo l'esempio seguente; questo è quello che viene chiamato *command line shell prompt*, o semplicemente *prompt* o varie altre abbreviazioni. Cioè *sono pronto a eseguire i comandi.* Ma chi è pronto, e ad eseguire quali comandi? e può fare altro? e perché il prompt non è sempre uguale?

```
[sb@pcsash sb]$ _
```

Dopo l'autenticazione, il sistema passa ad eseguire, con la vostra identità, un programma specificato nel file `/etc/passwd` (per l'esattezza quello indicato dell'ultimo campo, vedi l'esempio nella sezione 2). Se il vostro è un normale account Unix, questo programma sarà del genere chiamato *shell di comando.* In altri casi, potrebbe essere una *shell ristretta*, un programma di BBS, o niente del tutto (come nel caso degli account di posta presso un provider, il `/bin/false` dell'esempio).

[3] Questa identità spesso viene chiamata *account*: conto nel senso di conto bancario o simili - il nome deriva dai tempi in cui i grandi computer venivano utilizzati in batch o in time sharing, e agli utilizzatori veniva fatto pagare in proporzione a quanto *tempo di CPU* veniva utilizzato dall'utente - quindi a ogni utente corrispondeva appunto un conto a cui addebitare il tempo di utilizzo.

Un programma shell è una interfaccia fra voi e gli altri programmi presenti nel sistema. Un esempio di shell che probabilmente tutti conoscete è il `command.com` del DOS[4].

Sotto Unix, la scelta della shell è una questione strettamente personale, e più in là avrete modo di provare altre shell, e decidere quale preferite; io vi parlerò di bash [5], che per Linux è la shell di default, e probabilmente la più potente e flessibile.

Un elenco delle varie shell è il seguente:

- ***Bourne shell e derivate*** Derivano dalla shell usata nelle prime versioni di Unix. Ormai non viene più utilizzata.
  - **Bourne shell**. La prima shell di Unix. In Linux è sostituita da bash. Sui sistemi standard POSIX, è sostituita da ksh.
  - **ash**. Una shell *minima*, con molti comandi integrati, occupa poca RAM e poco spazio disco, anche perché ha poche funzioni. Usata talvolta nei dischetti di installazione o recovery, può essere utile per gli script perché più veloce di bash.
  - **ksh - Korn Shell** La Korn shell (dal nome dell'autore) introduce la history (accesso ai comandi precedenti) e l'editing della linea. Poco usata in Linux - e in effetti non ce n'è ragione, dato che bash ne ha tutte le caratteristiche; è però utile conoscerne l'esistenza dato che è facile trovarla su altri Unix nei quali è una accettabile sostituita di bash che solitamente manca.
  - **zsh**. Offre praticamente le stesse funzioni di ksh, e alcune altre cose esoteriche, che però ovviamente trovate anche in bash.
  - **bash - GNU Bourne-Again SHell**. Incorpora le caratteristiche di ksh e csh, command line completion, e altro; si vocifera (come per altro a proposito di moltissimi altri programmi di Unix) che esista una opzione per fargli preparare il caffè.

    Dalla man page: *BUGS: It's too big and too slow..*
- ***C shell e derivate*** Così chiamate perché usano una sintassi che ricorda la sintassi usata dal linguaggio di programmazione C.
  - **csh - C shell**. Utilizza una sintassi derivata dal linguaggio C. In Linux è sostituita da tcsh.
  - **tcsh**. Evoluzione della precedente, alla quale aggiunge history e editing di linea.

Quale shell scegliere? Su Linux, Bash senza dubbio, a meno che non siate, per arcane ragioni, affezionati a tcsh (se lo siete, è strano che stiate leggendo questi appunti). Su altri Unix, ksh o tcsh, che offrono l'editing di linea. tcsh è più usata, e quindi è più facile che sia già stata ben configurata dal system manager; ksh spesso è poco usata, e quindi potreste dover configurare l'uso dei tasti cursore e editing, cosa molto noiosa.

2.3 Caratteristiche comuni

Il prompt, quindi, è il segnale che il programma shell vi invia per avvisarvi che sta aspettando i vostri comandi. I comandi sono solitamente il nome di un programma da eseguire, e i relativi parametri. La shell quindi cerca il programma e lo lancia, passandogli i parametri dati.

Ma la vera funzione della shell è quella di semplificarvi la vita; per questo vi permette diverse cose:

[4] Il cui primo comandamento era *non avrai altra shell al di fuori di command.com*. Qualcuno di voi forse conoscerà 4DOS, e saprà quindi che si poteva infrangere quel comandamento.

[5] BASh sta per Bourne Again Shell, che è un gioco di parole fra Bourne Shell, storicamente la prima shell di Unix, e il fatto che è Bourne si pronuncia (quasi) come *born*: *la Bourne Shell rinata*.

- **path search**. Consente di non specificare l'identificativo completo del programma. La shell, se gli viene richiesto un nome di programma senza specificare la directory, lo cerca nel PATH (che è una variabile d'ambiente, vedi la sezione 2.3).

- **pipeline**. Permette di direzionare l'input e/o l'output di un programma, da/verso file o verso altri programmi.

  Come una catena di montaggio, un comando Unix ha del materiale in ingresso (*input*), delle operazioni svolte su questo materiale, e un prodotto in uscita (*output*). Come per una linea di montaggio, si possono applicare piccole variazioni alle operazioni, cambiando i giusti interruttori (*option switches*), ma quando serve qualcosa di abbastanza diverso, la cosa migliore è affidare il prodotto a un'altra linea di montaggio, che, partendo dal primo prodotto (l'output del primo comando) ottenga quello richiesto.

- **alias**. Permette di definire abbreviazioni per i comandi molto usati, o definire degli dei nomi per alcune combinazioni di opzioni specifiche; ad esempio si può definire:

  ```
  alias ll='ls --color=auto  -l'
  ```

  per avere una versione colorate e "verbosa" di `ls`.

- **filename globbing** Per operare su gruppi di file con nomi simili si possono usare dei *caratteri jolly*:

  ```
  [sb@pcsash sb]$ ls prova*1998.txt
  ```

  Come nel caso del DOS, il carattere * viene sostituito da uno o più caratteri mentre il carattere ? viene sempre sostituito da un solo carattere. Ma vengono anche supportati elenchi di possibili alternative (`prova[a,c,d].txt`) o range ((`prova[a-d].txt`).

  Infine l'uso del carattere tilde ~ per significare la directory home dell'utente o `~username` per la home di un'altro utente.

- **variabili di ambiente**. La shell si ricorda e vi permette di modificare certe variabili (per convenzione si indicano con parole maiuscole) che sono spesso usate anche per controllare le impostazioni usate dai programmi.

  Queste impostazioni sono memorizzate in alcune variabili come `$USER`, `$HOME`, `$TERM`, `$DISPLAY`, `$PATH` e altre (che per l'appunto cominciano tutte per `$`) che possono essere valide per tutti gli utenti (quando impostate in `/etc/profile`) e/o per un singolo utente, (quando impostate, se si usa la bash, nei file `~/.bash\_profile` e `~/.bashrc`).

- **variable expansion**. Espansione delle variabili. Oltre a quelle definite di default, si possono definire tutte le variabili che si vuole per poi riutilizzarle, ad esempio:

  ```
  [sb@pcsash sb]$ Saluto=Ciao
  [sb@pcsash sb]$ echo "$Saluto $USER, sei su $HOSTNAME"
  Ciao sb, sei su pcsash.sash.lan
  ```

  In questo caso il primo comando imposta la variabile Saluto al valore Ciao, mentre le altre due variabili, USER e HOSTNAME sono due variabili che vengono normalmente impostate in ogni sistema.

  Il comando `echo` invece mostra ciò che gli viene scritto, espandendo per l'appunto le variabili che eventualmente incontra.

- **command (backtick) expansion** Lo standard output di un comando può diventare una stringa che fa parte di una linea di comando:

```
[sb@pcsash sb]$ hn=`cat /etc/HOSTNAME`
[sb@pcsash sb]$ echo $hn
pcsash.sash.lan
```

- **history expansion**. Non solo la shell ricorda i comandi immessi in precedenza, accessibili tramite i tasti di freccia , ma è possibile sia effettuare un ricerca (con `^r`) su quanto richiamarle con l'uso del carattere `!`. Ad esempio per richiamare righe di comando precedenti si può fare qualcosa come:

```
[sb@pcsash sb]$ which tar
/bin/tar
[sb@pcsash sb]$ ls $(!wh)
ls $(which tar)
-rwxr-xr-x   1 root     root        90764 Apr 27  1998 /bin/tar
```

- **controllo di esecuzione**. Si possono eseguire processi in foreground (cioè sul terminale, con uno solo attivo alla volta) e in background (cioè in modalità non interattiva, e questi possono essere quanti si vuole), e passare dall'uno all'altro, selezionare l'identificativo di un processo, bloccare, riavviare, terminare un processo con i comandi: `&`, `%n`, `^c`, `^z`, `fg`, `bg`, `kill`.

- **scripting**. I *batch file* del DOS - ipervitaminizzati. Sono infatti disponibili costrutti complessi, come istruzioni di loop, condizionali, `case` ed un sacco di altre cose.

Il prompt della shell può essere diverso, e può contenere diverse informazioni. Nel caso della bash, ma anche per la maggior parte delle altre shell, è configurabile. Si comincia dal minimalista

.

a cose fin troppo complete come lo

```
[user@host: directory]$
```

degli esempi. La scelta dipende da cosa vi serve: se vi trovate a lavorare contemporaneamente con diverse identità, su computer diversi, allora vi servirà sapere *chi siete* in una certa finestra terminale; altrimenti potete farne a meno e risparmiare spazio utile sulla linea di comando.

Solitamente viene impostato per tutti in `/etc/profile`, tramite il valore della variabile `PS1`. Ovviamente ogni utente ne può modificare le impostazioni nel proprio `~/.bash\_profile`.

Quello che rimane uno standard è che un utente normale ha un prompt che finisce in `$`, e il superuser ha un prompt che finisce in `#` *a meno che non abbiate pasticciato con la definizione del prompt.*

2.4 Ma chi è il superuser

Una questione psicoanalitica ovvero: se volete sempre usare il superuser, avete un problema *e ne avrete ancora di più in seguito.*

Abbiamo visto che al login ci viene assegnata una identità. Abituati ai PC Win95, dove ai diversi login corrispondono, tutt'al più, diverse configurazioni, ci troviamo invece di fronte a un grosso cambiamento: diversi account hanno diversi diritti - per quanto riguarda l'accesso ai file, alle periferiche.

Esistono, su tutti i sistemi Unix, delle identità che non appartengono a persone reali, ma a certi ruoli. Ad esempio i programmi che gestiscono la posta sono eseguiti con una identità speciale, `mail`, che gli dà il diritto di scrivere nelle caselle di posta degli utenti.

Esiste un account che ha tutti i diritti su tutto, senza alcuna limitazione, ed è l'account del system manager o *superuser*: `root`.

L'utente `root` ha diritto di accesso, in lettura e scrittura, a qualsiasi cosa sul sistema: dalle configurazioni, alle periferiche, la posta di tutti gli utenti, persino ai programmi che sono in esecuzione. Il system manager ha poteri illimitati, e gli utenti devono avere illimitata fiducia nella sua affidabilità e discrezione.

Come utenti Linux, probabilmente sarete voi stessi i system manager del vostro PC. Non dimenticatevi mai che il superuser ha possibilità illimitate - e quindi capacità illimitata di causare danni. La prima regola, quindi, è: non fidatevi di voi stessi. Quando avete l'identità di root, pensate sempre due volte al comando che state scrivendo, o preparatevi a sopportarne le conseguenze.

Ad esempio, il comando:

```
# rm -rf /*
```

è sufficiente a cancellare qualsiasi file sulle vostre partizioni... e altri possono facilmente fare polpette di tutto l'harddisk. Dato da utente normale non avrebbe avuto alcun effetto.

Se avete appena installato il Linux, come prima contromisura, createvi un utente *normale* col vostro nome:

```
# adduser mario
```

e assegnategli una password:

```
# passwd mario
```

e usate sempre questo user. Usate root solo quando non ne potete fare a meno - per configurare il sistema, per installare del nuovo software.

La tentazione di usare sempre root è forte - soprattutto se in precedenza avete avuto la sfortuna di scontrarvi con un system manager che faceva il bello e il cattivo tempo; non fosse altro per la sensazione di potenza, o per contrastare la sensazione di impotenza (*ma come, non posso neanche scrivere un file in quella directory? ma è il MIO computer!* == paura della castrazione), ma è bene resistere più che si può, anche per evitare le fobie a livello di timor panico che si sviluppano dopo aver distrutto il sistema per la terza volta consecutiva...

3 Come lavorare

Una volta che ci è chiaro chi siamo, e quali sono le nostre possibilità di operare incominciamo a vedere cosa si può fare.

3.1 Dove siamo, cosa abbiamo d'intorno

Il nostro prompt, subito dopo il login,

```
[sb@pcsash sb]$
```

ci dice che siamo in una directory di nome sb. Quale è ce lo dice il comando `pwd` - *Present work directory*

```
[sb@pcsash sb]$ pwd
/home/sb
```

In un'altra lezione vi spiegheranno il misto di ragioni storiche e logiche per cui il file system di Unix, e quello di Linux in particolare, è organizzato così. Per ora, limitatevi a dimenticarvi dei dischi (A:, C: etc.), e a imparare che i nomi delle directory si separano con `/` (slash) e non con \ (backslash).[6]

La directory in cui ci troviamo dopo il login è la nostra, personale *home directory*. Questo, in pratica, significa che ci possiamo tenere i nostri dati, programmi e quant'altro. Ma per ora non abbiamo ancora fatto nulla.

Diamoci un'occhiata:

```
[sb@pcsash sb]$ ls
```

il comando `ls` - *LiSt files* è l'equivalente del dir del DOS, e ci ha appena detto che non c'è nulla.

Ma è proprio vero ?

```
[sb@pcsash sb]$ ls -la
total 12
drwxr-xr-x   3 sb      sb          1024 Jan  8  1998 .
drwxr-xr-x  32 root     root         3072 Nov 21 10:11 ..
-rw-r--r--   1 sb       sb           3768 Nov  7  1997 .Xdefaults
-rw-r--r--   1 sb       sb             24 Jul 14  1994 .bash_logout
-rw-r--r--   1 sb       sb            220 Aug 23  1995 .bash_profile
-rw-r--r--   1 sb       sb            124 Aug 23  1995 .bashrc
drwxr-xr-x   2 sb       sb           1024 Jan  8  1998 .xfm
```

no, ci sono un sacco di cose!

L'opzione `-l` (`long`) dice a `ls` di darci tutti i dettagli sui file che elenca, e l'opzione `-a` (*all*) gli dice di mostrare anche i file nascosti[7].

Tutti questi file sono file di configurazione, che vengono automaticamente copiati nella home directory dell'utente quando viene creato, prendendoli dalla directory `/etc/skel`. Questi sono file di configurazione personali, che possiamo cambiare senza essere superuser. A noi, più avanti, interesseranno `.bashrc`, `.bash_profile` e `.bash_logout`, che sono i file di configurazione di bash.

Proviamo a spostarci:

```
[sb@pcsash sb]$ cd /
[sb@pcsash /]$
```

Il comando `cd` - *Change Directory* serve a cambiare directory; anche il prompt è cambiato: controlliamo dove siamo

```
[sb@pcsash /]$ pwd
/
```

e guardiamo cosa c'è:

```
[sb@pcsash /]$ ls
-rw-r--r--   1 root     root        67902 Jul  7 12:53 System.map
drwxr-xr-x   2 root     root         2048 Oct 28 10:10 bin
drwxr-xr-x   4 root     root         1024 Oct 11 11:35 boot
```

[6] Se per caso qualcuno si chiedeva perché gli indirizzi HTTP usano le / (slash), adesso lo sa: perché il World Wide Web è nato (al CERN, nel 1989) su NeXt, un sistema derivato, di fondo, da Unix.

[7] *File nascosto*, in Unix, significa semplicemente che il nome del file inizia con un `.` - non esiste un attributo *nascosto* per i file, è semplicemente una convenzione, per cui la shell e altri programmi non prendono in considerazione questi file, a meno che non gli venga richiesto esplicitamente.

```
drwxr-xr-x   3 root     root        22528 Nov 21 10:11 dev
drwxr-xr-x  32 root     root         3072 Nov 21 10:11 etc
drwxr-xr-x   1 root     root         1024 Jul  9 13:27 home
drwxr-xr-x   4 root     root         2048 Oct 27 19:50 lib
drwxr-xr-x   2 root     root        12288 May  5  1998 lost+found
drwxr-xr-x   6 root     root         1024 Nov 11 09:21 mnt
dr-xr-xr-x   5 root     root            0 Nov 21 11:10 proc
drwxr-xr-x  19 root     root         2048 Nov 21 10:14 root
drwxr-xr-x   3 root     root         2048 Oct 15 11:29 sbin
drwxrwxrwt   5 root     root         3072 Nov 21 13:05 tmp
drwxr-xr-x  22 root     root         1024 Oct 27 19:50 usr
-rw-r--r--   1 root     root       304546 Jul  7 12:53 vmlinuz
```

(qualcuno prima o poi ve ne spiegherà il significato).

Adesso torniamo nella directory di partenza (home):

```
[sb@pcsash /]$ cd
[sb@pcsash sb]$
```

(se usiamo `cd` senza specificare nulla il comportamento di default è tornare alla home directory dell'utente). Poi ancora torniamo nella directory in cui eravamo prima:

```
[sb@pcsash sb]$ cd -
[sb@pcsash /]$
```

per caso, ho un dischetto DOS nel drive - guardiamo un po' ...

```
[sb@pcsash /]$ mdir
Volume in drive A has no label
Volume Serial Number is 14E4-1A2F
Directory for A:/

command  com     96212 05-05-1998  13:23
keybrd2  sys     31942 08-24-1996  11:11
aspicd   sys     30076 07-12-1993   3:02
...
ncedit   exe     63264 05-10-1993  18:52
sys      com     19255 08-24-1996  11:11
24 file(s)           786 606 bytes
385 536 bytes free
```

Il comando `mdir` - *msdos dir* è uno dei comandi del pacchetto mtools, che serve a lavorare con i dischi DOS. Dato che cercano di rispettare sia i canoni Unix, che le abitudini di chi lavora in DOS, gli mtools hanno diverse peculiarità, e comportamenti leggermente diversi dagli strumenti standard - fate attenzione[8].

3.2 Curiosare

Beh, se state leggendo queste pagine, vuol dire che siete curiosi. Allora, perché non andare a curiosare? Proviamo:

[8] Analoghi agli mtools per il dos esistono le hfsutils per accedere ai files prodotti da computer che usano il MacOS della Apple, con le complicazioni del caso, chiaramente.

```
[sb@pcsash sb]$ cat /System.map
... etc etc ...
001b4c20 A _end
```

Il comando `cat` - *conCATenate* scrive in uscita, a video in questo caso, il contenuto dei file indicati.

Chissà cos'era - era troppo lungo, ed è scorso via troppo velocemente; proviamo a guardarlo con calma:

```
[sb@pcsash sb]$ less /System.map
```

Il comando `less` - *più di more*[9] è un altro comando per *scorrere lentamente* il contenuto di un file.

Oh, adesso possiamo scorrerlo come vogliamo, tornando indietro, persino cercando delle parole scrivendo `/end` e molto altro: `h` per l'aiuto del programma e `q` per terminarlo.

A volte quello che si interessa è solo l'inizio di un file:

```
[sb@pcsash sb]$ head /System.map
```

o solo la fine:

```
[sb@pcsash sb]$ tail /System.map
```

o seguire cosa viene via via aggiunto ad un file:

```
[sb@pcsash sb]$ tail -f /var/log/messages
```

3.3 Creare

Dopo aver guardato dei file già fatti, proviamo a farne uno noi:

```
[sb@pcsash sb]$ touch fileprova
[sb@pcsash sb]$ ls -l fileprova
-rw-rw-r--   1 sb       sb              0 Nov 21 17:49 fileprova
[sb@pcsash sb]$ cat fileprova
[sb@pcsash sb]$
```

è vuoto.

Il comando `touch` - *tocca* - aggiorna la data di modifica di un file a quando questo comando viene eseguito. Nel caso invece questo non esista allora crea un file vuoto.

Proviamo a scriverci qualcosa...

```
[sb@pcsash sb]$ echo "che bella scritta" fileprova
che bella scritta fileprova
```

no, questo comando ha scritto sul terminale - il secondo argomento non è il nome di un file su cui scrivere, ma viene semplicemente attaccato al primo...

Il comando `echo` - *mostra* - l'argomento sullo standard output, cioè lo schermo.

Allora, approfittiamo della capacità della shell di redirezionare l'output di un programma:

```
[sb@pcsash sb]$ echo "che bella scritta" >fileprova
[sb@pcsash sb]$ cat fileprova
che bella scritta
```

aggiungiamo qualcosa:

[9] il quale more è un programma che esiste e che può essere usato ma che ha minori possibilità

```
[sb@pcsash sb]$ echo "costa solo $10" >>fileprova
[sb@pcsash sb]$ cat fileprova
che bella scritta
costa solo 0
```

Strano, no? No. Siamo inciampati in una variabile di shell e bash ne ha sostituito il valore nell'espressione (vedi 2.3). Ma possiamo impedirglielo, mettendo la stringa fra virgolette singole, che impediscono la *shell expansion*, o precedendo il carattere incriminato con un \ (backslash):

```
[sb@pcsash sb]$ echo 'costa solo $10 ' >>fileprova
[sb@pcsash sb]$ cat fileprova
che bella scritta
costa solo 0
costa solo $10
```

proviamo di nuovo:

```
[sb@pcsash sb]$ echo "ma che bella scritta ! "> a
bash: !": event not found"
```

Whoops! Cos'è successo? Stavolta il carattere ! ha chiesto a bash di cercare nella history un comando che cominciasse con i caratteri subito successivi, ma possiamo rimediare, come nel caso di `$`, o con le virgolette singole, o con un backslash davanti al `!`.

Comunque, a parte questi piccoli inconvenienti, abbiamo visto quello che volevamo: come usare la redirezione dell'output per scrivere in un file. Ci sono altre cose in proposito:

- con `cmd < file` si redireziona l'input
- con `cmd 2> file` si redireziona l'error output (stderr, per chi sa il C), che altrimenti viene scritto su video anche se l'output (stdout) è redirezionato.

  Molto utile quando quello che interessa sono gli errori...
- con `cmd > file 2>&1` si rimettono insieme i due in file.

  Attenzione che a causa del buffering indipendente fra i due, l'ordine delle righe può non corrispondere a quello che si avrebbe a video, senza la redirezione.
- con `cmd_a 2>&1 | cmd_b` si passa anche l'error output nella pipe - vedi sopra per i problemi.

3.4 Diritti utente

Benissimo, adesso avete il vostro file nuovo nuovo - ma avete bisogno che un altro utente ci possa scrivere: la soluzione è

```
[sb@pcsash sb]$ chmod g+w fileprova
```

A volte questo può non essere sufficiente (ad esempio nel caso di distribuzioni come la RedHat che hanno scelto il sistema del *personal group*), quindi è necessario anche provvedere a impostare il corretto *proprietario* di un file. Al possessore di un file è permesso cambiare il gruppo di appartenenza di un file, facendo un

```
[sb@pcsash sb]$ chgrp altrogruppo fileprova
```

ma solo root può cambiare il proprietario di un file, usando

```
[root@pcsash sb]# chown altrouser fileprova
```

A questo punto, mi sembra necessario ricordare che sono molto importanti le protezioni assegnate ad una directory (che si gestiscono con gli stessi comandi utilizzati per i file): in particolare, se assegnate ad un gruppo i diritti di scrittura su una directory, tutti gli utenti appartenenti a quel gruppo potranno non solo scrivere dei file in quella directory (che probabilmente è ciò che volete) ma anche cancellare qualsiasi file in quella directory (cosa che probabilmente *NON* volete). L'assegnazione dei diritti su un file o su una directory, quindi, in un sistema multiutente, è una cosa piuttosto delicata, e bisogna sempre pensare ai possibili effetti collaterali di quello che si fa.

3.5 Cercare

La ricerca di uno specifico file è una operazione piuttosto comune, e di conseguenza sono stati sviluppati alcuni comandi molto flessibili per questo scopo.

Il primo gruppo è quello che vi permette di cercare una stringa di caratteri all'interno di uno o più file: `grep` e simili. Per sfruttare fino in fondo il più potente `egrep`, è necessario conoscere le regexp [10] in modo più o meno completo.

Il secondo comando di uso comune sotto Linux è `locate`, che utilizza un database (solitamente generato e aggiornato da un processo processo periodico, configurato in `/etc/crontab` o in una delle directory `/etc/cron.daily`, `/etc/cron.weekly` etc. tramite il comando `updatedb`).

Il comando `locate` è molto veloce, ma purtroppo è poco flessibile: vi consente solo una ricerca sul nome, e non utilizza un completo sistema di regexp. E non bisogna dimenticarsi che le informazioni fornite dal locate risalgono all'ultima esecuzione dell'updatedb. Oltretutto, è raramente disponibile su altri tipi di Unix, ed è quindi un *vizio* a cui bisogna poter rinunciare. Per chi amministra una macchina multi-utente, è importante ricordare che l'updatedb non deve assolutamente essere eseguito da root, ma piuttosto dall'utente nobody, onde evitare che appaiano nel database i contenuti di directory protette in lettura (ad es. le `/mail`).

A sopperire alle carenze del locate, provvede il `find`, che non utilizza un database, ma ricerca direttamente nel filesystem. La conseguenza di questo è un notevole impegno del sistema dei dischi, e quindi dei tempi decisamente più lunghi; in cambio si hanno una serie pressoché infinita di opzioni di ricerca (su tipo del file, proprietario, date, etc) combinabili nei modi più svariati, e l'interessante (ma potenzialmente pericolosa) possibilità di eseguire un comando su ciascuno dei file che hanno passato il vaglio del `find`, tramite l'opzione `-exec`.

Scusatemi, ma devo necessariamente raccomandarvi di utilizzare il `find` per un *test*, senza l'exec, o con un exec *innocuo* come un `-exec echo` ; prima di usare l'exec per qualsiasi operazione pericolosa.

Un altro comando utile per la ricerca, soprattutto per root, è il `which`, che ci indica a quale file eseguibile corrisponde un certo comando:

```
[sb@pcsash sb]$ which less
/usr/bin/less
```

il che risulta molto utile sia per essere sicuri che venga, ad esempio, utilizzata la versione giusta di un comando (nel caso ne esista più di una installata, ad esempio in `/usr/bin` e in `/usr/local/bin`), che per rintracciare la *provenienza* di un comando: ad esempio con la RedHat:

```
[sb@pcsash sb]$ rpm -qf `which less`
less-332-2
```

è un modo molto veloce di risalire al pacchetto di origine, e quindi alla versione, di qualsiasi comando.

[10] REGular EXPressions: sono una estensione del sistema del globbing (vedi la sezione 2.3) in cui valgono corrispondenze fra stringhe con un notevole numero di possibilità, che vede tante applicazioni sotto Unix, e che si ritrova in molte occasioni. Sono anche particolarmente complicate.

3.6 Trasportare

Ci sono vari comandi per spostare file e directory, quella che segue è una lista dei principali:

mkdir Questo è il comando che permette di creare le directory, sempre che abbiate i permessi di scrittura nella directory nella quale volete creare la nuova.

mv Questo è il comando per muovere files da una directory ad un'altra o per rinominarli.

cp Questo è il comando per copiare files da una directory ad un'altra.

mcopy L'equivalente del comando cp da usare quando si opera su file system fat (quelli del dos).

3.7 Danneggiare

Per cancellare i file (e qui bisogna stare attenti, perché non vanno nel cestino e una volta cancellati sono persi)[11], ci sono altri comandi:

rm Cancella i files specificati.

rmdir Elimina le directory specificate.

mdel L'equivalente del comando rm da usare quando si opera su file system fat (quelli del dos).

3.8 Rilassiamoci

Finora abbiamo lavorato parecchio - adesso prendiamo un cd audio, e proviamo:

```
[sb@pcsash sb]$ cdplay
```

e ascoltiamoci della buona musica...

Se non funziona, provate

```
[sb@pcsash sb]$ man cdplay
```

in particolare dove dice

```
cdplay expects to find the device: /dev/cdrom
```

Ancora non funziona? beh, dovete configurare il supporto audio del kernel. Sulla RedHat, provate (compito a casa: quale sarà l'utente che potrà eseguire con qualche successo il comando?):

```
[root@pcsash sb]# soundconfig
```

Altrimenti aspettate il corso avanzato - o attaccate le cuffie al CDROM :-)

Non sento, come dici? funziona? sì? ma solo a tutto volume? beh, questo è l'esercizio per casa di ricerca documentazione ;-)

3.9 Dischi non di sistema

La struttura dei dischi, filesystem nella dizione corretta, in Linux (ma vale anche per molti altri Unix) è estremamente complessa e sicuramente molto differente a quanto si può essere abituati provenendo da altri sistemi operativi come dos e MacOS.

In particolare per poter accedere ad un qualunque disco, floppy, CDROM, zip e altro (ma in verità anche l'harddisk da cui si parte viene montato nelle primissime fasi dell'avvio del sistema), questo deve essere *montato*, cioè *attaccato* alla directory principale, detta root.

Il comando `mount` serve esattamente a questo.

L'argomento è estremamente complesso e per questo viene lasciato ad una lezione apposita, alla quale vi rimando.

[11] a meno di parecchio lavoro di recupero

3.10 Configurare il sistema

I file di configurazione del sistema sono sempre (o quasi) files di testo che si trovano nella directory `/etc`.

3.11 Varie

Come accennato la vera potenza della shell sta nel fatto di poter combinare in una pipeline tanti comandi, ciascuno dei quali esegue un compito elementare, in modo da poter arrivare alla fine ad operazioni molto complesse.

Un breve elenco di alcuni di questi comandi è il seguente:

sort Un piccolo programmino che serve ad ordinare l'input secondo svariati criteri. Molto usato all'interno di script e di pipe.

sed Elaboratore di testi ultra-minimale (non elabora files ma solo *stream*), da non usare direttamente ma solo all'interno di script e di pipe per effettuare modifiche "al volo" all'interno di una pipe.

diff Serve ad indicare le differenze che esistono fra due files di testo. Viene usato dai programmatori, anche questo spesso all'interno di script. In particolare un file prodotto da `diff` può essere dato in pasto ad un altro programma `patch`, che applica le differenze (toglie o mette a seconda della necessità) ad uno dei due file.

cut Serve a tagliare (per l'appunto il nome) files di testo in modi molto doversi. Permette ad esempio di selezionare parti di una riga.

uniq Rimuove linee duplicate da un file, molto utile in combinazione con `sort` per generare liste.

grep Serve a cercare delle parole in uno o più files di testo. Come già fatto vedere ottimo per ricercare files particolari.

Suoi fratelli sono il già citato `egrep` per fare ricerche usando le regexp e `zgrep` per fare ricerche anche su files compressi.

wc conta caratteri, parole e linee di un file.

tr effettua traslazioni o cancellazione di insiemi di caratteri (tipo conversioni maiuscole/minuscole).

3.12 Automazione

Linux, come d'altronde tutti gli Unix, è stato creato da programmatori pensando soprattutto a sé stessi :-))) Il che comporta che per questo tipo di piattaforma esiste una pletora di linguaggi di programmazione di tutti i generi, gusti e apparenza.

Messa così si capisce come mai praticamente è possibile automatizzare qualunque compito grazie alla programmabilità di programmi *normali* come le shell oppure grazie all'uso di linguaggi appositi.

Le opzioni principali sono:

shell script Ogni shell contiene al suo interno un meccanismo che le permette di automatizzare dei compiti scritti dentro a files usando una sintassi specifica.

perl Questo invece è un linguaggio di programmazione vero e proprio nato per manipolare stringhe ma adesso usato per moltissimi altri compiti (anche ad esempio in molti script cgi di tanta moda all'interno di siti web).

3.13 Documentazione

Tutti, e ribadisco proprio tutti, i comandi e in genere i programmi di Linux dispongono sempre un notevole corredo di documentazione, che va sfruttata ed usata maggiormente possibile perché è spesso ben fatta ed esauriente.

La documentazione per i programmi da linea di comando si divide in questi quattro grandi gruppi:

- Pagine man

  Digitando `man comando` si ottiene un testo (spesso lungo) che spiega dettagliatamente tutte le opzioni del comando.

  Tutti i comandi e i programmi hanno una loro pagina man, quasi sempre anche tradotta in italiano.

- Pagine info

  Digitando `info comando` si ottiene invece la pagina info del comando.

  Questo formato era la vecchia versione della documentazione che ha delle caratteristiche molto comode come la possibilità di navigare usando link anche ad altre pagine.

  Purtroppo questo formato non viene ormai molto usato, specialmente dai nuovi programmi e spesso non la si trova tradotta.

- Documentazione in formato testo

  Tutti i programmi e i comandi vengono distribuiti con la documentazione scritta direttamente dagli autori e la si trova nella directory `/usr/doc/nomeprogramma` (oppure nelle nuove distribuzioni `/usr/share/doc/nomeprogramma`).

- Spiegazioni dalla riga di comando

  Digitando `comando --help`, oppure `-h`, oppure `-?`, si ottiene sempre (o quasi) una lista delle opzioni possibili per il tale comando, a volte con una minima spiegazione del significato delle varie opzioni.

  Molto comodo per ricordarsi al volo quale opzioni abbiamo a disposizione.

Normalmente questa documentazione è in normale formato testo ma spesso si trovano anche pagine in html e quasi sempre solo in inglese.

La documentazione sotto Linux è una questione molto complessa perché ne esiste una quantità immane e per questa ragione esiste una lezione apposita alla quale vi rimando ancora.