

Seconda dispensa: la shell e i comandi di linea

[(c)2002 – Jean François Panico]

Motivazioni

Perchè 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, alle soglie del [2000](#), con le moderne [CPU](#), per utilizzare qualcosa di così **retro**, complicato e poco intuitivo come la linea di comando, la modalità testo. Ho quattro 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 100MB 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 diecimila altri, o delle opzioni di configurazione magari inaccessibili se non cambiando chissà quale file .ini o voce del Register. E' sufficiente che applichiate una sola regola: **RTFM - Read The Fucking Manual** . Basta `man comando`.

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 completamete inabili alla tastiera, scrivere pochi caratteri è sicuramente più veloce che trascinare un mouse fra infiniti menu e finestre di dialogo. Ma 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, una 486 con 8MB può fare un ottimo servizio come server Web, e-mail, file server e un sacco di altre cose. Ma se provate a far partire l'interfaccia grafica, diventerà un sistema inutilizzabile...

5. **accessibilità**

Provate a eseguire un programma su un PC a 10.000Km 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 sempre disponibile.

6. sicurezza

Bene, cominciamo così: installate un 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. E potete far partire il sistema con due floppy.

Filosofia

Il S.O. Unix nasce (dal 1969 ai Bell Labs della AT&T) per riutilizzare vecchi, 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 testo). Molti di voi conosceranno la linea di comando del 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 punti fondamentali della filosofia Unix:

- **comandi semplici**
molti comandi, molto specializzati
`wc /etc/passwd`
- **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`
- **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`
- **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 files 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!

Login

```
Red Hat Linux release 7.4 (Renewal)
Kernel 2.5.12 on an i586
login: sb
Password: Cappuccino
Last login: Sat Jun 3 10:12:48 on tty2
Il System Manager ti saluta
```

```
You have mail.  
[sb@pcsash sb]$
```

Questo significa che vi identificate e autenticate con il sistema:

*chi va là ? sono quello che tu chiami sb
parola d'ordine ? Cappuccino*

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`

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

Nota: 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 l'utilizzo.

```
[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 e-mail è un account (con diritti limitati) su un server Unix. Per la strada ne scopriremo il significato.

Prompt e shell

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

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 perchè il prompt non è sempre uguale ?

Dopo l'autenticazione, il sistema passa ad eseguire, con la vostra "identità", un programma specificato nel file `/etc/passwd`. 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) 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 - il cui primo comandamento è "non avrai altra shell al di fuori di `command.com`". Qualcuno di voi forse conoscerà 4DOS, e saprà quindi che si può infrangere quel comandamento. 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; ma io vi parlerò di **bash**, che per Linux è la shell di default, e probabilmente la più potente e flessibile.

Nota: 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 "Bourne" si pronuncia (quasi) come "born" : "la Bourne Shell rinata".

Nota: le altre shell si raggruppano in due categorie:

Bourne shell e derivate:

- sh 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. Usata talvolta nei dischetti di installazione o recovery, può essere utile per gli script perchè più veloce di bash. Funzioni minime.
- ksh - Korn Shell
ksh 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 altre Unix su cui 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 che esista una opzione per fargli preparare il caffè.
Dalla man page: BUGS: It's too big and too slow.

C shell e derivate:

- csh - C shell
utilizza una sintassi derivata dal linguaggio C.
In Linux è sostituita da tcsh.
- tcsh -
introduce history e editing di linea.

Quale 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 altre 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 - molto noioso.

Il prompt, quindi, è il programma shell che aspetta 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:

1. 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**
2. pipeline:
redirezionare 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.

3. alias
abbreviazioni definibili per comandi molto usati
4. filename globbing
per operare su gruppi di file con nomi simili:
[sb@pcsash sb]\$ ls prova*1998.txt
5. variable expansion
[sb@pcsash sb]\$ Saluto=Ciao
[sb@pcsash sb]\$ echo "\$Saluto \$USER, sei su \$HOSTNAME"
Ciao sb, sei su pcsash.sash.lan
6. 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
7. history expansion
per richiamare righe di comando precedenti:
[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
8. variabili di ambiente
la shell si ricorda e vi permette di modificare certe impostazioni usate dai programmi: \$USER, \$HOME, \$TERM, \$DISPLAY, \$PATH e altre; permette di avere impostazioni valide per tutti gli utenti (in /etc/profile) e/o per un singolo utente.
9. controllo di esecuzione:
processi in foreground e background, identificativo di un processo, blocco di un processo, riavvio di un processo, terminazione di un processo: & , %n, ^C, ^Z, fg, bg, kill
10. scripting
i "batch file" del DOS - ipervitaminizzati.

e altre piccole cose, come ~ per significare la directory home dell'utente o ~<username> per la home di un'altro utente.

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 (serve quando vi capita spesso di avere 10 finestre terminale aperte su diverse macchine, con diversi id... beh, magari per un po' non vi capiterà). 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.

User e Superuser

una questione psicanalitica

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**.

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.

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

è sufficiente a cancellare qualsiasi file sulle vostre partizioni... e altri possono facilmente fare polpette di tutto l'hard disk.

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...

Dove siamo, cosa abbiamo intorno

Il nostro prompt, subito dopo il login,

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

ci dice che siamo in una directory di nome sb. Quale ? ce lo dice

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

```
/home/sb
```

pwd - Present Working Directory

In un'altra lezione vi spiegheremo 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**). Se per caso qualcuno si chiedeva perchè gli indirizzi HTTP usano le / (slash), adesso lo sa: perchè il World Wide Web è nato (al CERN, nel 1989) su **NeXt**, un sistema derivato, di fondo, da Unix.

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
```

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. "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.

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

proviamo a spostarci:

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

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

cd - Change Directory

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
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) torniamo nella directory di partenza (home):

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

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
```

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 mtool hanno diverse peculiarità, e comportamenti leggermente diversi dagli strumenti standard - fate attenzione.

Curiosare

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

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

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
```

less - più di more che è un' altro comando per "scorrere lentamente" il contenuto di un file

Oh, adesso possiamo scorrerlo come vogliamo, tornando indietro, persino cercando delle parole

/end

e molto altro

h

e per finire,

q

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
```

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: proviamo a sciverci 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...

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 :

```
[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 (cominciano per \$), e bash ne ha sostituito il valore nell'espressione. 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 redirectione 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 redirectione.
- con `cmd_a 2>&1 | cmd_b` si passa anche l'error output nella pipe - vedi sopra per i problemi

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.

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 `regex` - **REGular EXPressions**. Le regex sono una estensione del sistema del globbing - una

corrispondenza fra stringhe non un notevole numero di possibilità, che vede tante applicazioni sotto Unix, e si ritrova in molte occasioni, 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 periodico, configurato in `/etc/crontab` o in una delle directory `/etc/cron.daily`, `/etc/cron.weekly` etc. tramite il comando `updatedb`). `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 pressochè 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 es. in `/usr/bin` e in `/usr/local/bin`), che per rintracciare la "provenienza" di un comando: ad es. 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.

trasportare

- `mkdir`
- `mv`
- `cp`
- `mcopy`

danneggiare

- `rm`
- `rmdir`
- `mdel`

Rilassiamoci

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

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

e ascoltatiamoci della buona musica...

Se non funziona, provate

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

in particolare dove dice

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

Ancora non funziona ? beh, dovete configurare il supporto audio del kernel. Sulla RedHat, provate:

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

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

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

dischi non Linux

- mount

cambiare un file di configurazione

gli editor di testo

- vi
- emacs e jed
- pico
- joe

varie

- sort
- sed
- diff
- cut

automazione

- shell script
- perl